PostGIS 3.7.0rc1 Manual

Contents

1 Introduction **1**
1.1 What Is PostGIS? . . . 1
1.2 What Is in the PostGIS Distribution? . . . 1
1.3 The PostGIS Project Family . . . 2
1.4 How to Use This Manual . . . 2
1.5 Standards Support . . . 2
1.6 Project and Community . . . 3

2 PostGIS Installation **4**
2.1 Short Version . . . 4
2.2 Compiling and Install from Source . . . 4
2.2.1 Getting the Source . . . 5
2.2.2 Install Requirements . . . 5
2.2.3 Build configuration . . . 6
2.2.4 Building . . . 8
2.2.5 Building PostGIS Extensions and Deploying them . . . 8
2.2.6 Testing . . . 10
2.2.7 Installation . . . 13
2.3 Common Problems during installation . . . 14

3 PostGIS Administration **15**
3.1 Performance Tuning . . . 15
3.1.1 Startup . . . 15
3.1.2 Runtime . . . 16
3.2 Configuring raster support . . . 16
3.3 Creating spatial databases . . . 17
3.3.1 Spatially enable database using EXTENSION . . . 17
3.3.2 Spatially enable database without using EXTENSION (discouraged) . . . 18
3.4 Upgrading spatial databases . . . 18
3.4.1 Soft upgrade . . . 18
3.4.1.1 Soft upgrade using extensions . . . 19
3.4.1.2 Soft upgrade without extensions . . . 20
3.4.1.3 Hard upgrade . . . 20

4 Data Management **22**

4.1 Spatial Data Model . . . 22
4.1.1 OGC Geometry . . . 22
4.1.1.1 Point . . . 23
4.1.1.2 LineString . . . 23
4.1.1.3 LinearRing . . . 23
4.1.1.4 Polygon . . . 23
4.1.1.5 MultiPoint . . . 23
4.1.1.6 MultiLineString . . . 24
4.1.1.7 MultiPolygon . . . 24
4.1.1.8 GeometryCollection . . . 24
4.1.1.9 PolyhedralSurface . . . 24
4.1.1.10 Triangle . . . 24
4.1.1.11 TIN . . . 25
4.1.2 SQL/MM Part 3 - Curves . . . 25
4.1.2.1 CircularString . . . 25
4.1.2.2 CompoundCurve . . . 25
4.1.2.3 CurvePolygon . . . 25
4.1.2.4 MultiCurve . . . 26
4.1.2.5 MultiSurface . . . 26
4.1.3 WKT and WKB . . . 26
4.2 Geometry Data Type . . . 27
4.2.1 PostGIS EWKB and EWKT . . . 28
4.3 Geography Data Type . . . 29
4.3.1 Creating Geography Tables . . . 30
4.3.2 Using Geography Tables . . . 31
4.3.3 When to use the Geography data type . . . 33
4.3.4 Geography Advanced FAQ . . . 33
4.4 Geometry Validation . . . 34
4.4.1 Simple Geometry . . . 34
4.4.2 Valid Geometry . . . 36
4.4.3 Managing Validity . . . 40
4.5 Spatial Reference Systems . . . 42
4.5.1 SPATIAL_REF_SYS Table . . . 43
4.5.2 User-Defined Spatial Reference Systems . . . 44
4.6 Spatial Tables . . . 44
4.6.1 Creating a Spatial Table . . . 44
4.6.2 GEOMETRY_COLUMNS View . . . 45
4.6.3 Manually Registering Geometry Columns . . . 46

4.7 Loading Spatial Data . . . 49
4.7.1 Using SQL to Load Data . . . 49
4.7.2 Using the Shapefile Loader . . . 49
4.8 Extracting Spatial Data . . . 51
4.8.1 Using SQL to Extract Data . . . 51
4.8.2 Using the Shapefile Dumper . . . 52
4.9 Spatial Indexes . . . 53
4.9.1 GiST Indexes . . . 53
4.9.2 BRIN Indexes . . . 54
4.9.3 SP-GiST Indexes . . . 56
4.9.4 Tuning Index Usage . . . 57

5 Spatial Queries **58**
5.1 Determining Spatial Relationships . . . 58
5.1.1 Dimensionally Extended 9-Intersection Model . . . 58
5.1.2 Named Spatial Relationships . . . 62
5.1.3 General Spatial Relationships . . . 62
5.2 Using Spatial Indexes . . . 64
5.3 Examples of Spatial SQL . . . 65

6 Performance Tips **68**
6.1 Small tables of large geometries . . . 68
6.1.1 Problem description . . . 68
6.1.2 Workarounds . . . 68
6.2 CLUSTERing on geometry indices . . . 69
6.3 Avoiding dimension conversion . . . 69

7 PostGIS Reference **70**
7.1 PostGIS Geometry/Geography/Box Data Types . . . 70
7.1.1 box2d . . . 70
7.1.2 box3d . . . 71
7.1.3 geometry . . . 71
7.1.4 geometry_dump . . . 72
7.1.5 geography . . . 72
7.2 Table Management Functions . . . 72
7.2.1 AddGeometryColumn . . . 72
7.2.2 DropGeometryColumn . . . 75
7.2.3 DropGeometryTable . . . 76
7.2.4 Find_SRID . . . 77
7.2.5 Populate_Geometry_Columns . . . 77

7.2.6 UpdateGeometrySRID . . . 79
7.3 Geometry Constructors . . . 80
7.3.1 ST_Collect . . . 80
7.3.2 ST_LineFromMultiPoint . . . 84
7.3.3 ST_MakeEnvelope . . . 85
7.3.4 ST_MakeLine . . . 86
7.3.5 ST_MakePoint . . . 90
7.3.6 ST_MakePointM . . . 92
7.3.7 ST_MakePolygon . . . 94
7.3.8 ST_Point . . . 98
7.3.9 ST_PointZ . . . 99
7.3.10 ST_PointM . . . 100
7.3.11 ST_PointZM . . . 100
7.3.12 ST_Polygon . . . 101
7.3.13 ST_TileEnvelope . . . 103
7.3.14 ST_HexagonGrid . . . 105
7.3.15 ST_Hexagon . . . 110
7.3.16 ST_SquareGrid . . . 111
7.3.17 ST_Square . . . 113
7.3.18 ST_Letters . . . 114
7.3.19 ST_MakeNurbsCurve . . . 116
7.4 Geometry Accessors . . . 119
7.4.1 GeometryType . . . 119
7.4.2 ST_Boundary . . . 121
7.4.3 ST_BoundingDiagonal . . . 126
7.4.4 ST_CoordDim . . . 128
7.4.5 ST_Dimension . . . 129
7.4.6 ST_Dump . . . 131
7.4.7 ST_DumpPoints . . . 134
7.4.8 ST_DumpSegments . . . 139
7.4.9 ST_DumpRings . . . 142
7.4.10 ST_EndPoint . . . 144
7.4.11 ST_Envelope . . . 148
7.4.12 ST_ExteriorRing . . . 152
7.4.13 ST_GeometryN . . . 155
7.4.14 ST_GeometryType . . . 158
7.4.15 ST_HasArc . . . 160
7.4.16 ST_InteriorRingN . . . 161
7.4.17 ST_NumCurves . . . 162

7.4.18 ST_CurveN . . . 163
7.4.19 ST_IsClosed . . . 165
7.4.20 ST_IsCollection . . . 169
7.4.21 ST_IsEmpty . . . 171
7.4.22 ST_IsPolygonCCW . . . 173
7.4.23 ST_IsPolygonCW . . . 174
7.4.24 ST_IsRing . . . 175
7.4.25 ST_IsSimple . . . 177
7.4.26 ST_M . . . 179
7.4.27 ST_MemSize . . . 180
7.4.28 ST_NDims . . . 181
7.4.29 ST_NPoints . . . 182
7.4.30 ST_NRings . . . 183
7.4.31 ST_NumGeometries . . . 184
7.4.32 ST_NumInteriorRings . . . 185
7.4.33 ST_NumInteriorRing . . . 186
7.4.34 ST_NumPatches . . . 186
7.4.35 ST_NumPoints . . . 187
7.4.36 ST_PatchN . . . 188
7.4.37 ST_PointN . . . 189
7.4.38 ST_Points . . . 192
7.4.39 ST_StartPoint . . . 193
7.4.40 ST_StartM . . . 198
7.4.41 ST_EndM . . . 201
7.4.42 ST_SetStartM . . . 204
7.4.43 ST_SetEndM . . . 207
7.4.44 ST_Summary . . . 210
7.4.45 ST_X . . . 211
7.4.46 ST_Y . . . 212
7.4.47 ST_Z . . . 213
7.4.48 ST_ZMFlag . . . 214
7.4.49 ST_HasZ . . . 215
7.4.50 ST_HasM . . . 216
7.4.51 ST_ControlPoints . . . 218
7.4.52 ST_Degree . . . 219
7.4.53 ST_Weights . . . 221
7.4.54 ST_Knots . . . 222
7.4.55 ST_NumControlPoints . . . 224
7.4.56 ST_NurbsCurveIsRational . . . 226

7.4.57 ST_IsNurbsCurve . . . 228
7.4.58 ST_Evaluate . . . 230
7.4.59 ST_NurbsToLineString . . . 231
7.5 Geometry Editors . . . 233
7.5.1 ST_AddPoint . . . 233
7.5.2 ST_CollectionExtract . . . 234
7.5.3 ST_CollectionHomogenize . . . 237
7.5.4 ST_CurveToLine . . . 241
7.5.5 ST_Scroll . . . 245
7.5.6 ST_FlipCoordinates . . . 246
7.5.7 ST_Force2D . . . 247
7.5.8 ST_Force3D . . . 249
7.5.9 ST_Force3DZ . . . 251
7.5.10 ST_Force3DM . . . 253
7.5.11 ST_Force4D . . . 255
7.5.12 ST_ForceCollection . . . 257
7.5.13 ST_ForceCurve . . . 260
7.5.14 ST_ForcePolygonCCW . . . 261
7.5.15 ST_ForcePolygonCW . . . 262
7.5.16 ST_ForceSFS . . . 264
7.5.17 ST_ForceRHR . . . 265
7.5.18 ST_LineExtend . . . 266
7.5.19 ST_LineToCurve . . . 267
7.5.20 ST_Multi . . . 269
7.5.21 ST_Normalize . . . 270
7.5.22 ST_Project . . . 271
7.5.23 ST_QuantizeCoordinates . . . 273
7.5.24 ST_RemovePoint . . . 275
7.5.25 ST_RemoveRepeatedPoints . . . 275
7.5.26 ST_RemoveIrrelevantPointsForView . . . 279
7.5.27 ST_RemoveSmallParts . . . 283
7.5.28 ST_Reverse . . . 285
7.5.29 ST_Segmentize . . . 286
7.5.30 ST_SetPoint . . . 289
7.5.31 ST_ShiftLongitude . . . 292
7.5.32 ST_WrapX . . . 294
7.5.33 ST_SnapToGrid . . . 295
7.5.34 ST_Snap . . . 298
7.5.35 ST_SwapOrdinates . . . 302

7.6 Geometry Validation . . . 303
7.6.1 ST_IsValid . . . 303
7.6.2 ST_IsValidDetail . . . 306
7.6.3 ST_IsValidReason . . . 307
7.6.4 ST_MakeValid . . . 310
7.7 Spatial Reference System Functions . . . 313
7.7.1 ST_InverseTransformPipeline . . . 313
7.7.2 ST_SetSRID . . . 315
7.7.3 ST_SRID . . . 317
7.7.4 ST_Transform . . . 318
7.7.5 ST_TransformPipeline . . . 322
7.7.6 postgis_srs_codes . . . 325
7.7.7 postgis_srs . . . 326
7.7.8 postgis_srs_all . . . 327
7.7.9 postgis_srs_search . . . 327
7.8 Geometry Input . . . 329
7.8.1 Well-Known Text (WKT) . . . 329
7.8.1.1 ST_BdPolyFromText . . . 329
7.8.1.2 ST_BdMPolyFromText . . . 329
7.8.1.3 ST_GeogFromText . . . 330
7.8.1.4 ST_GeographyFromText . . . 331
7.8.1.5 ST_GeomCollFromText . . . 331
7.8.1.6 ST_GeomFromEWKT . . . 332
7.8.1.7 ST_GeomFromMARC21 . . . 335
7.8.1.8 ST_GeometryFromText . . . 338
7.8.1.9 ST_GeomFromText . . . 339
7.8.1.10 ST_LineFromText . . . 344
7.8.1.11 ST_MLineFromText . . . 345
7.8.1.12 ST_MPointFromText . . . 346
7.8.1.13 ST_MPolyFromText . . . 346
7.8.1.14 ST_PointFromText . . . 348
7.8.1.15 ST_PolygonFromText . . . 350
7.8.1.16 ST_WKTToSQL . . . 352
7.8.2 Well-Known Binary (WKB) . . . 352
7.8.2.1 ST_GeogFromWKB . . . 352
7.8.2.2 ST_GeomFromEWKB . . . 353
7.8.2.3 ST_GeomFromWKB . . . 355
7.8.2.4 ST_LineFromWKB . . . 357
7.8.2.5 ST_LinestringFromWKB . . . 358

7.8.2.6 ST_PointFromWKB . . . 360
7.8.2.7 ST_WKBToSQL . . . 361
7.8.3 Other Formats . . . 361
7.8.3.1 ST_Box2dFromGeoHash . . . 361
7.8.3.2 ST_GeomFromGeoHash . . . 362
7.8.3.3 ST_GeomFromGML . . . 363
7.8.3.4 ST_GeomFromGeoJSON . . . 367
7.8.3.5 ST_GeomFromKML . . . 369
7.8.3.6 ST_GeomFromTWKB . . . 371
7.8.3.7 ST_GMLToSQL . . . 372
7.8.3.8 ST_LineFromEncodedPolyline . . . 373
7.8.3.9 ST_PointFromGeoHash . . . 374
7.8.3.10 ST_FromFlatGeobufToTable . . . 377
7.8.3.11 ST_FromFlatGeobuf . . . 377
7.9 Geometry Output . . . 377
7.9.1 Well-Known Text (WKT) . . . 377
7.9.1.1 ST_AsEWKT . . . 377
7.9.1.2 ST_AsText . . . 380
7.9.2 Well-Known Binary (WKB) . . . 382
7.9.2.1 ST_AsBinary . . . 382
7.9.2.2 ST_AsEWKB . . . 383
7.9.2.3 ST_AsHEXEWKB . . . 384
7.9.3 Other Formats . . . 385
7.9.3.1 ST_AsEncodedPolyline . . . 385
7.9.3.2 ST_AsFlatGeobuf . . . 387
7.9.3.3 ST_AsGeobuf . . . 387
7.9.3.4 ST_AsGeoJSON . . . 388
7.9.3.5 ST_AsGML . . . 390
7.9.3.6 ST_AsKML . . . 393
7.9.3.7 ST_AsLatLonText . . . 394
7.9.3.8 ST_AsMARC21 . . . 396
7.9.3.9 ST_AsMVTGeom . . . 398
7.9.3.10 ST_AsMVT . . . 400
7.9.3.11 ST_AsSVG . . . 402
7.9.3.12 ST_AsTWKB . . . 405
7.9.3.13 ST_AsX3D . . . 406
7.9.3.14 ST_GeoHash . . . 410
7.10 Operators . . . 411
7.10.1 Bounding Box Operators . . . 411

7.10.1.1 && . . . 411
7.10.1.2 &&(geometry,box2df) . . . 412
7.10.1.3 &&(box2df,geometry) . . . 413
7.10.1.4 &&(box2df,box2df) . . . 414
7.10.1.5 &&& . . . 415
7.10.1.6 &&&(geometry,gidx) . . . 417
7.10.1.7 &&&(gidx,geometry) . . . 418
7.10.1.8 &&&(gidx,gidx) . . . 419
7.10.1.9 &< . . . 420
7.10.1.10 &<| . . . 421
7.10.1.11 &> . . . 422
7.10.1.12 << . . . 424
7.10.1.13 <<| . . . 425
7.10.1.14 = . . . 426
7.10.1.15 >> . . . 429
7.10.1.16 @ . . . 431
7.10.1.17 @(geometry,box2df) . . . 432
7.10.1.18 @(box2df,geometry) . . . 433
7.10.1.19 @(box2df,box2df) . . . 434
7.10.1.20 |&> . . . 434
7.10.1.21 |>> . . . 436
7.10.1.22 ~ . . . 437
7.10.1.23 ~(geometry,box2df) . . . 438
7.10.1.24 ~(box2df,geometry) . . . 439
7.10.1.25 ~(box2df,box2df) . . . 440
7.10.1.26 ~= . . . 441
7.10.2 Distance Operators . . . 442
7.10.2.1 <-> . . . 442
7.10.2.2 |=| . . . 444
7.10.2.3 <#> . . . 445
7.10.2.4 <<->> . . . 446
7.11 Spatial Relationships . . . 447
7.11.1 Topological Relationships . . . 447
7.11.1.1 ST_3DIntersects . . . 447
7.11.1.2 ST_Contains . . . 449
7.11.1.3 ST_ContainsProperly . . . 460
7.11.1.4 ST_CoveredBy . . . 463
7.11.1.5 ST_Covers . . . 465
7.11.1.6 ST_Crosses . . . 467

7.11.1.7 ST_Disjoint . . . 470
7.11.1.8 ST_Equals . . . 473
7.11.1.9 ST_Intersects . . . 474
7.11.1.10 ST_LineCrossingDirection . . . 477
7.11.1.11 ST_OrderingEquals . . . 480
7.11.1.12 ST_Overlaps . . . 483
7.11.1.13 ST_Relate . . . 489
7.11.1.14 ST_RelateMatch . . . 492
7.11.1.15 ST_Touches . . . 494
7.11.1.16 ST_Within . . . 501
7.11.2 Distance Relationships . . . 504
7.11.2.1 ST_3DDWithin . . . 504
7.11.2.2 ST_3DDFullyWithin . . . 505
7.11.2.3 ST_DFullyWithin . . . 507
7.11.2.4 ST_DWithin . . . 508
7.11.2.5 ST_PointInsideCircle . . . 510
7.12 Measurement Functions . . . 512
7.12.1 ST_Area . . . 512
7.12.2 ST_Azimuth . . . 515
7.12.3 ST_Angle . . . 517
7.12.4 ST_ClosestPoint . . . 520
7.12.5 ST_3DClosestPoint . . . 522
7.12.6 ST_Distance . . . 525
7.12.7 ST_3DDistance . . . 527
7.12.8 ST_DistanceSphere . . . 528
7.12.9 ST_DistanceSpheroid . . . 530
7.12.10 ST_FrechetDistance . . . 531
7.12.11 ST_HausdorffDistance . . . 533
7.12.12 ST_Length . . . 535
7.12.13 ST_Length2D . . . 538
7.12.14 ST_3DLength . . . 538
7.12.15 ST_LengthSpheroid . . . 539
7.12.16 ST_LongestLine . . . 542
7.12.17 ST_3DLongestLine . . . 545
7.12.18 ST_MaxDistance . . . 548
7.12.19 ST_3DMaxDistance . . . 551
7.12.20 ST_MinimumClearance . . . 551
7.12.21 ST_MinimumClearanceLine . . . 553
7.12.22 ST_Perimeter . . . 554

7.12.23 ST_Perimeter2D . . . 558
7.12.24 ST_3DPerimeter . . . 558
7.12.25 ST_ShortestLine . . . 560
7.12.26 ST_3DShortestLine . . . 561
7.13 Overlay Functions . . . 564
7.13.1 ST_ClipByBox2D . . . 564
7.13.2 ST_Difference . . . 565
7.13.3 ST_Intersection . . . 567
7.13.4 ST_MemUnion . . . 570
7.13.5 ST_Node . . . 571
7.13.6 ST_Split . . . 573
7.13.7 ST_Subdivide . . . 576
7.13.8 ST_SymDifference . . . 579
7.13.9 ST_UnaryUnion . . . 581
7.13.10 ST_Union . . . 581
7.14 Geometry Processing . . . 587
7.14.1 ST_Buffer . . . 587
7.14.2 ST_BuildArea . . . 596
7.14.3 ST_Centroid . . . 599
7.14.4 ST_ChaikinSmoothing . . . 604
7.14.5 ST_CatmullRomSmoothing . . . 608
7.14.6 ST_ConcaveHull . . . 610
7.14.7 ST_ConvexHull . . . 614
7.14.8 ST_DelaunayTriangles . . . 616
7.14.9 ST_FilterByM . . . 619
7.14.10 ST_GeneratePoints . . . 621
7.14.11 ST_GeometricMedian . . . 622
7.14.12 ST_LineMerge . . . 623
7.14.13 ST_MaximumInscribedCircle . . . 628
7.14.14 ST_LargestEmptyCircle . . . 630
7.14.15 ST_MinimumBoundingCircle . . . 632
7.14.16 ST_MinimumBoundingRadius . . . 634
7.14.17 ST_OrientedEnvelope . . . 635
7.14.18 ST_OffsetCurve . . . 637
7.14.19 ST_PointOnSurface . . . 644
7.14.20 ST_Polygonize . . . 649
7.14.21 ST_ReducePrecision . . . 652
7.14.22 ST_SharedPaths . . . 656
7.14.23 ST_Simplify . . . 658

7.14.24 ST_SimplifyPreserveTopology . . . 661
7.14.25 ST_SimplifyPolygonHull . . . 664
7.14.26 ST_SimplifyVW . . . 667
7.14.27 ST_SetEffectiveArea . . . 670
7.14.28 ST_TriangulatePolygon . . . 672
7.14.29 ST_VoronoiLines . . . 675
7.14.30 ST_VoronoiPolygons . . . 676
7.15 Coverages . . . 678
7.15.1 ST_CoverageInvalidEdges . . . 678
7.15.2 ST_CoverageClean . . . 680
7.15.3 ST_CoverageSimplify . . . 681
7.15.4 ST_CoverageEdges . . . 683
7.15.5 ST_CoverageUnion . . . 684
7.16 Affine Transformations . . . 685
7.16.1 ST_Affine . . . 685
7.16.2 ST_Rotate . . . 688
7.16.3 ST_RotateX . . . 691
7.16.4 ST_RotateY . . . 692
7.16.5 ST_RotateZ . . . 693
7.16.6 ST_Scale . . . 695
7.16.7 ST_Translate . . . 699
7.16.8 ST_TransScale . . . 702
7.17 Clustering Functions . . . 704
7.17.1 ST_ClusterDBSCAN . . . 704
7.17.2 ST_ClusterIntersecting . . . 706
7.17.3 ST_ClusterIntersectingWin . . . 707
7.17.4 ST_ClusterRelateWin . . . 709
7.17.5 ST_MinimumSpanningTree . . . 710
7.17.6 ST_ClusterKMeans . . . 711
7.17.7 ST_ClusterWithin . . . 716
7.17.8 ST_ClusterWithinWin . . . 717
7.18 Bounding Box Functions . . . 719
7.18.1 Box2D . . . 719
7.18.2 Box3D . . . 721
7.18.3 ST_EstimatedExtent . . . 723
7.18.4 ST_Expand . . . 724
7.18.5 ST_Extent . . . 726
7.18.6 ST_3DExtent . . . 728
7.18.7 ST_MakeBox2D . . . 729

7.18.8 ST_3DMakeBox . . . 730
7.18.9 ST_XMax . . . 730
7.18.10 ST_XMin . . . 732
7.18.11 ST_YMax . . . 734
7.18.12 ST_YMin . . . 736
7.18.13 ST_ZMax . . . 738
7.18.14 ST_ZMin . . . 740
7.18.15 ST_XSize . . . 742
7.18.16 ST_YSize . . . 744
7.18.17 ST_ZSize . . . 745
7.18.18 ST_MMin . . . 747
7.18.19 ST_MMax . . . 747
7.18.20 ST_MSize . . . 748
7.19 Linear Referencing . . . 749
7.19.1 ST_LineInterpolatePoint . . . 749
7.19.2 ST_3DLineInterpolatePoint . . . 753
7.19.3 ST_LineInterpolatePoints . . . 754
7.19.4 ST_LineLocatePoint . . . 755
7.19.5 ST_LineSubstring . . . 758
7.19.6 ST_LocateAlong . . . 764
7.19.7 ST_LocateBetween . . . 765
7.19.8 ST_LocateBetweenElevations . . . 767
7.19.9 ST_InterpolatePoint . . . 769
7.19.10 ST_3DInterpolatePoint . . . 770
7.19.11 ST_AddMeasure . . . 772
7.20 Trajectory Functions . . . 775
7.20.1 ST_IsValidTrajectory . . . 775
7.20.2 ST_ClosestPointOfApproach . . . 776
7.20.3 ST_DistanceCPA . . . 778
7.20.4 ST_CPAWithin . . . 779
7.21 Version Functions . . . 780
7.21.1 PostGIS_Extensions_Upgrade . . . 780
7.21.2 PostGIS_Full_Version . . . 781
7.21.3 PostGIS_GEOS_Version . . . 782
7.21.4 PostGIS_GEOS_Compiled_Version . . . 782
7.21.5 PostGIS_Liblwgeom_Version . . . 783
7.21.6 PostGIS_LibXML_Version . . . 783
7.21.7 PostGIS_LibJSON_Version . . . 784
7.21.8 PostGIS_Lib_Build_Date . . . 784

7.21.9 PostGIS_Lib_Version . . . 785
7.21.10 PostGIS_PROJ_Version . . . 785
7.21.11 PostGIS_PROJ_Compiled_Version . . . 786
7.21.12 PostGIS_Wagyu_Version . . . 786
7.21.13 PostGIS_Scripts_Build_Date . . . 787
7.21.14 PostGIS_Scripts_Installed . . . 787
7.21.15 PostGIS_Scripts_Released . . . 788
7.21.16 PostGIS_Version . . . 789
7.22 Grand Unified Custom Variables (GUCs) . . . 789
7.22.1 postgis.gdal_datapath . . . 789
7.22.2 postgis.gdal_enabled_drivers . . . 790
7.22.3 postgis.enable_outdb_rasters . . . 792
7.22.4 postgis.gdal_vsi_options . . . 793
7.22.5 postgis.gdal_cpl_debug . . . 793
7.23 Troubleshooting Functions . . . 794
7.23.1 PostGIS_AddBBox . . . 794
7.23.2 PostGIS_DropBBox . . . 794
7.23.3 PostGIS_HasBBox . . . 795

8 SFCGAL Functions Reference 797
8.1 SFCGAL Management Functions . . . 797
8.1.1 postgis_sfcgal_version . . . 797
8.1.2 postgis_sfcgal_full_version . . . 797
8.2 SFCGAL Accessors and Setters . . . 798
8.2.1 CG_ForceLHR . . . 798
8.2.2 CG_IsPlanar . . . 799
8.2.3 CG_IsSolid . . . 799
8.2.4 CG_MakeSolid . . . 801
8.2.5 CG_Orientation . . . 801
8.2.6 CG_Area . . . 802
8.2.7 CG_3DArea . . . 803
8.2.8 CG_Volume . . . 804
8.3 SFCGAL Processing and Relationship Functions . . . 805
8.3.1 CG_Intersection . . . 805
8.3.2 CG_Intersects . . . 806
8.3.3 CG_3DIntersects . . . 808
8.3.4 CG_Difference . . . 809
8.3.5 CG_3DDifference . . . 811
8.3.6 CG_Distance . . . 812

8.3.7 CG_3DDistance . . . 813
8.3.8 CG_3DConvexHull . . . 814
8.3.9 CG_3DIntersection . . . 817
8.3.10 CG_Union . . . 820
8.3.11 CG_3DUnion . . . 821
8.3.12 CG_AlphaShape . . . 823
8.3.13 CG_ApproxConvexPartition . . . 826
8.3.14 CG_ApproximateMedialAxis . . . 827
8.3.15 CG_GenerateFlatRoof . . . 830
8.3.16 CG_GenerateHippedRoof . . . 831
8.3.17 CG_GenerateGableRoof . . . 832
8.3.18 CG_GenerateSkillionRoof . . . 833
8.3.19 CG_GenerateRoof . . . 835
8.3.20 CG_PolygonRepair . . . 836
8.3.21 CG_ConstrainedDelaunayTriangles . . . 838
8.3.22 CG_Extrude . . . 840
8.3.23 CG_ExtrudeStraightSkeleton . . . 842
8.3.24 CG_GreeneApproxConvexPartition . . . 843
8.3.25 CG_MinkowskiSum . . . 845
8.3.26 CG_OptimalAlphaShape . . . 847
8.3.27 CG_OptimalConvexPartition . . . 849
8.3.28 CG_StraightSkeleton . . . 850
8.3.29 CG_Tesselate . . . 852
8.3.30 CG_Triangulate . . . 855
8.3.31 CG_Visibility . . . 856
8.3.32 CG_YMonotonePartition . . . 858
8.3.33 CG_StraightSkeletonPartition . . . 859
8.3.34 CG_3DBuffer . . . 861
8.3.35 CG_Rotate . . . 864
8.3.36 CG_2DRotate . . . 865
8.3.37 CG_3DRotate . . . 866
8.3.38 CG_RotateX . . . 867
8.3.39 CG_RotateY . . . 868
8.3.40 CG_RotateZ . . . 869
8.3.41 CG_Scale . . . 870
8.3.42 CG_3DScale . . . 871
8.3.43 CG_3DScaleAroundCenter . . . 872
8.3.44 CG_Translate . . . 873
8.3.45 CG_3DTranslate . . . 874

8.3.46 CG_Simplify . . . 875
8.3.47 CG_3DAlphaWrapping . . . 879
8.3.48 CG_NurbsCurveInterpolate . . . 880
8.3.49 CG_NurbsCurveApproximate . . . 882
8.3.50 CG_NurbsCurveDerivative . . . 884
8.4 Deprecated SFCGAL Function Aliases . . . 888
8.4.1 ST_3DIntersection . . . 888
8.4.2 ST_3DDifference . . . 889
8.4.3 ST_3DUnion . . . 889
8.4.4 ST_Tesselate . . . 889
8.4.5 ST_3DArea . . . 890
8.4.6 ST_Extrude . . . 890
8.4.7 ST_ForceLHR . . . 891
8.4.8 ST_Orientation . . . 891
8.4.9 ST_MinkowskiSum . . . 891
8.4.10 ST_StraightSkeleton . . . 892
8.4.11 ST_ApproximateMedialAxis . . . 892
8.4.12 ST_IsPlanar . . . 893
8.4.13 ST_Volume . . . 893
8.4.14 ST_MakeSolid . . . 894
8.4.15 ST_IsSolid . . . 894
8.4.16 ST_ConstrainedDelaunayTriangles . . . 895
8.4.17 ST_3DConvexHull . . . 895
8.4.18 ST_AlphaShape . . . 895
8.4.19 ST_OptimalAlphaShape . . . 896

9 Topology **897**
9.1 Building Faces from Linework . . . 897
9.2 Topology Primitive Tables . . . 899
9.3 Topology Types . . . 899
9.3.1 getfaceedges_returntype . . . 899
9.3.2 TopoGeometry . . . 900
9.3.3 validatetopology_returntype . . . 900
9.4 Topology Domains . . . 901
9.4.1 TopoElement . . . 901
9.4.2 TopoElementArray . . . 902
9.5 Topology and TopoGeometry Management . . . 903
9.5.1 AddTopoGeometryColumn . . . 903
9.5.2 RenameTopoGeometryColumn . . . 904

9.5.3 DropTopology . . . 904
9.5.4 RenameTopology . . . 905
9.5.5 DropTopoGeometryColumn . . . 905
9.5.6 FixCorruptTopoGeometryColumn . . . 906
9.5.7 Populate_Topology_Layer . . . 906
9.5.8 TopologySummary . . . 908
9.5.9 ValidateTopology . . . 909
9.5.10 ValidateTopologyRelation . . . 911
9.5.11 ValidateTopologyPrecision . . . 911
9.5.12 MakeTopologyPrecise . . . 912
9.5.13 FindTopology . . . 913
9.5.14 FindLayer . . . 914
9.5.15 TotalTopologySize . . . 914
9.5.16 UpgradeTopology . . . 915
9.6 Topology Statistics Management . . . 915
9.7 Topology Constructors . . . 916
9.7.1 CreateTopology . . . 916
9.7.2 CopyTopology . . . 917
9.7.3 ST_InitTopoGeo . . . 917
9.7.4 ST_CreateTopoGeo . . . 918
9.7.5 TopoGeo_AddPoint . . . 919
9.7.6 TopoGeo_AddLineString . . . 919
9.7.7 TopoGeo_AddPolygon . . . 920
9.7.8 TopoGeo_LoadGeometry . . . 921
9.8 Topology Editors . . . 921
9.8.1 ST_AddIsoNode . . . 921
9.8.2 ST_AddIsoEdge . . . 922
9.8.3 ST_AddEdgeNewFaces . . . 922
9.8.4 ST_AddEdgeModFace . . . 923
9.8.5 ST_RemEdgeNewFace . . . 924
9.8.6 ST_RemEdgeModFace . . . 924
9.8.7 ST_ChangeEdgeGeom . . . 925
9.8.8 ST_ModEdgeSplit . . . 926
9.8.9 ST_ModEdgeHeal . . . 927
9.8.10 ST_NewEdgeHeal . . . 927
9.8.11 ST_MoveIsoNode . . . 928
9.8.12 ST_NewEdgesSplit . . . 929
9.8.13 ST_RemoveIsoNode . . . 930
9.8.14 ST_RemoveIsoEdge . . . 930

9.9 Topology Accessors . . . 931
9.9.1 GetEdgeByPoint . . . 931
9.9.2 GetFaceByPoint . . . 932
9.9.3 GetFaceContainingPoint . . . 933
9.9.4 GetNodeByPoint . . . 934
9.9.5 GetTopologyID . . . 935
9.9.6 GetTopologySRID . . . 935
9.9.7 GetTopologyName . . . 936
9.9.8 ST_GetFaceEdges . . . 936
9.9.9 ST_GetFaceGeometry . . . 937
9.9.10 GetRingEdges . . . 938
9.9.11 GetNodeEdges . . . 939
9.9.12 FindVertexSegmentPairsBelowDistance . . . 939
9.10 Topology Processing . . . 940
9.10.1 Polygonize . . . 940
9.10.2 AddNode . . . 940
9.10.3 AddEdge . . . 941
9.10.4 AddFace . . . 942
9.10.5 ST_Simplify . . . 944
9.10.6 RemoveUnusedPrimitives . . . 945
9.11 TopoGeometry Constructors . . . 945
9.11.1 CreateTopoGeom . . . 945
9.11.2 toTopoGeom . . . 947
9.11.3 TopoElementArray_Agg . . . 949
9.11.4 TopoElement . . . 950
9.12 TopoGeometry Editors . . . 950
9.12.1 clearTopoGeom . . . 950
9.12.2 TopoGeom_addElement . . . 951
9.12.3 TopoGeom_remElement . . . 951
9.12.4 TopoGeom_addTopoGeom . . . 952
9.12.5 toTopoGeom . . . 953
9.13 TopoGeometry Accessors . . . 953
9.13.1 GetTopoGeomElementArray . . . 953
9.13.2 GetTopoGeomElements . . . 953
9.13.3 ST_SRID . . . 954
9.14 TopoGeometry Outputs . . . 955
9.14.1 AsGML . . . 955
9.14.2 AsTopoJSON . . . 957
9.15 Topology Spatial Relationships . . . 958

9.15.1 Equals . . . 958
9.15.2 Intersects . . . 959
9.16 Importing and exporting Topologies . . . 960
9.16.1 Using the Topology exporter . . . 960
9.16.2 Using the Topology importer . . . 961

10 Raster Data Management, Queries, and Applications 962
10.1 Loading and Creating Rasters . . . 962
10.1.1 Using raster2pgsql to load rasters . . . 962
10.1.1.1 Example Usage . . . 962
10.1.1.2 raster2pgsql options . . . 963
10.1.2 Creating rasters using PostGIS raster functions . . . 965
10.1.3 Using "out db" cloud rasters . . . 965
10.2 Raster Coverages and Raster/Vector Analysis . . . 966
10.3 Raster Catalogs . . . 967
10.3.1 Raster Columns Catalog . . . 968
10.3.2 Raster Overviews . . . 969
10.4 Building Custom Applications with PostGIS Raster . . . 969
10.4.1 PHP Example Outputting using ST_AsPNG in concert with other raster functions . . . 970
10.4.2 ASP.NET C# Example Outputting using ST_AsPNG in concert with other raster functions . . . 970
10.4.3 Java console app that outputs raster query as Image file . . . 972
10.4.4 Use PLPython to dump out images via SQL . . . 974
10.4.5 Outputting Rasters with PSQL . . . 974

11 Raster Reference 976
11.1 Raster Support Data types . . . 977
11.1.1 geomval . . . 977
11.1.2 addbandarg . . . 977
11.1.3 rastbandarg . . . 977
11.1.4 raster . . . 978
11.1.5 reclassarg . . . 978
11.1.6 summarystats . . . 979
11.1.7 unionarg . . . 980
11.2 Raster Management . . . 980
11.2.1 AddRasterConstraints . . . 980
11.2.2 DropRasterConstraints . . . 982
11.2.3 AddOverviewConstraints . . . 983
11.2.4 DropOverviewConstraints . . . 984
11.2.5 PostGIS_GDAL_Version . . . 984

11.2.6 PostGIS_Raster_Lib_Build_Date . . . 985
11.2.7 PostGIS_Raster_Lib_Version . . . 985
11.2.8 ST_GDALDrivers . . . 986
11.2.9 UpdateRasterSRID . . . 989
11.2.10 ST_CreateOverview . . . 989
11.3 Raster Constructors . . . 990
11.3.1 ST_AddBand . . . 990
11.3.2 ST_AsRaster . . . 993
11.3.3 ST_AsRasterAgg . . . 996
11.3.4 ST_Band . . . 997
11.3.5 ST_MakeEmptyCoverage . . . 1001
11.3.6 ST_MakeEmptyRaster . . . 1002
11.3.7 ST_Tile . . . 1004
11.3.8 ST_Retile . . . 1006
11.3.9 ST_FromGDALRaster . . . 1006
11.4 Raster Accessors . . . 1007
11.4.1 ST_GeoReference . . . 1007
11.4.2 ST_Height . . . 1008
11.4.3 ST_IsEmpty . . . 1009
11.4.4 ST_MemSize . . . 1010
11.4.5 ST_MetaData . . . 1010
11.4.6 ST_NumBands . . . 1011
11.4.7 ST_PixelHeight . . . 1012
11.4.8 ST_PixelWidth . . . 1013
11.4.9 ST_ScaleX . . . 1015
11.4.10 ST_ScaleY . . . 1015
11.4.11 ST_RasterToWorldCoord . . . 1016
11.4.12 ST_RasterToWorldCoordX . . . 1017
11.4.13 ST_RasterToWorldCoordY . . . 1018
11.4.14 ST_Rotation . . . 1019
11.4.15 ST_SkewX . . . 1020
11.4.16 ST_SkewY . . . 1021
11.4.17 ST_SRID . . . 1022
11.4.18 ST_Summary . . . 1022
11.4.19 ST_UpperLeftX . . . 1023
11.4.20 ST_UpperLeftY . . . 1024
11.4.21 ST_Width . . . 1024
11.4.22 ST_WorldToRasterCoord . . . 1025
11.4.23 ST_WorldToRasterCoordX . . . 1026

11.4.24 ST_WorldToRasterCoordY . . . 1026
11.5 Raster Band Accessors . . . 1027
11.5.1 ST_BandMetaData . . . 1027
11.5.2 ST_BandNoDataValue . . . 1029
11.5.3 ST_BandIsNoData . . . 1029
11.5.4 ST_BandPath . . . 1031
11.5.5 ST_BandFileSize . . . 1031
11.5.6 ST_BandFileTimestamp . . . 1032
11.5.7 ST_BandPixelType . . . 1032
11.5.8 ST_MinPossibleValue . . . 1033
11.5.9 ST_HasNoBand . . . 1034
11.6 Raster Pixel Accessors and Setters . . . 1034
11.6.1 ST_PixelAsPolygon . . . 1034
11.6.2 ST_PixelAsPolygons . . . 1035
11.6.3 ST_PixelAsPoint . . . 1037
11.6.4 ST_PixelAsPoints . . . 1038
11.6.5 ST_PixelAsCentroid . . . 1039
11.6.6 ST_PixelAsCentroids . . . 1040
11.6.7 ST_Value . . . 1042
11.6.8 ST_NearestValue . . . 1045
11.6.9 ST_SetZ . . . 1047
11.6.10 ST_SetM . . . 1049
11.6.11 ST_Neighborhood . . . 1051
11.6.12 ST_SetValue . . . 1053
11.6.13 ST_SetValues . . . 1054
11.6.14 ST_DumpValues . . . 1062
11.6.15 ST_PixelOfValue . . . 1063
11.7 Raster Editors . . . 1064
11.7.1 ST_SetGeoReference . . . 1064
11.7.2 ST_SetRotation . . . 1065
11.7.3 ST_SetScale . . . 1066
11.7.4 ST_SetSkew . . . 1067
11.7.5 ST_SetSRID . . . 1069
11.7.6 ST_SetUpperLeft . . . 1069
11.7.7 ST_Resample . . . 1070
11.7.8 ST_Rescale . . . 1071
11.7.9 ST_Reskew . . . 1073
11.7.10 ST_SnapToGrid . . . 1074
11.7.11 ST_Resize . . . 1075

11.7.12 ST_Transform . . . 1077
11.8 Raster Band Editors . . . 1081
11.8.1 ST_SetBandNoDataValue . . . 1081
11.8.2 ST_SetBandIsNoData . . . 1082
11.8.3 ST_SetBandPath . . . 1084
11.8.4 ST_SetBandIndex . . . 1085
11.9 Raster Band Statistics and Analytics . . . 1086
11.9.1 ST_Count . . . 1086
11.9.2 ST_CountAgg . . . 1087
11.9.3 ST_Histogram . . . 1088
11.9.4 ST_Quantile . . . 1090
11.9.5 ST_SummaryStats . . . 1092
11.9.6 ST_SummaryStatsAgg . . . 1094
11.9.7 ST_ValueCount . . . 1095
11.10 Raster Inputs . . . 1098
11.10.1 ST_RastFromWKB . . . 1098
11.10.2 ST_RastFromHexWKB . . . 1099
11.11 Raster Outputs . . . 1099
11.11.1 ST_AsBinary/ST_AsWKB . . . 1099
11.11.2 ST_AsHexWKB . . . 1100
11.11.3 ST_AsGDALRaster . . . 1101
11.11.4 ST_AsJPEG . . . 1102
11.11.5 ST_AsPNG . . . 1103
11.11.6 ST_AsTIFF . . . 1104
11.12 Raster Processing: Map Algebra . . . 1105
11.12.1 ST_Clip . . . 1105
11.12.2 ST_ColorMap . . . 1110
11.12.3 ST_Grayscale . . . 1113
11.12.4 ST_Intersection . . . 1117
11.12.5 ST_MapAlgebra (callback function version) . . . 1120
11.12.6 ST_MapAlgebra (expression version) . . . 1127
11.12.7 ST_MapAlgebraExpr . . . 1129
11.12.8 ST_MapAlgebraExpr . . . 1133
11.12.9 ST_MapAlgebraFct . . . 1139
11.12.10 ST_MapAlgebraFct . . . 1144
11.12.11 ST_MapAlgebraFctNgb . . . 1150
11.12.12 ST_Reclass . . . 1154
11.12.13 ST_ReclassExact . . . 1155
11.12.14 ST_Union . . . 1157

11.13Built-in Map Algebra Callback Functions . . . 1158
11.13.1 ST_Distinct4ma . . . 1158
11.13.2 ST_InvDistWeight4ma . . . 1159
11.13.3 ST_Max4ma . . . 1160
11.13.4 ST_Mean4ma . . . 1161
11.13.5 ST_Min4ma . . . 1163
11.13.6 ST_MinDist4ma . . . 1164
11.13.7 ST_Range4ma . . . 1165
11.13.8 ST_StdDev4ma . . . 1166
11.13.9 ST_Sum4ma . . . 1167
11.14Raster Processing: DEM (Elevation) . . . 1168
11.14.1 ST_Aspect . . . 1168
11.14.2 ST_HillShade . . . 1171
11.14.3 ST_Roughness . . . 1173
11.14.4 ST_Slope . . . 1173
11.14.5 ST_TPI . . . 1175
11.14.6 ST_TRI . . . 1176
11.14.7 ST_InterpolateRaster . . . 1177
11.14.8 ST_Contour . . . 1178
11.15Raster Processing: Raster to Geometry . . . 1179
11.15.1 Box3D . . . 1179
11.15.2 ST_ConvexHull . . . 1180
11.15.3 ST_DumpAsPolygons . . . 1181
11.15.4 ST_Envelope . . . 1182
11.15.5 ST_MinConvexHull . . . 1182
11.15.6 ST_Polygon . . . 1184
11.15.7 ST_IntersectionFractions . . . 1186
11.16Raster Operators . . . 1187
11.16.1 && . . . 1187
11.16.2 &< . . . 1187
11.16.3 &> . . . 1188
11.16.4 = . . . 1189
11.16.5 @ . . . 1189
11.16.6 ~= . . . 1190
11.16.7 ~ . . . 1190
11.17Raster and Raster Band Spatial Relationships . . . 1191
11.17.1 ST_Contains . . . 1191
11.17.2 ST_ContainsProperly . . . 1192
11.17.3 ST_Covers . . . 1193

11.17.4 ST_CoveredBy . . . 1194
11.17.5 ST_Disjoint . . . 1195
11.17.6 ST_Intersects . . . 1196
11.17.7 ST_Overlaps . . . 1197
11.17.8 ST_Touches . . . 1198
11.17.9 ST_SameAlignment . . . 1199
11.17.10 ST_NotSameAlignmentReason . . . 1200
11.17.11 ST_Within . . . 1200
11.17.12 ST_DWithin . . . 1201
11.17.13 ST_DFullyWithin . . . 1202
11.18 Raster Tips . . . 1203
11.18.1 Out-DB Rasters . . . 1203
11.18.1.1 Directory containing many files . . . 1203
11.18.1.2 Maximum Number of Open Files . . . 1204
11.18.1.2.1 Maximum number of open files for the entire system . . . 1204
11.18.1.2.2 Maximum number of open files per process . . . 1205

12 PostGIS Extras 1207

13 PostGIS Special Functions Index 1208
13.1 PostGIS Aggregate Functions . . . 1208
13.2 PostGIS Window Functions . . . 1209
13.3 PostGIS SQL-MM Compliant Functions . . . 1209
13.4 PostGIS Geography Support Functions . . . 1213
13.5 PostGIS Raster Support Functions . . . 1214
13.6 PostGIS Geometry / Geography / Raster Dump Functions . . . 1219
13.7 PostGIS Box Functions . . . 1220
13.8 PostGIS Functions that support 3D . . . 1221
13.9 PostGIS Curved Geometry Support Functions . . . 1226
13.10 PostGIS Polyhedral Surface Support Functions . . . 1230
13.11 PostGIS Function Support Matrix . . . 1233
13.12 New, Enhanced or changed PostGIS Functions . . . 1251
13.12.1 PostGIS Functions new or enhanced in 3.7 . . . 1251
13.12.2 PostGIS Functions new or enhanced in 3.6 . . . 1253
13.12.3 PostGIS Functions new or enhanced in 3.5 . . . 1255
13.12.4 PostGIS Functions new or enhanced in 3.4 . . . 1256
13.12.5 PostGIS Functions new or enhanced in 3.3 . . . 1258
13.12.6 PostGIS Functions new or enhanced in 3.2 . . . 1258
13.12.7 PostGIS Functions new or enhanced in 3.1 . . . 1260

13.12.8 PostGIS Functions new or enhanced in 3.0 . . . 1261
13.12.9 PostGIS Functions new or enhanced in 2.5 . . . 1262
13.12.10 PostGIS Functions new or enhanced in 2.4 . . . 1264
13.12.11 PostGIS Functions new or enhanced in 2.3 . . . 1265
13.12.12 PostGIS Functions new or enhanced in 2.2 . . . 1266
13.12.13 PostGIS Functions new or enhanced in 2.1 . . . 1269
13.12.14 PostGIS Functions new or enhanced in 2.0 . . . 1273
13.12.15 PostGIS Functions new or enhanced in 1.5 . . . 1282
13.12.16 PostGIS Functions new or enhanced in 1.4 . . . 1284
13.12.17 PostGIS Functions new or enhanced in 1.3 . . . 1284

14 Reporting Problems **1285**
14.1 Reporting Software Bugs . . . 1285
14.2 Reporting Documentation Issues . . . 1285

15 Credits and Acknowledgments **1287**
15.1 Project Steering Committee . . . 1287
15.2 Core Contributors Present . . . 1287
15.3 Core Contributors Past . . . 1288
15.4 PostGIS Raster Origins . . . 1288
15.5 Other Contributors . . . 1289

A Release Notes **1293**
A.1 PostGIS 3.7.0 . . . 1293
A.1.1 Breaking Changes . . . 1293
A.1.2 New Features . . . 1293
A.1.3 Enhancements . . . 1294

Abstract

PostGIS is an extension to the PostgreSQL object-relational database system which allows GIS (Geographic Information Systems) objects to be stored in the database. PostGIS includes support for GiST-based R-Tree spatial indexes, and functions for analysis and processing of GIS objects.





This is the manual for version 3.7.0rc1



This work is licensed under a Creative Commons Attribution-Share Alike 3.0 License. Feel free to use this material any way you like, but we ask that you attribute credit to the PostGIS Project and wherever possible, a link back to https://postgis.net.

Chapter 1

Introduction

1.1 What Is PostGIS?

PostGIS extends PostgreSQL so that geospatial data can be stored, indexed, queried, and processed in the database. It keeps location data beside ordinary relational data, so a spatial query can use the same SQL, transactions, constraints, access controls, backup tools, and application connections as the rest of a PostgreSQL database.

The main `postgis` extension adds:

- `geometry` for planar coordinate systems and `geography` for geodetic coordinates on the earth.
- Spatial indexes that let the PostgreSQL planner avoid scanning every row when objects can be filtered by location.
- Functions for measurement, relationships, overlays, geometry construction and editing, coordinate transformation, clustering, and spatial data exchange.
- Support for common geospatial encodings and formats, including WKT, WKB, GeoJSON, GML, KML, FlatGeobuf, Geobuf, and Mapbox Vector Tiles.

This makes PostgreSQL a spatial database: applications can ask both ordinary questions and location-aware questions without moving data to a separate GIS processing system. Desktop GIS, map servers, ETL tools, and application frameworks can connect through normal PostgreSQL interfaces.

1.2 What Is in the PostGIS Distribution?

This manual documents the main PostGIS source distribution. Depending on how PostGIS was built and packaged, it can provide several PostgreSQL extensions and command-line tools:

`postgis` The core geometry and geography types, spatial indexes, and most spatial functions.

`postgis_raster` Raster storage, analysis, loading, and export. Raster support uses GDAL and can be omitted from a build.

`postgis_topology` SQL/MM topology types and functions for models based on shared nodes, edges, and faces.

`postgis_sfcgal` Additional 2D and 3D processing functions backed by the optional SFCGAL library.

Loaders and utilities **shp2pgsql**, **pgsql2shp**, and **raster2pgsql** move data between common file formats and PostGIS. Other utilities support upgrades, profiling, and development.

The source tree also contains internal libraries such as `liblwgeom` and `libpgcommon`. They are implementation layers shared by PostGIS components, not separately released PostgreSQL extensions. Most users start by enabling the core extension with `CREATE EXTENSION postgis`; see Chapter 2 for available components and build options.

1.3 The PostGIS Project Family

This manual covers the main `postgis/postgis` distribution. The official PostGIS source organization also hosts or mirrors complementary projects. These projects have their own release cycles, installation instructions, and documentation; installing the core extension does not install them.

- Address Standardizer parses and normalizes postal addresses. It moved from the main PostGIS source tree to a separate repository for the PostGIS 3.7 development cycle.
- PostGIS TIGER Geocoder provides United States address geocoding and reverse geocoding using Census TIGER data. It is also maintained separately from the main distribution.
- H3-pg provides PostgreSQL bindings for the H3 hierarchical hexagonal indexing system and interoperates with PostGIS.
- PostGIS Java provides Java bindings for PostGIS types.
- Introduction to PostGIS is the hands-on workshop maintained by the PostGIS project.

Other PostgreSQL spatial extensions can complement PostGIS while remaining independent projects. For example, pgRouting adds routing and graph analysis. Use each project's own manual for its functions and support policy.

1.4 How to Use This Manual

The manual is both a guide to the PostGIS data model and the complete reference for the SQL interface. Choose a starting point based on what you are trying to do:

If you are new to spatial SQL Start with the Introduction to PostGIS workshop for a tutorial, then use Chapter 4 to understand spatial types, coordinate systems, validity, loading, and indexes.

If you are installing or upgrading PostGIS Read Chapter 2 and Chapter 3. Package-specific instructions and supported release information are maintained on the PostGIS website.

If you are writing spatial queries Read Chapter 5 for spatial relationships and index-aware query patterns, and Chapter 6 for common performance problems.

If you are connecting a desktop GIS Use the client's current database documentation for its user interface and supported geometry types. For example, the QGIS DB Manager lesson covers PostgreSQL connections, table layers, SQL query layers, import, and export. A query layer should return a stable unique identifier and a geometry column with a known SRID and geometry type. Expensive read-only queries can be exposed through a view or materialized view and indexed as appropriate.

If you need a particular function Use Chapter 7 or Chapter 13. Function pages state accepted types, dimensional support, standards conformance, dependency requirements, and version availability.

If you use an optional component See Chapter 9, Chapter 10, Chapter 11, or Chapter 8.

If something goes wrong Collect the versions reported by PostGIS_Full_Version and follow Chapter 14. Check Appendix A when behavior differs between release lines.

1.5 Standards Support

This manual marks individual functions as compliant with spatial standards so users can tell which behavior is portable and which behavior is a PostGIS extension. PostGIS implements the OGC Simple Features for SQL model, including the geometry type, Well-Known Text (WKT), Well-Known Binary (WKB), and the common spatial accessors, constructors, predicates, and processing functions. PostGIS also implements many SQL/MM Spatial types and functions.

PostGIS supports geometry families defined by later standards, including SQL/MM curved geometry and OGC Simple Features Access 1.2 PolyhedralSurface, Triangle, and TIN. Support for these geometry families is function-specific, and each function page documents the supported input types and conformance level.

PostGIS also includes widely used extensions and integrations. Examples of PostGIS extensions include Extended WKT (EWKT), Extended WKB (EWKB), SRID metadata in those extended formats, geography, and raster. PostGIS also supports input and output exchange formats defined by other standards, such as GeoJSON, GML, and KML. These features are documented individually and should not be read as OGC Simple Features or SQL/MM requirements unless the specific function or format says so.

When the implementation intentionally differs from a standard, or when a standard leaves behavior ambiguous, the relevant function page describes the PostGIS behavior. Chapter 13 summarizes SQL/MM compliance and geometry-type support across functions.

1.6 Project and Community

PostGIS began in 2001 as a spatial database research project at Refractions Research. It is now an OSGeo project developed by contributors and organizations around the world. The Project Steering Committee coordinates releases, project policy, and the general direction of development. The complete contributor and sponsor record is in Chapter 15 at the end of this manual.

The manual is the authoritative reference for the released SQL interface. The project website and source tree provide the surrounding material:

- Documentation routes readers to getting-started instructions, versioned manuals, tips, training material, and the FAQ.
- Community lists mailing lists, chat, events, videos, and other support channels.
- Developer documentation covers development environments, testing, coding style, internals, maintenance, governance, releases, and website work.

Chapter 2

PostGIS Installation

This chapter details the steps required to install PostGIS.

2.1 Short Version

To compile assuming you have all the dependencies in your search path:

Code

```
tar -xvzf postgis-3.7.0rc1.tar.gz
cd postgis-3.7.0rc1
./configure
make
make install
```

Once PostGIS is installed, it needs to be enabled (Section 3.3) or upgraded (Section 3.4) in each individual database you want to use it in.

2.2 Compiling and Install from Source



Note

Many OS systems now include pre-built packages for PostgreSQL/PostGIS. In many cases compilation is only necessary if you want the most bleeding edge versions or you are a package maintainer.

This section includes general compilation instructions. If you are compiling for Windows or another operating system, see the PostGIS getting started guides and the development environment guides.

Pre-Built Packages for various OS are listed in the PostGIS getting started guides

If you are a windows user, you can get stable builds via Stackbuilder or PostGIS Windows download site We also have very bleeding-edge windows experimental builds that are built usually once or twice a week or whenever anything exciting happens. You can use these to experiment with the in progress releases of PostGIS

The PostGIS module is an extension to the PostgreSQL backend server. As such, PostGIS 3.7.0rc1 *requires* full PostgreSQL server headers access in order to compile. It can be built against PostgreSQL versions 14 - 19beta2. Earlier versions of PostgreSQL are *not* supported.

Refer to the PostgreSQL installation guides if you haven't already installed PostgreSQL. https://www.postgresql.org .



Note
For GEOS functionality, when you install PostgreSQL you may need to explicitly link PostgreSQL against the standard C++ library:

Code

```
LDFLAGS=-lstdc++ ./configure [YOUR OPTIONS HERE]
```

This is a workaround for bogus C++ exceptions interaction with older development tools. If you experience weird problems (backend unexpectedly closed or similar things) try this trick. This will require recompiling your PostgreSQL from scratch, of course.

The following steps outline the configuration and compilation of the PostGIS source. They are written for Linux users and will not work on Windows or Mac.

2.2.1 Getting the Source

Retrieve the PostGIS source archive from the downloads website https://download.osgeo.org/postgis/source/postgis-3.7.0rc1.tar.gz

Code

```
wget https://download.osgeo.org/postgis/source/postgis-3.7.0rc1.tar.gz
tar -xvzf postgis-3.7.0rc1.tar.gz
cd postgis-3.7.0rc1
```

This will create a directory called `postgis-3.7.0rc1` in the current working directory.

Alternatively, checkout the source from the git repository https://git.osgeo.org/gitea/postgis/postgis/ .

Code

```
git clone https://git.osgeo.org/gitea/postgis/postgis.git postgis
cd postgis
sh autogen.sh
```

Change into the newly created `postgis` directory to continue the installation.

Code

```
./configure
```

2.2.2 Install Requirements

PostGIS has the following requirements for building and usage:

Required

- PostgreSQL 14 - 19beta2. A complete installation of PostgreSQL (including server headers) is required. PostgreSQL is available from https://www.postgresql.org 18 .

  For current release-line support and end-of-life policy, see https://postgis.net/development/versions_eol/

- GNU C compiler (`gcc`). Some other ANSI C compilers can be used to compile PostGIS, but we find far fewer problems when compiling with `gcc`.

- GNU Make (`gmake` or `make`). For many systems, GNU `make` is the default version of make. Check the version by invoking `make -v`. Other versions of `make` may not process the PostGIS `Makefile` properly.
- Proj reprojection library. Proj 6.1 or above is required. The Proj library is used to provide coordinate reprojection support within PostGIS. Proj is available for download from https://proj.org/ .
- GEOS geometry library, version 3.10.0 or greater, but GEOS 3.15+ is required to take full advantage of all the new functions and features. GEOS is available for download from https://libgeos.org .
- LibXML2, version 2.5.x or higher. LibXML2 is currently used in some imports functions (ST_GeomFromGML and ST_GeomFromKM LibXML2 is available for download from https://gitlab.gnome.org/GNOME/libxml2/-/releases.
- JSON-C, version 0.9 or higher. JSON-C is currently used to import GeoJSON via the function ST_GeomFromGeoJson. JSON-C is available for download from https://github.com/json-c/json-c/releases/.
- The GNU Multiple Precision Arithmetic Library (libgmp). Used to control precision of math operations in PostGIS Topology. GMP is available for download from https://gmplib.org/.
- GDAL, version 2.4 or higher, is required for raster support. GDAL 3+ is preferred. https://gdal.org/download.html.
- If compiling with PostgreSQL+JIT, LLVM version >=6 is required https://trac.osgeo.org/postgis/ticket/4125.

Optional

- GDAL (pseudo optional) only if you don't want raster you can leave it out. Also make sure to enable the drivers you want to use as described in Section 3.2.
- GTK (requires GTK+2.0, 2.8+) to compile the shp2pgsql-gui shape file loader. http://www.gtk.org/ .
- SFCGAL, 1.3.1 or higher is required, 2.3+ is needed to be able to use all functionality. SFCGAL can be used to provide additional 2D and 3D advanced analysis functions to PostGIS cf Chapter 8. https://sfcgal.gitlab.io/sfcgal/.
- To enable ST_AsMVT protobuf-c library 1.1.0 or higher (for usage) and the protoc-c compiler (for building) are required. Also, pkg-config is required to verify the correct minimum version of protobuf-c. See protobuf-c. By default, Postgis will use Wagyu to validate MVT polygons faster which requires a c++11 compiler. It will use CXXFLAGS and the same compiler as the PostgreSQL installation. To disable this and use GEOS instead use the `--without-wagyu` during the configure step.
- CUnit (`CUnit`). This is needed for regression testing. http://cunit.sourceforge.net/
- DocBook (`xsltproc`) is required for building the documentation. Docbook is available from http://www.docbook.org/ .
- DBLatex (`dblatex`) is required for building the documentation in PDF format. DBLatex is available from http://dblatex.sourceforge.n .
- PDF documentation needs an SVG rasterizer for its high-resolution figure fallbacks. The build prefers `rsvg-convert`, and can also use GraphicsMagick (`gm convert`) or ImageMagick (`magick` or `convert`).

2.2.3 Build configuration

As with most linux installations, the first step is to generate the Makefile that will be used to build the source code. This is done by running the shell script

Code

```
./configure
```

With no additional parameters, this command will attempt to automatically locate the required components and libraries needed to build the PostGIS source code on your system. Although this is the most common usage of **./configure**, the script accepts several parameters for those who have the required libraries and programs in non-standard locations.

The following list shows only the most commonly used parameters. For a complete list, use the **--help** or **--help=short** parameters.

--with-library-minor-version Starting with PostGIS 3.0, the library files generated by default will no longer have the minor version as part of the file name. This means all PostGIS 3 libs will end in `postgis-3`. This was done to make pg_upgrade easier, with downside that you can only install one version PostGIS 3 series in your server. To get the old behavior of file including the minor version: e.g. `postgis-3.0` add this switch to your configure statement.

--prefix=PREFIX This is the installation prefix for PostGIS command-line utilities. By default, utilities are installed in the executable directory reported by the detected PostgreSQL installation.



Caution
PostgreSQL extension files are installed in the directories reported by **pg_config**, so the server can load them. This includes the PostGIS shared library and SQL extension files. Use `--with-pgconfig=FILE` to choose the PostgreSQL installation that PostGIS builds and installs against.

--with-pgconfig=FILE PostgreSQL provides a utility called **pg_config** to enable extensions like PostGIS to locate the PostgreSQL installation directory. Use this parameter (**--with-pgconfig=/path/to/pg_config**) to manually specify a particular PostgreSQL installation that PostGIS will build against.

--with-gdalconfig=FILE GDAL, a required library, provides functionality needed for raster support **gdal-config** to enable software installations to locate the GDAL installation directory. Use this parameter (**--with-gdalconfig=/path/to/gdal-config**) to manually specify a particular GDAL installation that PostGIS will build against.

--with-geosconfig=FILE GEOS, a required geometry library, provides a utility called **geos-config** to enable software installations to locate the GEOS installation directory. Use this parameter (**--with-geosconfig=/path/to/geos-config**) to manually specify a particular GEOS installation that PostGIS will build against.

--with-xml2config=FILE LibXML is the library required for doing GeomFromKML/GML processes. It normally is found if you have libxml installed, but if not or you want a specific version used, you'll need to point PostGIS at a specific `xml2-config` confi file to enable software installations to locate the LibXML installation directory. Use this parameter (**>--with-xml2config=/path/to/xml2-config**) to manually specify a particular LibXML installation that PostGIS will build against.

--with-projdir=DIR Proj is a reprojection library required by PostGIS. Use this parameter (**--with-projdir=/path/to/projdir**) to manually specify a particular Proj installation directory that PostGIS will build against.

--with-libiconv=DIR Directory where iconv is installed.

--with-jsondir=DIR JSON-C is an MIT-licensed JSON library required by PostGIS ST_GeomFromJSON support. Use this parameter (**--with-jsondir=/path/to/jsondir**) to manually specify a particular JSON-C installation directory that PostGIS will build against.

--with-gui Compile the data import GUI (requires GTK+2.0). This will create shp2pgsql-gui graphical interface to shp2pgsql.

--without-raster Compile without raster support.

--without-topology Compile without topology support.

--with-gettext=no By default PostGIS will try to detect gettext support and compile with it, however if you run into incompatibility issues that cause breakage of loader, you can disable it entirely with this command. Refer to ticket http://trac.osgeo.org/postgis/ticket/748 for an example issue solved by configuring with this. NOTE: that you aren't missing much by turning this off. This is used for international help/label support for the GUI loader which is not yet documented and still experimental.

--with-sfcgal=PATH By default PostGIS will not install with sfcgal support without this switch. `PATH` is an optional argument that allows to specify an alternate PATH to sfcgal-config.

--without-phony-revision Disable updating postgis_revision.h to match current HEAD of the git repository.



Note
If you obtained PostGIS from the code repository , the first step is really to run the script

Code

```
./autogen.sh
```

This script will generate the **configure** script that in turn is used to customize the installation of PostGIS.

If you instead obtained PostGIS as a tarball, running **./autogen.sh** is not necessary as **configure** has already been generated.

2.2.4 Building

Once the Makefile has been generated, building PostGIS is as simple as running

Code

```
make
```

Output

```
[last line of output]
PostGIS was built successfully. Ready to install.
```

All the functions have comments generated from the documentation. If you wish to install these comments into your spatial databases later, run the command which requires docbook. The postgis_comments.sql and other package comment files are also packaged in the tar.gz distribution in the doc folder so no need to make comments if installing from the tar ball. Comments are also included as part of the CREATE EXTENSION install.

Code

```
make comments
```

The **make cheatsheets** target generates html cheat sheets suitable for quick reference or for student handouts. This requires xsltproc to build and will generate 4 files in doc folder `topology_cheatsheet.html`, `raster_cheatsheet.html`, `postgis_cheatsheet.html`

You can download some pre-built ones available in html and pdf from PostGIS / PostgreSQL Study Guides

Code

```
make cheatsheets
```

2.2.5 Building PostGIS Extensions and Deploying them

The PostGIS extensions are built and installed automatically when PostgreSQL extension support is available.

If you are building from source repository, you need to build the function descriptions first. These get built if you have docbook installed. You can also manually build with the statement:

Code

```
make comments
```

Building the comments is not necessary if you are building from a release tar ball since these are packaged pre-built with the tar ball already.

The extensions should automatically build as part of the make install process. You can if needed build from the extensions folders or copy files if you need them on a different server.

Set `PGUSER` if you need to override the psql connection user, then test before installing. After installation, run the extension regression tests.

Code

```
cd extensions
cd postgis
make clean
make
export PGUSER=postgres
make check
make install
make check RUNTESTFLAGS=--extension
```



Note

`make check` uses psql to run tests and as such can use psql environment variables. Common ones useful to override are `PGUSER`,`PGPORT`, and `PGHOST`. Refer to psql environment variables

The extension files will always be the same for the same version of PostGIS and PostgreSQL regardless of OS, so it is fine to copy over the extension files from one OS to another as long as you have the PostGIS binaries already installed on your servers.

If you want to install the extensions manually on a separate server different from your development, You need to copy the following files from the extensions folder into the `PostgreSQL / share / extension` folder of your PostgreSQL install as well as the needed binaries for regular PostGIS if you don't have them already on the server.

- These are the control files that denote information such as the version of the extension to install if not specified. `postgis.control, postgis_topology.control`.
- All the files in the /sql folder of each extension. Note that these need to be copied to the root of the PostgreSQL share/extension folder `extensions/postgis/sql/*.sql`, `extensions/postgis_topology/sql/*.sql`

Once you do that, you should see `postgis`, `postgis_topology` as available extensions in PgAdmin -> extensions.

If you are using psql, you can verify that the extensions are installed by running this query:

Code

```
SELECT name, default_version, installed_version
FROM pg_available_extensions
WHERE name LIKE 'postgis%'
ORDER BY name;
```

Output

```
+------------------+-----------------+-------------------+
| name             | default_version | installed_version |
+------------------+-----------------+-------------------+
| postgis          | 3.7.0rc1        | 3.7.0rc1          |
| postgis_raster   | 3.7.0rc1        | 3.7.0rc1          |
| postgis_sfcgal   | 3.7.0rc1        |                   |
| postgis_topology | 3.7.0rc1        |                   |
+------------------+-----------------+-------------------+
```

If you have the extension installed in the database you are querying, you'll see mention in the `installed_version` column. If you get no records back, it means you don't have postgis extensions installed on the server at all. PgAdmin III 1.14+ will also provide this information in the `extensions` section of the database browser tree and will even allow upgrade or uninstall by right-clicking.

If you have the extensions available, you can install postgis extension in your database of choice by either using pgAdmin extension interface or running these sql commands:

Code

```
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_raster;
CREATE EXTENSION postgis_sfcgal;
CREATE EXTENSION postgis_topology;
```

In psql you can use to see what versions you have installed and also what schema they are installed.

Code

```
\connect mygisdb
\x
\dx postgis*
```

Output

```
List of installed extensions
+-[ RECORD 1 ]-------------------------------------------------+
| Name        | postgis                                        |
| Version     | 3.7.0rc1                                         |
| Schema      | public                                         |
| Description | PostGIS geometry, geography, and raster...      |
+-[ RECORD 2 ]-------------------------------------------------+
| Name        | postgis_raster                                 |
| Version     | 3.7.0rc1                                         |
| Schema      | public                                         |
| Description | PostGIS raster types and functions             |
+-[ RECORD 3 ]-------------------------------------------------+
| Name        | postgis_topology                               |
| Version     | 3.7.0rc1                                         |
| Schema      | topology                                       |
| Description | PostGIS topology spatial types and functions   |
+-------------------------------------------------------------+
```



Warning

Extension tables `spatial_ref_sys`, `layer`, `topology` can not be explicitly backed up. They can only be backed up when the respective `postgis` or `postgis_topology` extension is backed up, which only happens when you back up the whole database. Only srid records not packaged with PostGIS are captured in backups, so don't change the entries we ship and expect the modifications to persist. Put in a ticket if you find an issue. The structures of extension tables are never backed up since they are created with `CREATE EXTENSION` and assumed to be the same for a given version of an extension. These behaviors are built into the current PostgreSQL extension model.

If you installed 3.7.0rc1, without using our wonderful extension system, you can change it to be extension based by running the below commands to package the functions in their respective extension. Installing using `unpackaged` was removed in PostgreSQL 13, so you are advised to switch to an extension build before upgrading to PostgreSQL 13.

Code

```
CREATE EXTENSION postgis FROM unpackaged;
CREATE EXTENSION postgis_raster FROM unpackaged;
CREATE EXTENSION postgis_topology FROM unpackaged;
```

2.2.6 Testing

If you wish to test the PostGIS build, run

Code

```
make check
```

The above command will run through various checks and regression tests using the generated library against an actual PostgreSQL database.



Note
If you configured PostGIS using non-standard PostgreSQL, GEOS, or Proj locations, you may need to add their library locations to the `LD_LIBRARY_PATH` environment variable.



Caution
Currently, the **make check** relies on the `PATH` and `PGPORT` environment variables when performing the checks - it does *not* use the PostgreSQL version that may have been specified using the configuration parameter **--with-pgconfig**. So make sure to modify your PATH to match the detected PostgreSQL installation during configuration or be prepared to deal with the impending headaches.



Note
Sandboxed build accounts that are not PostgreSQL superusers can delegate database ownership during the regression cycle by exporting `POSTGIS_REGRESS_DB_OWNER`. The harness will create the temporary regression database owned by the nominated role while continuing to connect using the less privileged account. Combine this with `POSTGIS_REGRESS_ROLE_EXT_CREATOR` when the extension creation role must differ from the database owner.
These variables allow automated environments to exercise the full upgrade and extension install paths without promoting the calling account to superuser, provided the target PostgreSQL instance permits extension installation by those delegate roles.

If successful, make check will produce the output of almost 500 tests. The results will look similar to the following (numerous lines omitted below):

Code

```
     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

  .
  .
  .

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites     44     44    n/a      0        0
               tests    300    300    300      0        0
             asserts   4215   4215   4215      0      n/a
Elapsed time =    0.229 seconds

  .
  .
  .

Running tests

  .
  .
```

```
  .

Run tests: 134
Failed: 0


-- if you build with SFCGAL

  .
  .
  .

Running tests

  .
  .
  .

Run tests: 13
Failed: 0

-- if you built with raster support

  .
  .
  .

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites     12     12    n/a      0        0
               tests     65     65     65      0        0
             asserts  45896  45896  45896      0      n/a


  .
  .
  .

Running tests

  .
  .
  .

Run tests: 101
Failed: 0

-- topology regress

.
.
.

Running tests

  .
  .
  .

Run tests: 51
Failed: 0

-- if you built --with-gui, you should see this too
```

```
    CUnit - A unit testing framework for C - Version 2.1-2
    http://cunit.sourceforge.net/

  .
  .
  .

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      2      2    n/a      0        0
               tests      4      4      4      0        0
             asserts      4      4      4      0      n/a
```

output should look like:

Output

```
============== dropping database "contrib_regression" ==============
DROP DATABASE
============== creating database "contrib_regression" ==============
CREATE DATABASE
ALTER DATABASE
============== installing fuzzystrmatch               ==============
CREATE EXTENSION
============== installing postgis                     ==============
CREATE EXTENSION
============== running regression test queries        ==============
test test-normalize_address   ... ok

====================
All 2 tests passed.
====================
```

2.2.7 Installation

To install PostGIS, type

Code

```
make install
```

This will copy the PostGIS installation files into their appropriate subdirectory specified by the **--prefix** configuration parameter. In particular:

- The loader and dumper binaries are installed in `[prefix]/bin`.
- The SQL files, such as `postgis.sql`, are installed in `[prefix]/share/contrib`.
- The PostGIS libraries are installed in `[prefix]/lib`.

If you previously ran the **make comments** command to generate the `postgis_comments.sql`, `raster_comments.sql` file, install the sql file by running

Code

```
make comments-install
```



Note

`postgis_comments.sql`, `raster_comments.sql`, `topology_comments.sql` was separated from the typical build and installation targets since with it comes the extra dependency of **xsltproc**.

2.3 Common Problems during installation

There are several things to check when your installation or upgrade doesn't go as you expected.

PostgreSQL version mismatch.

Check that you have installed PostgreSQL 14 or newer, and that you are compiling against the same version of the PostgreSQL source as the version of PostgreSQL that is running. Mix-ups can occur when your (Linux) distribution has already installed PostgreSQL, or you have otherwise installed PostgreSQL before and forgotten about it. PostGIS will only work with PostgreSQL 14 or newer, and strange, unexpected error messages will result if you use an older version. To check the version of PostgreSQL which is running, connect to the database using psql and run this query:

Code

```
SELECT version();
```

If you are running an RPM based distribution, you can check for the existence of pre-installed packages using the **rpm** command as follows:

Code

```
rpm -qa | grep postgresql
```

Restoring an upgrade into a database without PostGIS.

If your upgrade fails, make sure you are restoring into a database that already has PostGIS installed.

Code

```
SELECT PostGIS_Full_Version();
```

Also check that configure has correctly detected the location and version of PostgreSQL, the Proj library and the GEOS library.

The output from configure is used to generate the `postgis_config.h` file. Check that the `POSTGIS_PGSQL_VERSION`, `POSTGIS_PROJ_VERSION` and `POSTGIS_GEOS_VERSION` variables have been set correctly.

Chapter 3

PostGIS Administration

3.1 Performance Tuning

Tuning for PostGIS performance is much like tuning for any PostgreSQL workload. The only additional consideration is that geometries and rasters are usually large, so memory-related optimizations generally have more of an impact on PostGIS than other types of PostgreSQL queries.

For general details about optimizing PostgreSQL, refer to Tuning your PostgreSQL Server.

PostgreSQL configuration can be set at the server level without touching `postgresql.conf` or `postgresql.auto.conf` by using the `ALTER SYSTEM` command.

Set `work_mem`, reload the configuration so new connections use the change, and display the current value. Use **SHOW ALL** to list all settings.

Code

```
ALTER SYSTEM SET work_mem = '256MB';
SELECT pg_reload_conf();
SHOW work_mem;
```

In addition to the Postgres settings, PostGIS has some custom settings which are listed in Section 7.22.

3.1.1 Startup

These settings are configured in `postgresql.conf`:

constraint_exclusion

- Default: partition
- This is generally used for table partitioning. The default of "partition" forces the planner to only analyze tables for constraint consideration if they are in an inherited hierarchy and avoids paying the planner penalty otherwise.

shared_buffers

- Default: ~128MB
- Set to about 25% to 40% of available RAM. On windows you may not be able to set as high.

max_worker_processes controls the maximum number of background processes available to the server and therefore also bounds how many workers can participate in parallel queries.

- Default: 8
- Sets the maximum number of background processes that the system can support. This parameter can only be set at server start.

3.1.2 Runtime

work_mem - sets the size of memory used for sort operations and complex queries

- Default: 1-4MB
- Adjust up for large dbs, complex queries, lots of RAM
- Adjust down for many concurrent users or low RAM.
- If you have lots of RAM and few developers:

Code

```
SET work_mem TO '256MB';
```

maintenance_work_mem - the memory size used for VACUUM, CREATE INDEX, etc.

- Default: 16-64MB
- Generally too low - ties up I/O, locks objects while swapping memory
- Recommend 32MB to 1GB on production servers w/lots of RAM, but depends on the # of concurrent users. If you have lots of RAM and few developers:

Code

```
SET maintenance_work_mem TO '1GB';
```

max_parallel_workers_per_gather

Parallel query support in modern PostgreSQL and PostGIS uses this setting to decide how many workers a plan may request. If set to higher than 0, then some queries such as those involving relation functions like `ST_Intersects` can use multiple processes and can run more than twice as fast when doing so. If you have a lot of processors to spare, you should change the value of this to as many processors as you have. Also make sure to bump up `max_worker_processes` to at least as high as this number.

- Default: 0
- Sets the maximum number of workers that can be started by a single `Gather` node. Parallel workers are taken from the pool of processes established by `max_worker_processes`. Note that the requested number of workers may not actually be available at run time. If this occurs, the plan will run with fewer workers than expected, which may be inefficient. Setting this value to 0, which is the default, disables parallel query execution.

3.2 Configuring raster support

If you enabled raster support you may want to read below how to properly configure it.

Out-of-db rasters and all raster drivers are disabled by default. In order to re-enable these, you need to set the following environment variables `POSTGIS_GDAL_ENABLED_DRIVERS` and `POSTGIS_ENABLE_OUTDB_RASTERS` in the server environment. You can also use the more cross-platform approach of setting the corresponding Section 7.22.

If you want to enable offline raster:

Code

```
POSTGIS_ENABLE_OUTDB_RASTERS=1
```

Any other setting or no setting at all will disable out of db rasters.

In order to enable all GDAL drivers available in your GDAL install, set this environment variable as follows

Code

```
POSTGIS_GDAL_ENABLED_DRIVERS=ENABLE_ALL
```

If you want to only enable specific drivers, set your environment variable as follows:

Code

```
POSTGIS_GDAL_ENABLED_DRIVERS="GTiff PNG JPEG GIF XYZ"
```



Note
If you are on windows, do not quote the driver list

Setting environment variables varies depending on OS. For PostgreSQL installed on Ubuntu or Debian via apt-postgresql, the preferred way is to edit `/etc/postgresql/MAJOR/CLUSTER/environment` where the placeholders refer to the PostgreSQL major version and cluster name.

On windows, if you are running as a service, you can set via System variables which for Windows 7 you can get to by right-clicking on Computer->Properties Advanced System Settings or in explorer navigating to `Control Panel\All Control Panel Items\System`. Then clicking *Advanced System Settings ->Advanced->Environment Variables* and adding new system variables.

After you set the environment variables, you'll need to restart your PostgreSQL service for the changes to take effect.

3.3 Creating spatial databases

3.3.1 Spatially enable database using EXTENSION

If you have compiled and installed the extensions/postgis modules, you can turn a database into a spatial one using the EXTENSION mechanism.

Core postgis extension includes geometry, geography, spatial_ref_sys and all the functions and comments. Raster and topology are packaged as a separate extension.

Run the following SQL snippet in the database you want to enable spatially:

Code

```
CREATE EXTENSION IF NOT EXISTS plpgsql;
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_raster; -- OPTIONAL
CREATE EXTENSION postgis_topology; -- OPTIONAL
```

3.3.2 Spatially enable database without using EXTENSION (discouraged)



Note
This is generally only needed if you cannot or don't want to get PostGIS installed in the PostgreSQL extension directory (for example during testing, development or in a restricted environment).

Adding PostGIS objects and function definitions into your database is done by loading the various sql files located in `[prefix]/share/contrib` as specified during the build phase.

The core PostGIS objects (geometry and geography types, and their support functions) are in the `postgis.sql` script. Raster objects are in the `rtpostgis.sql` script. Topology objects are in the `topology.sql` script.

For a complete set of EPSG coordinate system definition identifiers, you can also load the `spatial_ref_sys.sql` definitions file and populate the `spatial_ref_sys` table. This will permit you to perform ST_Transform() operations on geometries.

If you wish to add comments to the PostGIS functions, you can find them in the `postgis_comments.sql` script. Comments can be viewed by simply typing **\dd [function_name]** from a **psql** terminal window.

Run the following Shell commands in your terminal:

Code

```
DB=[yourdatabase]
SCRIPTSDIR=`pg_config --sharedir`/contrib/postgis-3.6/

# Core objects
psql -d ${DB} -f ${SCRIPTSDIR}/postgis.sql
psql -d ${DB} -f ${SCRIPTSDIR}/spatial_ref_sys.sql
psql -d ${DB} -f ${SCRIPTSDIR}/postgis_comments.sql # OPTIONAL

# Raster support (OPTIONAL)
psql -d ${DB} -f ${SCRIPTSDIR}/rtpostgis.sql
psql -d ${DB} -f ${SCRIPTSDIR}/raster_comments.sql # OPTIONAL

# Topology support (OPTIONAL)
psql -d ${DB} -f ${SCRIPTSDIR}/topology.sql
psql -d ${DB} -f ${SCRIPTSDIR}/topology_comments.sql # OPTIONAL
```

3.4 Upgrading spatial databases

Upgrading existing spatial databases can be tricky as it requires replacement or introduction of new PostGIS object definitions.

Unfortunately not all definitions can be easily replaced in a live database, so sometimes your best bet is a dump/reload process.

PostGIS provides a SOFT UPGRADE procedure for minor or bugfix releases, and a HARD UPGRADE procedure for major releases.

Before attempting to upgrade PostGIS, it is always worth to backup your data. If you use the -Fc flag to pg_dump you will always be able to restore the dump with a HARD UPGRADE.

3.4.1 Soft upgrade

If you installed your database using extensions, you'll need to upgrade using the extension model as well. If you installed using the old sql script way, you are advised to switch your install to extensions because the script way is no longer supported.

3.4.1.1 Soft upgrade using extensions

If you originally installed PostGIS with extensions, then you need to upgrade using extensions as well. Doing a minor upgrade with extensions is fairly painless.

Use the PostGIS_Extensions_Upgrade function to upgrade to the latest version you have installed.

Code

```
SELECT postgis_extensions_upgrade();
```

If you are upgrading a database that still runs PostGIS 2.5, issue an **ALTER EXTENSION** followed by `postgis_extensions_upg` to make sure the legacy `postgis_raster` objects are folded back into the main extension.

Code

```
ALTER EXTENSION postgis UPDATE;
SELECT postgis_extensions_upgrade();
```

This second call is needed to rebundle the `postgis_raster` extension.

Code

```
SELECT postgis_extensions_upgrade();
```

If you have multiple versions of PostGIS installed, and you don't want to upgrade to the latest, you can explicitly specify the version as follows:

Code

```
ALTER EXTENSION postgis UPDATE TO "3.7.0rc1";
ALTER EXTENSION postgis_topology UPDATE TO "3.7.0rc1";
```

If you get an error notice something like:

Output

```
No migration path defined for ... to 3.7.0rc1
```

Then you'll need to backup your database, create a fresh one as described in Section 3.3.1 and then restore your backup on top of this new database.

If you get a notice message like:

Output

```
Version "3.7.0rc1" of extension "postgis" is already installed
```

Then everything is already up to date and you can safely ignore it. **UNLESS** you're attempting to upgrade from an development version to the next (which doesn't get a new version number); in that case you can append "next" to the version string, and next time you'll need to drop the "next" suffix again:

Code

```
ALTER EXTENSION postgis UPDATE TO "3.7.0rc1next";
  ALTER EXTENSION postgis_topology UPDATE TO "3.7.0rc1next";
```



Note
If you installed PostGIS originally without a version specified, you can often skip the reinstallation of postgis extension before restoring since the backup just has `CREATE EXTENSION postgis` and thus picks up the newest latest version during restore.



Note
If you are upgrading PostGIS extension from a version prior to 3.0.0, you will have a new extension *postgis_raster* which you can safely drop, if you don't need raster support.

Code

```
DROP EXTENSION postgis_raster;
```

3.4.1.2 Soft upgrade without extensions

Extensions have been the supported installation method for many releases. If you still rely on an unpackaged installation, perform a HARD UPGRADE into a database created with extensions and migrate your data there.

3.4.1.3 Hard upgrade

By HARD UPGRADE we mean full dump/reload of postgis-enabled databases. You need a HARD UPGRADE when PostGIS objects' internal storage changes or when SOFT UPGRADE is not possible. The Release Notes appendix reports for each version whether you need a dump/reload (HARD UPGRADE) to upgrade.

The dump/reload process is assisted by the **postgis_restore** script which takes care of skipping from the dump all definitions which belong to PostGIS (including old ones), allowing you to restore your schemas and data into a database with PostGIS installed without getting duplicate symbol errors or bringing forward deprecated objects.

Supplementary instructions for windows users are available in the PostGIS Windows installation guide.

The Procedure is as follows:

1. Create a "custom-format" dump of the database you want to upgrade (let's call it `olddb`) include binary blobs (-b) and verbose (-v) output. The user can be the owner of the db, need not be postgres super account.

   Code

   ```
   pg_dump -h localhost -p 5432 -U postgres -Fc -b -v -f "/somepath/olddb.backup" olddb
   ```

2. Do a fresh install of PostGIS in a new database -- we'll refer to this database as `newdb`. Please refer to Section 3.3.2 and Section 3.3.1 for instructions on how to do this.

   The spatial_ref_sys entries found in your dump will be restored, but they will not override existing ones in spatial_ref_sys. This is to ensure that fixes in the official set will be properly propagated to restored databases. If for any reason you really want your own overrides of standard entries just don't load the spatial_ref_sys.sql file when creating the new db.

   If your database is really old or you know you've been using long deprecated functions in your views and functions, you might need to load `legacy.sql` for all your functions and views etc. to properly come back. Only do this if _really_ needed. Consider upgrading your views and functions before dumping instead, if possible. The deprecated functions can be later removed by loading `uninstall_legacy.sql`.

3. Restore your backup into your fresh `newdb` database using postgis_restore. Unexpected errors, if any, will be printed to the standard error stream by psql. Keep a log of those.

   Code

   ```
   postgis_restore "/somepath/olddb.backup" | psql -h localhost -p 5432 -U postgres newdb ↩
       2> errors.txt
   ```

Errors may arise in the following cases:

1. Some of your views or functions make use of deprecated PostGIS objects. In order to fix this you may try loading `legacy.sql` script prior to restore or you'll have to restore to a version of PostGIS which still contains those objects and try a migration again after porting your code. If the `legacy.sql` way works for you, don't forget to fix your code to stop using deprecated functions and drop them loading `uninstall_legacy.sql`.

2. Some custom records of spatial_ref_sys in dump file have an invalid SRID value. Valid SRID values are bigger than 0 and smaller than 999000. Values in the 999000.999999 range are reserved for internal use while values > 999999 can't be used at all. All your custom records with invalid SRIDs will be retained, with those > 999999 moved into the reserved range, but the spatial_ref_sys table would lose a check constraint guarding for that invariant to hold and possibly also its primary key (when multiple invalid SRIDS get converted to the same reserved SRID value).

   In order to fix this you should copy your custom SRS to a SRID with a valid value (maybe in the 910000..910999 range), convert all your tables to the new srid (see UpdateGeometrySRID), delete the invalid entry from spatial_ref_sys and re-construct the check(s) with:

   Code

   ```
   ALTER TABLE spatial_ref_sys ADD CONSTRAINT spatial_ref_sys_srid_check check (srid > 0 ↩
       AND srid < 999000 );
   ```

   Code

   ```
   ALTER TABLE spatial_ref_sys ADD PRIMARY KEY(srid));
   ```

   If you are upgrading an old database containing french IGN cartography, you will have probably SRIDs out of range and you will see, when importing your database, issues like this :

   Code

   ```
    WARNING: SRID 310642222 converted to 999175 (in reserved zone)
   ```

   In this case, you can try following steps : first throw out completely the IGN from the sql which is resulting from postgis_restore. So, after having run :

   Code

   ```
   postgis_restore "/somepath/olddb.backup" > olddb.sql
   ```

   run this command :

   Code

   ```
   grep -v IGNF olddb.sql > olddb-without-IGN.sql
   ```

   Create your `newdb`, activate the required PostGIS extensions, and insert the French IGN spatial reference definitions using this script. After these operations, import your data:

   Code

   ```
   psql -h localhost -p 5432 -U postgres -d newdb -f olddb-without-IGN.sql  2> errors.txt
   ```

Chapter 4

Data Management

4.1 Spatial Data Model

4.1.1 OGC Geometry

The Open Geospatial Consortium (OGC) developed the *Simple Features Access* standard (SFA) to provide a model for geospatial data. It defines the fundamental spatial type of **Geometry**, along with operations which manipulate and transform geometry values to perform spatial analysis tasks. PostGIS implements the OGC Geometry model as the PostgreSQL data types geometry and geography.

Geometry is an *abstract* type. Geometry values belong to one of its *concrete* subtypes which represent various kinds and dimensions of geometric shapes. These include the **atomic** types Point, LineString, LinearRing and Polygon, and the **collection** types MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. The *Simple Features Access - Part 1: Common architecture v1.2.1* adds subtypes for the structures PolyhedralSurface, Triangle and TIN.

Geometry models shapes in the 2-dimensional Cartesian plane. The PolyhedralSurface, Triangle, and TIN types can also represent shapes in 3-dimensional space. The size and location of shapes are specified by their **coordinates**. Each coordinate has a X and Y **ordinate** value determining its location in the plane. Shapes are constructed from points or line segments, with points specified by a single coordinate, and line segments by two coordinates.

Coordinates may contain optional Z and M ordinate values. The Z ordinate is often used to represent elevation. The M ordinate contains a measure value, which may represent time or distance. If Z or M values are present in a geometry value, they must be defined for each point in the geometry. If a geometry has Z or M ordinates the **coordinate dimension** is 3D; if it has both Z and M the coordinate dimension is 4D. The coordinate dimension is at least 2, because every geometry has at least X and Y coordinates.

Geometry values are associated with a **spatial reference system** indicating the coordinate system in which it is embedded. The spatial reference system is identified by the geometry SRID number. The units of the X and Y axes are determined by the spatial reference system. In **planar** reference systems the X and Y coordinates typically represent easting and northing, while in **geodetic** systems they represent longitude and latitude. SRID 0 represents an infinite Cartesian plane with no units assigned to its axes. See Section 4.5.

The geometry **dimension** is a property of geometry types. Point types have dimension 0, linear types have dimension 1, polygonal types have dimension 2, and solid types have dimension 3. Collections have the dimension of the maximum element dimension.

A geometry value may be **empty**. Empty values contain no vertices (for atomic geometry types) or no elements (for collections).

An important property of geometry values is their spatial **extent** or **bounding box**, which the OGC model calls **envelope**. This is the 2 or 3-dimensional box which encloses the coordinates of a geometry. It is an efficient way to represent a geometry's extent in coordinate space and to check whether two geometries interact.

The geometry model allows evaluating topological spatial relationships as described in Section 5.1.1. To support this the concepts of **interior**, **boundary** and **exterior** are defined for each geometry type. Geometries are topologically closed, so they always contain their boundary. The boundary is a geometry of dimension one less than that of the geometry itself. For example, the

boundary of a Polygon is its rings (LineStrings), the boundary of a LineString is its endpoints (Points), and the boundary of a Point is empty.

The OGC geometry model defines validity rules for each geometry type. These rules ensure that geometry values represents realistic situations (e.g. it is possible to specify a polygon with a hole lying outside its shell, but this makes no sense geometrically and is thus invalid). PostGIS also allows storing and manipulating invalid geometry values. This allows detecting and fixing them if needed. See Section 4.4

4.1.1.1 Point

A Point is a 0-dimensional geometry that represents a single location in coordinate space.

Code

```
POINT (1 2)
POINT Z (1 2 3)
POINT ZM (1 2 3 4)
```

4.1.1.2 LineString

A LineString is a 1-dimensional line formed by a contiguous sequence of line segments. Each line segment is defined by two points, with the end point of one segment forming the start point of the next segment. An OGC-valid LineString has either zero or two or more points, but PostGIS also allows single-point LineStrings. LineStrings may cross themselves (self-intersect). A LineString is **closed** if the start and end points are the same. A LineString is **simple** if it does not self-intersect.

Code

```
LINESTRING (1 2, 3 4, 5 6)
```

4.1.1.3 LinearRing

A LinearRing is a LineString which is both closed and simple. The first and last points must be equal, and the line must not self-intersect.

Code

```
LINEARRING (0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0)
```

4.1.1.4 Polygon

A Polygon is a 2-dimensional planar region, delimited by an exterior boundary (the shell) and zero or more interior boundaries (holes). Each boundary is a LinearRing.

Code

```
POLYGON ((0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0), (1 1 0, 2 1 0, 2 2 0, 1 2 0, 1 1 0))
```

4.1.1.5 MultiPoint

A MultiPoint is a collection of Points.

Code

```
MULTIPOINT ( (0 0), (1 2) )
```

4.1.1.6 MultiLineString

A MultiLineString is a collection of LineStrings. A MultiLineString is closed if each of its elements is closed.

Code

```
MULTILINESTRING ( (0 0, 1 1, 1 2), (2 3, 3 2, 5 4) )
```

4.1.1.7 MultiPolygon

A MultiPolygon is a collection of non-overlapping, non-adjacent Polygons. Polygons in the collection may touch only at a finite number of points. (Two polygons are adjacent if they share an edge. They touch if they share only points or edges on their boundaries). For more details on MultiPolygon validity, see Section 4.4.

Code

```
MULTIPOLYGON (((1 5, 5 5, 5 1, 1 1, 1 5)), ((6 5, 9 1, 6 1, 6 5)))
```

4.1.1.8 GeometryCollection

A GeometryCollection is a heterogeneous (mixed) collection of geometries.

Code

```
GEOMETRYCOLLECTION ( POINT(2 3), LINESTRING(2 3, 3 4))
```

4.1.1.9 PolyhedralSurface

A PolyhedralSurface is a contiguous collection of patches or facets which share some edges. Each patch is a planar Polygon. If the Polygon coordinates have Z ordinates then the surface is 3-dimensional.

Code

```
POLYHEDRALSURFACE Z (
((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )
```

4.1.1.10 Triangle

A Triangle is a polygon defined by three distinct non-collinear vertices. Because a Triangle is a polygon it is specified by four coordinates, with the first and fourth being equal.

Code

```
TRIANGLE ((0 0, 0 9, 9 0, 0 0))
```

4.1.1.11 TIN

A TIN is a collection of non-overlapping Triangles representing a Triangulated Irregular Network.

Code

```
TIN Z ( ((0 0 0, 0 0 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 0 0 0)) )
```

4.1.2 SQL/MM Part 3 - Curves

The *ISO/IEC 13249-3 SQL Multimedia - Spatial* standard (SQL/MM) extends the OGC SFA to define Geometry subtypes containing curves with circular arcs. The SQL/MM types support 3DM, 3DZ and 4D coordinates.



Note
All floating point comparisons within the SQL-MM implementation are performed to a specified tolerance, currently 1E-8.

4.1.2.1 CircularString

CircularString is the basic curve type, similar to a LineString in the linear world. A single arc segment is specified by three points: the start and end points (first and third) and some other point on the arc. To specify a closed circle the start and end points are the same and the middle point is the opposite point on the circle diameter (which is the center of the arc). In a sequence of arcs the end point of the previous arc is the start point of the next arc, just like the segments of a LineString. This means that a CircularString must have an odd number of points greater than 1.

Code

```
CIRCULARSTRING(0 0, 1 1, 1 0)

CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0)
```

4.1.2.2 CompoundCurve

A CompoundCurve is a single continuous curve that may contain both circular arc segments and linear segments. That means that in addition to having well-formed components, the end point of every component (except the last) must be coincident with the start point of the following component.

Code

```
COMPOUNDCURVE( CIRCULARSTRING(0 0, 1 1, 1 0), (1 0, 0 1))
```

4.1.2.3 CurvePolygon

A CurvePolygon is like a polygon, with an outer ring and zero or more inner rings. The difference is that a ring can be a CircularString or CompoundCurve as well as a LineString.

As of PostGIS 1.4 PostGIS supports compound curves in a curve polygon.

Code

```
CURVEPOLYGON(
CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),
(1 1, 3 3, 3 1, 1 1) )
```

Example: A CurvePolygon with the shell defined by a CompoundCurve containing a CircularString and a LineString, and a hole defined by a CircularString

Code

```
CURVEPOLYGON(
COMPOUNDCURVE( CIRCULARSTRING(0 0, 2 0, 2 1, 2 3, 4 3),
               (4 3, 4 5, 1 4, 0 0)),
CIRCULARSTRING(1.7 1, 1.4 0.4, 1.6 0.4, 1.6 0.5, 1.7 1) )
```

4.1.2.4 MultiCurve

A MultiCurve is a collection of curves which can include LineStrings, CircularStrings or CompoundCurves.

Code

```
MULTICURVE( (0 0, 5 5), CIRCULARSTRING(4 0, 4 4, 8 4))
```

4.1.2.5 MultiSurface

A MultiSurface is a collection of surfaces, which can be (linear) Polygons or CurvePolygons.

Code

```
MULTISURFACE(
CURVEPOLYGON(
  CIRCULARSTRING( 0 0, 4 0, 4 4, 0 4, 0 0),
  (1 1, 3 3, 3 1, 1 1)),
((10 10, 14 12, 11 10, 10 10), (11 11, 11.5 11, 11 11.5, 11 11)))
```

4.1.3 WKT and WKB

The OGC SFA specification defines two formats for representing geometry values for external use: Well-Known Text (WKT) and Well-Known Binary (WKB). Both WKT and WKB include information about the type of the object and the coordinates which define it.

Well-Known Text (WKT) provides a standard textual representation of spatial data. Examples of WKT representations of spatial objects are:

- POINT(0 0)
- POINT Z (0 0 0)
- POINT ZM (0 0 0 0)
- POINT EMPTY
- LINESTRING(0 0,1 1,1 2)
- LINESTRING EMPTY
- POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1, 2 1, 2 2, 1 2,1 1))
- MULTIPOINT((0 0),(1 2))
- MULTIPOINT Z ((0 0 0),(1 2 3))

- MULTIPOINT EMPTY
- MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))
- MULTIPOLYGON(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))
- GEOMETRYCOLLECTION(POINT(2 3), LINESTRING(2 3,3 4))
- GEOMETRYCOLLECTION EMPTY

Input and output of WKT is provided by the functions ST_AsText and ST_GeomFromText:

Code

```
text WKT = geometry;
geometry = ST_GeomFromText(text WKT, SRID);
```

For example, a statement to create and insert a spatial object from WKT with SRID 312 is:

Code

```
INSERT INTO geotable ( geom, name )
VALUES ( ST_GeomFromText('POINT(-126.4 45.32)', 312), 'A Place');
```

Well-Known Binary (WKB) provides a portable, full-precision representation of spatial data as binary data (arrays of bytes). Examples of the WKB representations of spatial objects are:

- WKT: POINT(1 1)

  WKB: 010100000000000000000F03F000000000000F03
- WKT: LINESTRING (2 2, 9 9)

  WKB: 0102000000020000000000000000000040000000000000004000000000000022400000000000002240

Input and output of WKB is provided by the functions ST_AsBinary and ST_GeomFromWKB:

Code

```
bytea WKB = ST_AsBinary(geometry);
geometry = ST_GeomFromWKB(bytea WKB, SRID);
```

For example, a statement to create and insert a spatial object from WKB is:

Code

```
INSERT INTO geotable ( geom, name )
VALUES ( ST_GeomFromWKB('\x010100000000000000000f03f000000000000f03f', 312), 'A Place');
```

4.2 Geometry Data Type

PostGIS implements the OGC Simple Features model by defining a PostgreSQL data type called `geometry`. It represents all of the geometry subtypes by using an internal type code (see GeometryType and ST_GeometryType). This allows modelling spatial features as rows of tables defined with a column of type `geometry`.

The `geometry` data type is *opaque*, which means that all access is done via invoking functions on geometry values. Functions allow creating geometry objects, accessing or updating all internal fields, and computing new geometry values. PostGIS supports all the functions specified in the OGC *Simple feature access - Part 2: SQL option* (SFS) specification, as well many others. See Chapter 7 for the full list of functions.



Note

PostGIS follows the SFA standard by prefixing spatial functions with "ST_". This was intended to stand for "Spatial and Temporal", but the temporal part of the standard was never developed. Instead it can be interpreted as "Spatial Type".

The SFA standard specifies that spatial objects include a Spatial Reference System identifier (SRID). The SRID is required when creating spatial objects for insertion into the database (it may be defaulted to 0). See ST_SRID and Section 4.5

To make querying geometry efficient PostGIS defines various kinds of spatial indexes, and spatial operators to use them. See Section 4.9 and Section 5.2 for details.

4.2.1 PostGIS EWKB and EWKT

OGC SFA specifications initially supported only 2D geometries, and the geometry SRID is not included in the input/output representations. The OGC SFA specification 1.2.1 (which aligns with the ISO 19125 standard) adds support for 3D (ZYZ) and measured (XYM and XYZM) coordinates, but still does not include the SRID value.

Because of these limitations PostGIS defined extended EWKB and EWKT formats. They provide 3D (XYZ and XYM) and 4D (XYZM) coordinate support and include SRID information. Including all geometry information allows PostGIS to use EWKB as the format of record (e.g. in DUMP files).

EWKB and EWKT are used for the "canonical forms" of PostGIS data objects. For input, the canonical form for binary data is EWKB, and for text data either EWKB or EWKT is accepted. This allows geometry values to be created by casting a text value in either HEXEWKB or EWKT to a geometry value using `::geometry`. For output, the canonical form for binary is EWKB, and for text it is HEXEWKB (hex-encoded EWKB).

For example this statement creates a geometry by casting from an EWKT text value, and outputs it using the canonical form of HEXEWKB:

Code

```
SELECT 'SRID=4;POINT(0 0)'::geometry;
```

Output

```
geometry
----------------------------------------------------
0101000020040000000000000000000000000000000000000000
```

PostGIS EWKT output has a few differences to OGC WKT:

- For 3DZ geometries the Z qualifier is omitted:
  OGC: POINT Z (1 2 3)
  EWKT: POINT (1 2 3)
- For 3DM geometries the M qualifier is included:
  OGC: POINT M (1 2 3)
  EWKT: POINTM (1 2 3)
- For 4D geometries the ZM qualifier is omitted:
  OGC: POINT ZM (1 2 3 4)
  EWKT: POINT (1 2 3 4)

EWKT avoids over-specifying dimensionality and the inconsistencies that can occur with the OGC/ISO format, such as:

- POINT ZM (1 1)
- POINT ZM (1 1 1)

- POINT (1 1 1 1)



Caution
PostGIS extended formats are currently a superset of the OGC ones, so that every valid OGC WKB/WKT is also valid EWKB/EWKT. However, this might vary in the future, if the OGC extends a format in a way that conflicts with the PosGIS definition. Thus you SHOULD NOT rely on this compatibility!

Examples of the EWKT text representation of spatial objects are:

- POINT(0 0 0) -- XYZ
- SRID=32632;POINT(0 0) -- XY with SRID
- POINTM(0 0 0) -- XYM
- POINT(0 0 0 0) -- XYZM
- SRID=4326;MULTIPOINTM(0 0 0,1 2 1) -- XYM with SRID
- MULTILINESTRING((0 0 0,1 1 0,1 2 1),(2 3 1,3 2 1,5 4 1))
- POLYGON((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0))
- MULTIPOLYGON(((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0)),((-1 -1 0,-1 -2 0,-2 -2 0,-2 -1 0,-1 -1 0)))
- GEOMETRYCOLLECTIONM(POINTM(2 3 9), LINESTRINGM(2 3 4, 3 4 5))
- MULTICURVE((0 0, 5 5), CIRCULARSTRING(4 0, 4 4, 8 4))
- POLYHEDRALSURFACE(((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)), ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)), ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)))
- TRIANGLE ((0 0, 0 10, 10 0, 0 0))
- TIN(((0 0 0, 0 0 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 0 0 0)))

Input and output using these formats is available using the following functions:

Code

```
bytea EWKB = ST_AsEWKB(geometry);
text EWKT = geometry;
geometry = ST_GeomFromEWKB(bytea EWKB);
geometry = ST_GeomFromEWKT(text EWKT);
```

For example, a statement to create and insert a PostGIS spatial object using EWKT is:

Code

```
INSERT INTO geotable ( geom, name )
VALUES ( ST_GeomFromEWKT('SRID=312;POINTM(-126.4 45.32 15)'), 'A Place' )
```

4.3 Geography Data Type

The PostGIS `geography` data type provides native support for spatial features represented on "geographic" coordinates (sometimes called "geodetic" coordinates, or "lat/lon", or "lon/lat"). Geographic coordinates are spherical coordinates expressed in angular units (degrees).

The basis for the PostGIS geometry data type is a plane. The shortest path between two points on the plane is a straight line. That means functions on geometries (areas, distances, lengths, intersections, etc) are calculated using straight line vectors and cartesian mathematics. This makes them simpler to implement and faster to execute, but also makes them inaccurate for data on the spheroidal surface of the earth.

The PostGIS geography data type is based on a spherical model. The shortest path between two points on the sphere is a great circle arc. Functions on geographies (areas, distances, lengths, intersections, etc) are calculated using arcs on the sphere. By taking the spheroidal shape of the world into account, the functions provide more accurate results.

Because the underlying mathematics is more complicated, there are fewer functions defined for the geography type than for the geometry type. Over time, as new algorithms are added the capabilities of the geography type will expand. As a workaround one can convert back and forth between geometry and geography types.

Like the geometry data type, geography data is associated with a spatial reference system via a spatial reference system identifier (SRID). Any geodetic (long/lat based) spatial reference system defined in the `spatial_ref_sys` table can be used. You can add your own custom geodetic spatial reference system as described in Section 4.5.2.

For all spatial reference systems the units returned by measurement functions (e.g. ST_Distance, ST_Length, ST_Perimeter, ST_Area) and for the distance argument of ST_DWithin are in meters.

4.3.1 Creating Geography Tables

You can create a table to store geography data using the CREATE TABLE SQL statement with a column of type `geography`. The following example creates a table with a geography column storing 2D LineStrings in the WGS84 geodetic coordinate system (SRID 4326):

Code

```
CREATE TABLE global_points (
  id SERIAL PRIMARY KEY,
  name VARCHAR(64),
  location geography(POINT, 4326)
);
```

The geography type supports two optional type modifiers:

- the spatial type modifier restricts the kind of shapes and dimensions allowed in the column. Values allowed for the spatial type are: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION. The geography type does not support curves, TINS, or POLYHEDRALSURFACEs. The modifier supports coordinate dimensionality restrictions by adding suffixes: Z, M and ZM. For example, a modifier of 'LINESTRINGM' only allows linestrings with three dimensions, and treats the third dimension as a measure. Similarly, 'POINTZM' requires four dimensional (XYZM) data.
- the SRID modifier restricts the spatial reference system SRID to a particular number. If omitted, the SRID defaults to 4326 (WGS84 geodetic), and all calculations are performed using WGS84.

Examples of creating tables with geography columns:

Create a table with 2D POINT geography with the default SRID 4326 (WGS84 long/lat):

Code

```
CREATE TABLE ptgeogwgs(gid serial PRIMARY KEY, geog geography(POINT));
```

Create a table with 2D POINT geography in NAD83 longlat:

Code

```
CREATE TABLE ptgeognad83(gid serial PRIMARY KEY, geog geography(POINT, 4269));
```

Create a table with 3D (XYZ) POINTs and an explicit SRID of 4326:

Code

```
CREATE TABLE ptzgeogwgs84(gid serial PRIMARY KEY, geog geography(POINTZ, 4326));
```

Create a table with 2D LINESTRING geography with the default SRID 4326:

Code

```
CREATE TABLE lgeog(gid serial PRIMARY KEY, geog geography(LINESTRING));
```

Create a table with 2D POLYGON geography with the SRID 4267 (NAD 1927 long lat):

Code

```
CREATE TABLE lgeognad27(gid serial PRIMARY KEY, geog geography(POLYGON, 4267));
```

Geography fields are registered in the `geography_columns` system view. You can query the `geography_columns` view and see that the table is listed:

Code

```
SELECT * FROM geography_columns;
```

Creating a spatial index works the same as for geometry columns. PostGIS will note that the column type is GEOGRAPHY and create an appropriate sphere-based index instead of the usual planar index used for GEOMETRY.

Code

```
CREATE INDEX global_points_gix ON global_points USING GIST ( location );
```

4.3.2 Using Geography Tables

You can insert data into geography tables in the same way as geometry. Geometry data will autocast to the geography type if it has SRID 4326. The EWKT and EWKB formats can also be used to specify geography values.

Code

```
INSERT INTO global_points (name, location) VALUES ('Town', 'SRID=4326;POINT(-110 30)');
INSERT INTO global_points (name, location) VALUES ('Forest', 'SRID=4326;POINT(-109 29)');
INSERT INTO global_points (name, location) VALUES ('London', 'SRID=4326;POINT(0 49)');
```

Any geodetic (long/lat) spatial reference system listed in `spatial_ref_sys` table may be specified as a geography SRID. Non-geodetic coordinate systems raise an error if used.

A NAD83 longitude/latitude geography value:

Code

```
SELECT 'SRID=4269;POINT(-123 34)'::geography;
```

Output

```
                  geography
----------------------------------------------------
 0101000020AD10000000000000C05EC00000000000004140
```

A NAD27 longitude/latitude geography value:

Code

```
SELECT 'SRID=4267;POINT(-123 34)'::geography;
```

Output

```
                    geography
----------------------------------------------------
 0101000020AB1000000000000000C05EC00000000000004140
```

NAD83 UTM zone meters fails because it is a meter-based planar projection.

Code

```
SELECT 'SRID=26910;POINT(-123 34)'::geography;
```

Output

```
ERROR:  Only lon/lat coordinate systems are supported in geography.
```

Query and measurement functions use units of meters. So distance parameters should be expressed in meters, and return values should be expected in meters (or square meters for areas).

This query finds rows within a 1000 km tolerance:

Code

```
SELECT name FROM global_points WHERE ST_DWithin(location, 'SRID=4326;POINT(-110 29)':: ↩
    geography, 1000000);
```

You can see the power of geography in action by calculating how close a plane flying a great circle route from Seattle to London (LINESTRING(-122.33 47.606, 0.0 51.5)) comes to Reykjavik (POINT(-21.96 64.15)) (map the route).

The geography type calculates the true shortest distance of 122.235 km over the sphere between Reykjavik and the great circle flight path between Seattle and London.

Code

```
SELECT ST_Distance('LINESTRING(-122.33 47.606,0.0 51.5)'::geography, 'POINT(-21.96 64.15)':: ↩
    geography);
```

Output

```
   st_distance
-----------------
 122235.23815667
```

The geometry type calculates a meaningless cartesian distance between Reykjavik and the straight line path from Seattle to London plotted on a flat map of the world. The nominal units of the result is "degrees", but the result doesn't correspond to any true angular difference between the points, so even calling them "degrees" is inaccurate.

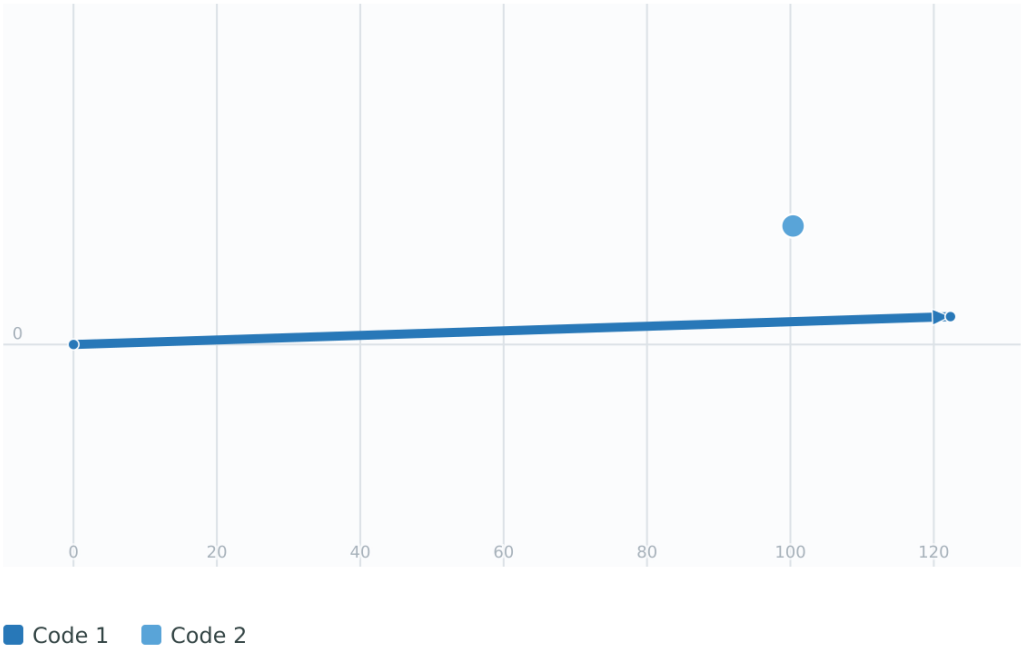
Code

```
SELECT ST_Distance('LINESTRING(-122.33 47.606,0.0 51.5)'::geometry, 'POINT(-21.96 64.15)':: ↩
    geometry);
```

Output

```
      st_distance
--------------------
 13.342271221453624
```

Figure



4.3.3 When to use the Geography data type

The geography data type allows you to store data in longitude/latitude coordinates, but at a cost: there are fewer functions defined on GEOGRAPHY than there are on GEOMETRY; those functions that are defined take more CPU time to execute.

The data type you choose should be determined by the expected working area of the application you are building. Will your data span the globe or a large continental area, or is it local to a state, county or municipality?

- If your data is contained in a small area, you might find that choosing an appropriate projection and using GEOMETRY is the best solution, in terms of performance and functionality available.
- If your data is global or covers a continental region, you may find that GEOGRAPHY allows you to build a system without having to worry about projection details. You store your data in longitude/latitude, and use the functions that have been defined on GEOGRAPHY.
- If you don't understand projections, and you don't want to learn about them, and you're prepared to accept the limitations in functionality available in GEOGRAPHY, then it might be easier for you to use GEOGRAPHY than GEOMETRY. Simply load your data up as longitude/latitude and go from there.

Refer to Section 13.11 for compare between what is supported for Geography vs. Geometry. For a brief listing and description of Geography functions, refer to Section 13.4

4.3.4 Geography Advanced FAQ

1. *Do you calculate on the sphere or the spheroid?*

   By default, all distance and area calculations are done on the spheroid. You should find that the results of calculations in local areas match up well with local planar results in good local projections. Over larger areas, the spheroidal calculations will be more accurate than any calculation done on a projected plane. The difference is that the spheroid takes into account the curvature and non-spherical shape of the Earth, whereas planar calculations assume a flat surface. All the geography functions have the option of using a sphere calculation, by setting a final boolean parameter to 'FALSE'. This will somewhat speed up calculations, particularly for cases where the geometries are very simple.

2. *What about the date-line and the poles?*

   All the calculations have no conception of date-line or poles, the coordinates are spherical (longitude/latitude) so a shape that crosses the dateline is, from a calculation point of view, no different from any other shape.

3. *What is the longest arc you can process?*

   We use great circle arcs as the "interpolation line" between two points. That means any two points are actually joined up two ways, depending on which direction you travel along the great circle. All our code assumes that the points are joined by the *shorter* of the two paths along the great circle. As a consequence, shapes that have arcs of more than 180 degrees will not be correctly modelled.

4. *Why is it so slow to calculate the area of Europe / Russia / insert big geographic region here ?*

   Because the polygon is so darned huge! Big areas are bad for two reasons: their bounds are huge, so the index tends to pull the feature no matter what query you run; the number of vertices is huge, and tests (distance, containment) have to traverse the vertex list at least once and sometimes N times (with N being the number of vertices in the other candidate feature). As with GEOMETRY, we recommend that when you have very large polygons, but are doing queries in small areas, you "denormalize" your geometric data into smaller chunks so that the index can effectively subquery parts of the object and so queries don't have to pull out the whole object every time. Please consult ST_Subdivide function documentation. Just because you *can* store all of Europe in one polygon doesn't mean you *should*.

4.4 Geometry Validation

PostGIS is compliant with the Open Geospatial Consortium's (OGC) Simple Features specification. That standard defines the concepts of geometry being *simple* and *valid*. These definitions allow the Simple Features geometry model to represent spatial objects in a consistent and unambiguous way that supports efficient computation. (Note: the OGC SF and SQL/MM have the same definitions for simple and valid.)

4.4.1 Simple Geometry

A *simple* geometry is one that has no anomalous geometric points, such as self intersection or self tangency.

A `POINT` is inherently *simple* as a 0-dimensional geometry object.

`MULTIPOINT`s are *simple* if no two coordinates (`POINT`s) are equal (have identical coordinate values).

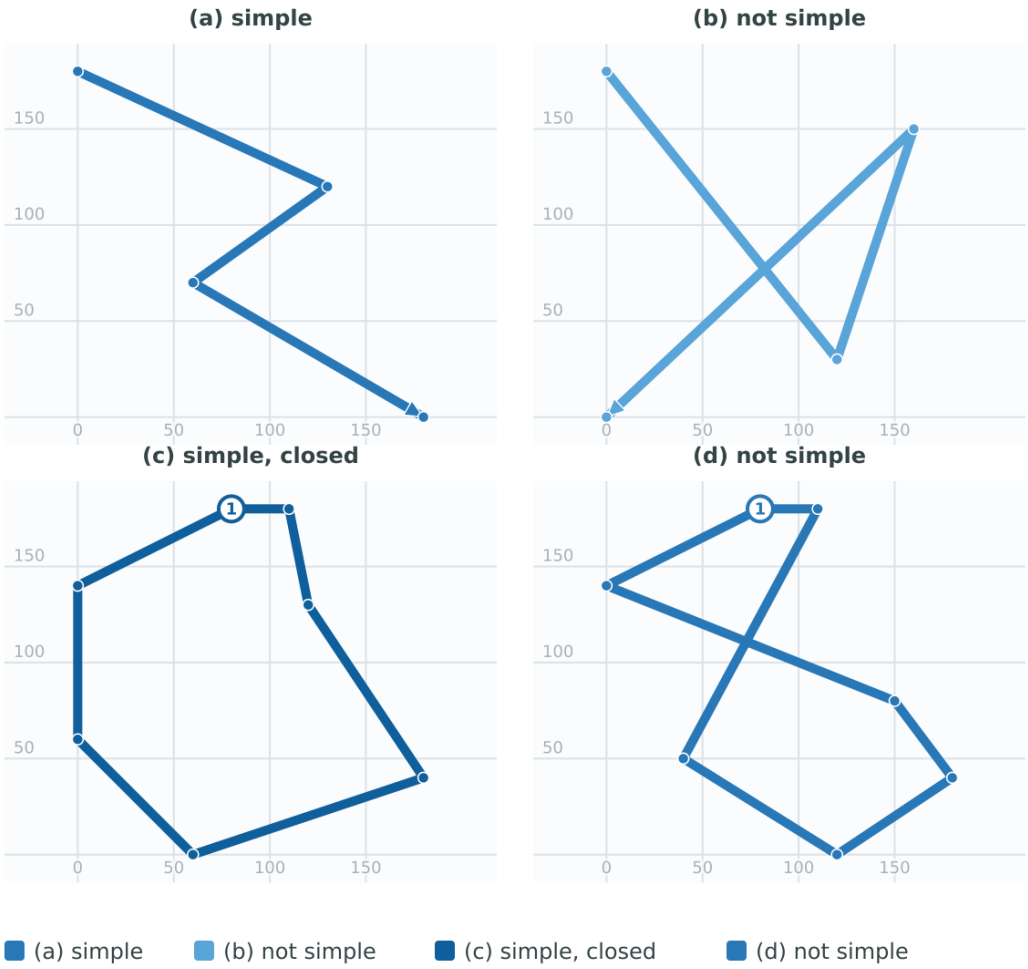
A `LINESTRING` is *simple* if it does not pass through the same point twice, except for the endpoints. If the endpoints of a simple LineString are identical it is called *closed* and referred to as a Linear Ring.

Use ST_IsSimple to classify linear geometries. Each labeled result below is returned only when the function agrees with the stated classification; example **(c)** also verifies that the simple LineString is closed.

Code

```
WITH cases(label, geom) AS (VALUES
  ('a', 'LINESTRING (10 190, 140 130, 70 80, 190 10)'::geometry),
  ('b', 'LINESTRING (10 190, 130 40, 170 160, 10 10)'::geometry),
  ('c', 'LINESTRING (90 190, 120 190, 130 140, 190 50, 70 10, 10 70, 10 150, 90 190)':: ↩
    geometry),
  ('d', 'LINESTRING (90 190, 120 190, 50 60, 130 10, 190 50, 160 90, 10 150, 90 190)':: ↩
    geometry)
)
SELECT
  (SELECT geom FROM cases WHERE label = 'a' AND ST_IsSimple(geom))
    AS "input_(a) simple",
  (SELECT geom FROM cases WHERE label = 'b' AND NOT ST_IsSimple(geom))
    AS "input_(b) not simple",
  (SELECT geom FROM cases
             WHERE label = 'c' AND ST_IsSimple(geom) AND ST_IsClosed(geom))
    AS "input_(c) simple, closed",
  (SELECT geom FROM cases WHERE label = 'd' AND NOT ST_IsSimple(geom))
    AS "input_(d) not simple";
```

Figure



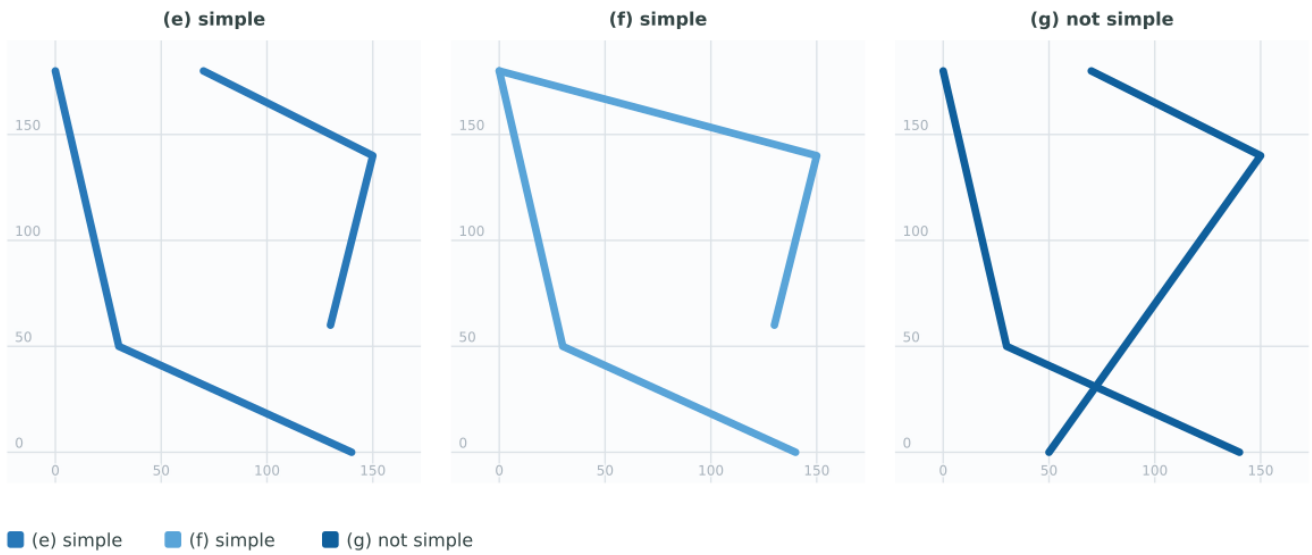
A `MULTILINESTRING` is *simple* only if all of its elements are simple and the only intersection between any two elements occurs at points that are on the boundaries of both elements.

The next example applies the same classification to the three MultiLineStrings. In **(f)** the component lines meet only at a shared endpoint, while in **(g)** they cross in their interiors.

Code

```
WITH cases(label, geom) AS (VALUES
  ('e', 'MULTILINESTRING ((30 190, 60 60, 170 10), (100 190, 180 150, 160 70))'::geometry),
  ('f', 'MULTILINESTRING ((30 190, 60 60, 170 10), (30 190, 180 150, 160 70))'::geometry),
  ('g', 'MULTILINESTRING ((30 190, 60 60, 170 10), (100 190, 180 150, 80 10))'::geometry)
)
SELECT
  (SELECT geom FROM cases WHERE label = 'e' AND ST_IsSimple(geom))
    AS "input_(e) simple",
  (SELECT geom FROM cases WHERE label = 'f' AND ST_IsSimple(geom))
    AS "input_(f) simple",
  (SELECT geom FROM cases WHERE label = 'g' AND NOT ST_IsSimple(geom))
    AS "input_(g) not simple";
```

Figure



POLYGONs are formed from linear rings, so valid polygonal geometry is always *simple*.

Generally, PostGIS functions do not require geometric arguments to be simple. Simplicity is primarily used as a basis for defining geometric validity. It is also a requirement for some kinds of spatial data models (for example, linear networks often disallow lines that cross). Multipoint and linear geometry can be made simple using ST_UnaryUnion.

4.4.2 Valid Geometry

Geometry validity primarily applies to 2-dimensional geometries (POLYGONs and MULTIPOLYGONs) . Validity is defined by rules that allow polygonal geometry to model planar areas unambiguously.

A POLYGON is *valid* if:

1. the polygon boundary rings (the exterior shell ring and interior hole rings) are *simple* (do not cross or self-touch). Because of this a polygon cannot have cut lines, spikes or loops. This implies that polygon holes must be represented as interior rings, rather than by the exterior ring self-touching (a so-called "inverted hole").
2. boundary rings do not cross
3. boundary rings may touch at points but only as a tangent (i.e. not in a line)
4. interior rings are contained in the exterior ring
5. the polygon interior is simply connected (i.e. the rings must not touch in a way that splits the polygon into more than one part)

Consecutive repeated points in a polygon boundary ring do not make the ring non-simple. Under the SFS curve model, a curve is interpreted by a continuous parameterization; consecutive equal coordinates are topologically indistinguishable from a single vertex. PostGIS therefore treats them as redundant vertices, so they do not make an otherwise valid polygon invalid.

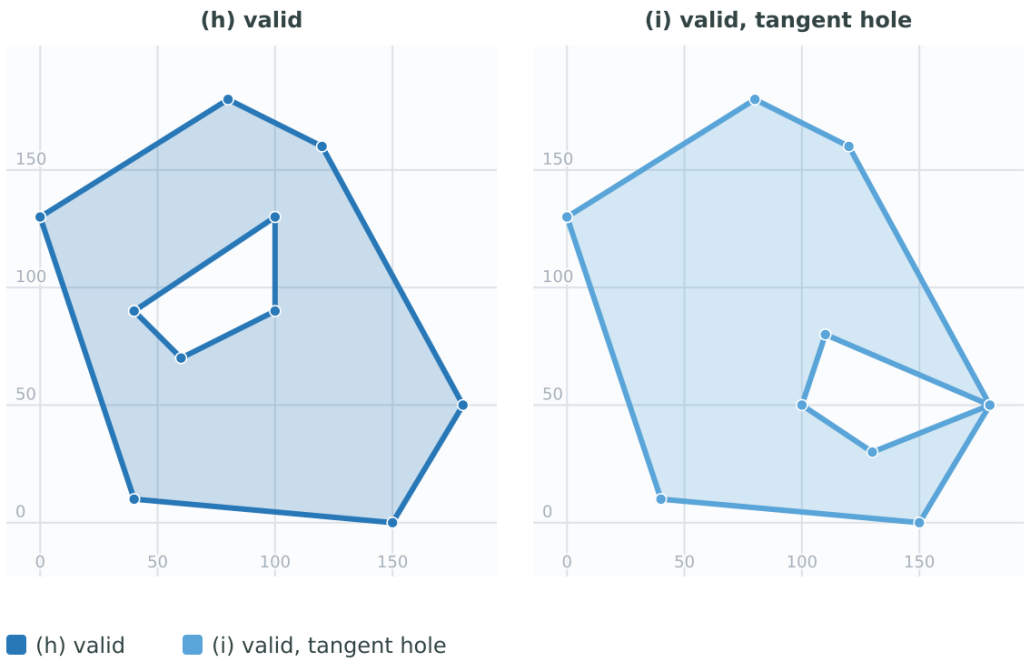
The following examples use ST_IsValid to check the validity rules. Each labeled geometry is returned only if its computed validity agrees with the label.

Code

```
WITH cases(label, geom) AS (VALUES
  ('h', 'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
                  (50 100, 70 80, 110 100, 110 140, 50 100))'::geometry),
```

```
  ('i', 'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
                  (190 60, 140 40, 110 60, 120 90, 190 60))'::geometry)
)
SELECT
  (SELECT geom FROM cases WHERE label = 'h' AND ST_IsValid(geom))
    AS "input_(h) valid",
  (SELECT geom FROM cases WHERE label = 'i' AND ST_IsValid(geom))
    AS "input_(i) valid, tangent hole";
```

Figure



Examples **(j-m)** are invalid. They show, respectively, a disconnected interior, crossing rings, a cut line, and a hole outside the shell. The invalidity reason is checked with ST_IsValidReason. The disconnected interior in **(j)** can instead be represented as a valid `MULTIPOLYGON`.

Code

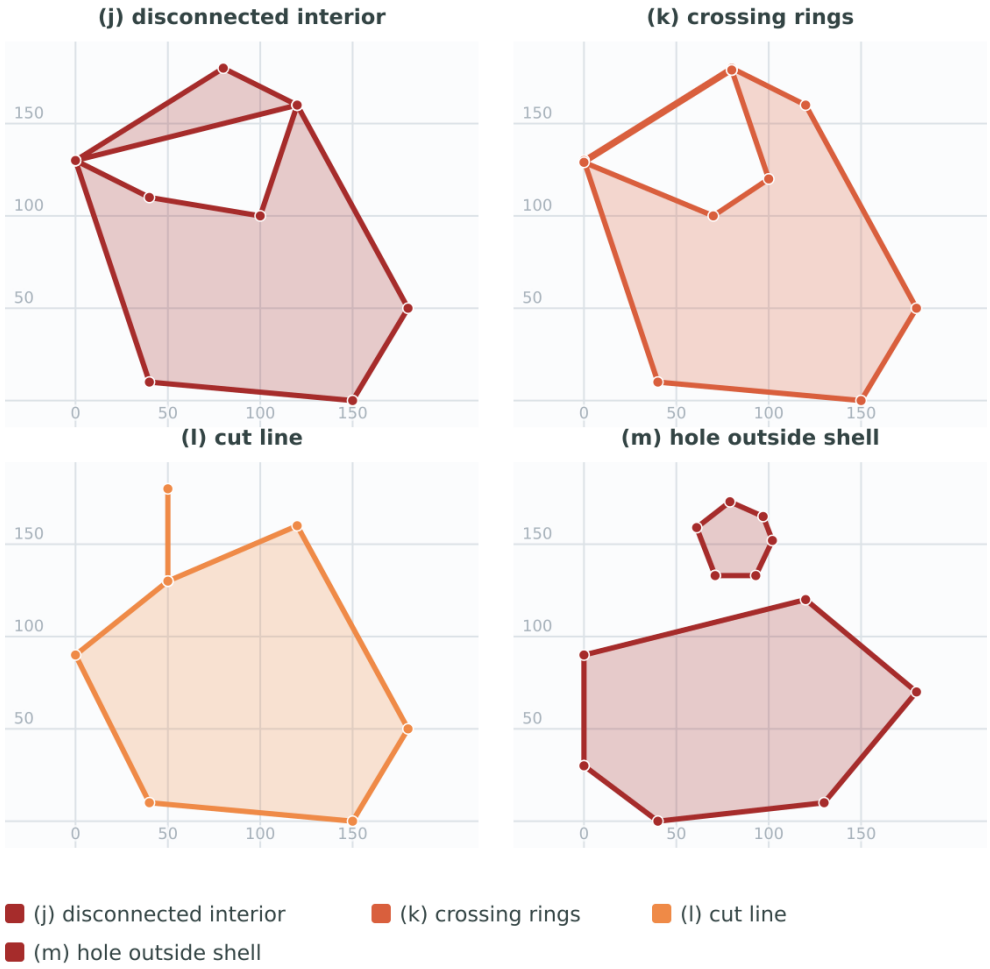
```
WITH cases(label, geom, reason_pattern) AS (VALUES
  ('(j) disconnected interior',
   'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
             (130 170, 10 140, 50 120, 110 110, 130 170))'::geometry,
   'Interior is disconnected%'),
  ('(k) crossing rings',
   'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
             (90 189, 10 139, 80 110, 110 130, 90 189))'::geometry,
   'Self-intersection%'),
  ('(l) cut line',
   'POLYGON ((10 100, 60 140, 60 190, 60 140, 130 170,
              190 60, 160 10, 50 20, 10 100))'::geometry,
   'Ring Self-intersection%'),
  ('(m) hole outside shell',
   'POLYGON ((10 40, 10 100, 130 130, 190 80, 140 20, 50 10, 10 40),
             (81 143, 103 143, 112 162, 107 175, 89 183, 71 169, 81 143))'::geometry,
   'Hole lies outside shell%')
)
SELECT label, geom
FROM cases
WHERE NOT ST_IsValid(geom)
```

```
  AND ST_IsValidReason(geom) LIKE reason_pattern
ORDER BY label;
```

Output

```
          label          | geom
-------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 (j) disconnected interior | POLYGON((10 140,90 190,130 170,190 60,160 10,50 20,10 140),(130 170,10 140,50 120,110 ↩
    110,130 170))
 (k) crossing rings        | POLYGON((10 140,90 190,130 170,190 60,160 10,50 20,10 140),(90 189,10 139,80 110,110 130,90 ↩
     189))
 (l) cut line              | POLYGON((10 100,60 140,60 190,60 140,130 170,190 60,160 10,50 20,10 100))
 (m) hole outside shell    | POLYGON((10 40,10 100,130 130,190 80,140 20,50 10,10 40),(81 143,103 143,112 162,107 175,89 ↩
     183,71 169,81 143))
```

Figure



A `MULTIPOLYGON` is *valid* if:

1. its element `POLYGON`s are valid
2. elements do not overlap (i.e. their interiors must not intersect)
3. elements touch only at points (i.e. not along a line)

The next example checks the MultiPolygon rules. In **(n)** the polygons touch at only one point. In **(o)** they share an edge, and in **(p)** their interiors overlap.

Code

```
WITH cases(label, geom, valid, reason_pattern) AS (VALUES
  ('(n) point touch',
   'MULTIPOLYGON (((10 40, 29 118, 100 75, 180 88, 140 30, 70 10, 10 40)),
                  ((50 170, 100 190, 160 180, 180 140, 180 90, 120 120,
                    30 120, 50 170)))'::geometry,
   true, NULL),
  ('(o) shared edge',
   'MULTIPOLYGON (((10 40, 30 120, 120 120, 180 88, 140 30, 70 10, 10 40)),
                  ((50 170, 100 190, 160 180, 180 140, 180 90, 120 120,
                    30 120, 50 170)))'::geometry,
   false, 'Self-intersection%'),
  ('(p) overlap',
   'MULTIPOLYGON (((10 40, 42 93, 130 150, 180 86, 140 30, 70 10, 10 40)),
                  ((50 170, 100 190, 138 182, 149 156, 146 137, 100 100,
                    58 140, 50 170)))'::geometry,
   false, 'Self-intersection%')
)
SELECT label, geom
FROM cases
WHERE ST_IsValid(geom) = valid
  AND (valid OR ST_IsValidReason(geom) LIKE reason_pattern)
ORDER BY label;
```

Output

```
      label       | geom
------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------

 (n) point touch  | MULTIPOLYGON(((10 40,29 118,100 75,180 88,140 30,70 10,10 40)),((50 170,100 190,160 180,180 140,180 ↩
     90,120 120,30 120,50 170)))
 (o) shared edge  | MULTIPOLYGON(((10 40,30 120,120 120,180 88,140 30,70 10,10 40)),((50 170,100 190,160 180,180  ↩
    140,180 90,120 120,30 120,50 170)))
 (p) overlap      | MULTIPOLYGON(((10 40,42 93,130 150,180 86,140 30,70 10,10 40)),((50 170,100 190,138 182,149 156,146 ↩
     137,100 100,58 140,50 170)))
```

Figure



These rules mean that valid polygonal geometry is also *simple*.

For linear geometry the only validity rule is that `LINESTRING`s must have at least two points and have non-zero length (or equivalently, have at least two distinct points.) Note that non-simple (self-intersecting) lines are valid.

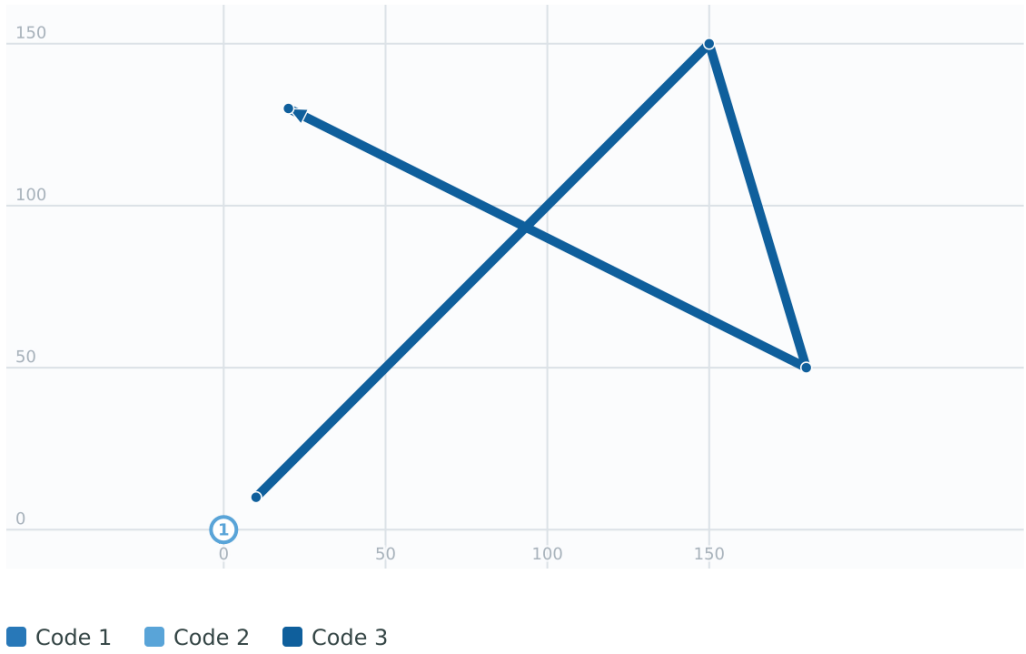
Code

```
SELECT
ST_IsValid('LINESTRING(0 0,1 1)') AS len_nonzero,
ST_IsValid('LINESTRING(0 0,0 0,0 0)') AS len_zero,
ST_IsValid('LINESTRING(10 10,150 150,180 50,20 130)') AS self_int;
```

Output

```
 len_nonzero | len_zero | self_int
-------------+----------+----------
 t           | f        | t
```

Figure



`POINT` and `MULTIPOINT` geometries have no validity rules.

4.4.3 Managing Validity

PostGIS allows creating and storing both valid and invalid Geometry. This allows invalid geometry to be detected and flagged or fixed. There are also situations where the OGC validity rules are stricter than desired (examples of this are zero-length linestrings and polygons with inverted holes.)

Many of the functions provided by PostGIS rely on the assumption that geometry arguments are valid. For example, it does not make sense to calculate the area of a polygon that has a hole defined outside of the polygon, or to construct a polygon from a non-simple boundary line. Assuming valid geometric inputs allows functions to operate more efficiently, since they do not need to check for topological correctness. (Notable exceptions are that zero-length lines and polygons with inversions are generally handled correctly.) Also, most PostGIS functions produce valid geometry output if the inputs are valid. This allows PostGIS functions to be chained together safely.

If you encounter unexpected error messages when calling PostGIS functions (such as "GEOS Intersection() threw an error!"), you should first confirm that the function arguments are valid. If they are not, then consider using one of the techniques below to ensure the data you are processing is valid.



Note
If a function reports an error with valid inputs, then you may have found an error in either PostGIS or one of the libraries it uses, and you should report this to the PostGIS project. The same is true if a PostGIS function returns an invalid geometry for valid input.

To test if a geometry is valid use the ST_IsValid function:

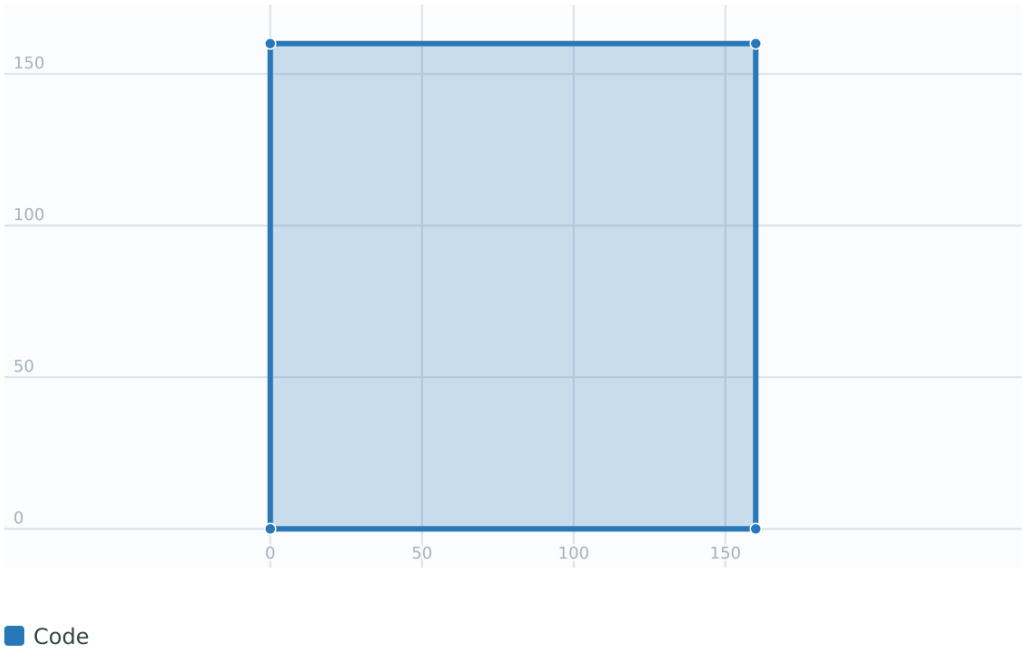
Code

```
SELECT ST_IsValid('POLYGON ((20 180,180 180,180 20,20 20,20 180))');
```

Output

```
 t
```

Figure



Information about the nature and location of an geometry invalidity are provided by the ST_IsValidDetail function:

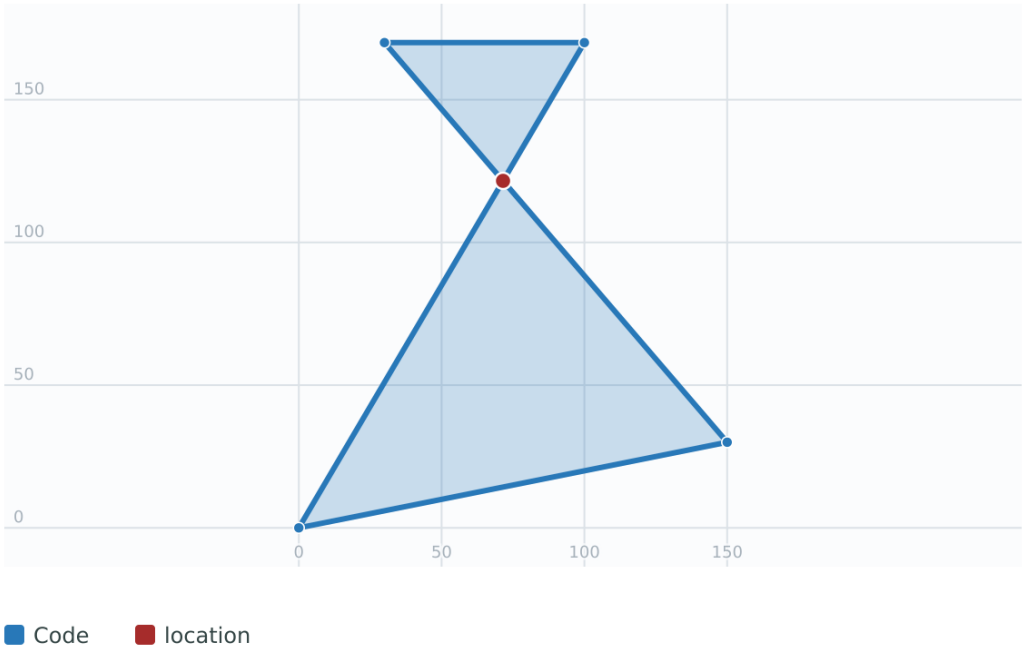
Code

```
SELECT valid, reason,location AS location
  FROM ST_IsValidDetail('POLYGON ((20 20,120 190,50 190,170 50,20 20))') AS t;
```

Output

```
 valid |      reason       |                  location
-------+-------------------+---------------------------------------------
 f     | Self-intersection | POINT(91.51162790697674 141.56976744186048)
```

Figure



In some situations it is desirable to correct invalid geometry automatically. Use the ST_MakeValid function to do this. (`ST_MakeValid` is a case of a spatial function that *does* allow invalid input!)

By default, PostGIS does not check for validity when loading geometry, because validity testing can take a lot of CPU time for complex geometries. If you do not trust your data sources, you can enforce a validity check on your tables by adding a check constraint:

Code

```
ALTER TABLE mytable
ADD CONSTRAINT geometry_valid_check
  CHECK (ST_IsValid(geom));
```

4.5 Spatial Reference Systems

A Spatial Reference System (SRS) (also called a Coordinate Reference System (CRS)) defines how geometry is referenced to locations on the Earth's surface. There are three types of SRS:

- A **geodetic** SRS uses angular coordinates (longitude and latitude) which map directly to the surface of the earth.
- A **projected** SRS uses a mathematical projection transformation to "flatten" the surface of the spheroidal earth onto a plane. It assigns location coordinates in a way that allows direct measurement of quantities such as distance, area, and angle. The coordinate system is Cartesian, which means it has a defined origin point and two perpendicular axes (usually oriented North and East). Each projected SRS uses a stated length unit (usually metres or feet). A projected SRS may be limited in its area of applicability to avoid distortion and fit within the defined coordinate bounds.
- A **local** SRS is a Cartesian coordinate system which is not referenced to the earth's surface. In PostGIS this is specified by a SRID value of 0.

There are many different spatial reference systems in use. Common SRSes are standardized in the European Petroleum Survey Group EPSG database. For convenience PostGIS (and many other spatial systems) refers to SRS definitions using an integer identifier called a SRID.

A geometry is associated with a Spatial Reference System by its SRID value, which is accessed by ST_SRID. The SRID for a geometry can be assigned using ST_SetSRID. Some geometry constructor functions allow supplying a SRID (such as ST_Point and ST_MakeEnvelope). The EWKT format supports SRIDs with the `SRID=n;` prefix.

Spatial functions processing pairs of geometries (such as overlay and relationship functions) require that the input geometries are in the same spatial reference system (have the same SRID). Geometry data can be transformed into a different spatial reference system using ST_Transform and ST_TransformPipeline. Geometry returned from functions has the same SRS as the input geometries.

4.5.1 SPATIAL_REF_SYS Table

The `SPATIAL_REF_SYS` table used by PostGIS is an OGC-compliant database table that defines the available spatial reference systems. It holds the numeric SRIDs and textual descriptions of the coordinate systems.

The `spatial_ref_sys` table definition is:

Code

```
CREATE TABLE spatial_ref_sys (
  srid       INTEGER NOT NULL PRIMARY KEY,
  auth_name  VARCHAR(256),
  auth_srid  INTEGER,
  srtext     VARCHAR(2048),
  proj4text  VARCHAR(2048)
)
```

The columns are:

`srid` An integer code that uniquely identifies the Spatial Reference System (SRS) within the database.

`auth_name` The name of the standard or standards body that is being cited for this reference system. For example, "EPSG" is a valid `auth_name`.

`auth_srid` The ID of the Spatial Reference System as defined by the Authority cited in the `auth_name`. In the case of EPSG, this is the EPSG code.

`srtext` The Well-Known Text representation of the Spatial Reference System. An example of a WKT SRS representation is:

Code

```
PROJCS["NAD83 / UTM Zone 10N",
  GEOGCS["NAD83",
    DATUM["North_American_Datum_1983",
      SPHEROID["GRS 1980", 6378137, 298.257222101]
    ],
    PRIMEM["Greenwich", 0],
    UNIT["degree", 0.0174532925199433]
  ],
  PROJECTION["Transverse_Mercator"],
  PARAMETER["latitude_of_origin", 0],
  PARAMETER["central_meridian", -123],
  PARAMETER["scale_factor", 0.9996],
  PARAMETER["false_easting", 500000],
  PARAMETER["false_northing", 0],
  UNIT["metre", 1]
]
```

For a discussion of SRS WKT, see the OGC standard Well-known text representation of coordinate reference systems.

`proj4text` PostGIS uses the PROJ library to provide coordinate transformation capabilities. The `proj4text` column contains the PROJ coordinate definition string for a particular SRID. For example:

Code

```
+proj=utm +zone=10 +ellps=clrk66 +datum=NAD27 +units=m
```

For more information see the PROJ web site. The `spatial_ref_sys.sql` file contains both `srtext` and `proj4text` definitions for all EPSG projections.

When retrieving spatial reference system definitions for use in transformations, PostGIS uses fhe following strategy:

- If `auth_name` and `auth_srid` are present (non-NULL) use the PROJ SRS based on those entries (if one exists).
- If `srtext` is present create a SRS using it, if possible.
- If `proj4text` is present create a SRS using it, if possible.

4.5.2 User-Defined Spatial Reference Systems

The PostGIS `spatial_ref_sys` table contains over 3000 of the most common spatial reference system definitions that are handled by the PROJ projection library. But there are many coordinate systems that it does not contain. You can add SRS definitions to the table if you have the required information about the spatial reference system. Or, you can define your own custom spatial reference system if you are familiar with PROJ constructs. Keep in mind that most spatial reference systems are regional and have no meaning when used outside of the bounds they were intended for.

A resource for finding spatial reference systems not defined in the core set is http://spatialreference.org/

Some commonly used spatial reference systems are: 4326 - WGS 84 Long Lat, 4269 - NAD 83 Long Lat, 3395 - WGS 84 World Mercator, 2163 - US National Atlas Equal Area, and the 60 WGS84 UTM zones. UTM zones are one of the most ideal for measurement, but only cover 6-degree regions. For WGS84 longitude/latitude data, the UTM zone is `least(floor((longitude + 180) / 6) + 1, 60)`; the EPSG code is `32600 + zone` in the northern hemisphere or `32700 + zone` in the southern hemisphere.

US states use State Plane spatial reference systems (meter or feet based) - usually one or 2 exists per state. Most of the meter-based ones are in the core set, but many of the feet-based ones or ESRI-created ones will need to be copied from spatialreference.org.

You can even define non-Earth-based coordinate systems, such as Mars 2000 This Mars coordinate system is non-planar (it's in degrees spheroidal), but you can use it with the `geography` type to obtain length and proximity measurements in meters instead of degrees.

Here is an example of loading a custom coordinate system using an unassigned SRID and the PROJ definition for a US-centric Lambert Conformal projection:

Code

```
INSERT INTO spatial_ref_sys (srid, proj4text)
VALUES ( 990000,
  '+proj=lcc  +lon_0=-95 +lat_0=25 +lat_1=25 +lat_2=25 +x_0=0 +y_0=0 +datum=WGS84 +units=m + ↩
    no_defs'
);
```

4.6 Spatial Tables

4.6.1 Creating a Spatial Table

You can create a table to store geometry data using the CREATE TABLE SQL statement with a column of type `geometry`. The following example creates a table with a geometry column storing 2D (XY) LineStrings in the BC-Albers coordinate system (SRID 3005):

Code

```
CREATE TABLE roads (
  id SERIAL PRIMARY KEY,
  name VARCHAR(64),
  geom geometry(LINESTRING, 3005)
);
```

The `geometry` type supports two optional **type modifiers**:

- the **spatial type modifier** restricts the kind of shapes and dimensions allowed in the column. The value can be any of the supported geometry subtypes (e.g. POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION, etc). The modifier supports coordinate dimensionality restrictions by adding suffixes: Z, M and ZM. For example, a modifier of 'LINESTRINGM' allows only linestrings with three dimensions, and treats the third dimension as a measure. Similarly, 'POINTZM' requires four dimensional (XYZM) data.
- the **SRID modifier** restricts the spatial reference system SRID to a particular number. If omitted, the SRID defaults to 0.

Examples of creating tables with geometry columns:

Create a table holding any kind of geometry with the default SRID:

Code

```
CREATE TABLE geoms(gid serial PRIMARY KEY, geom geometry);
```

Create a table with 2D POINT geometry with the default SRID:

Code

```
CREATE TABLE pts(gid serial PRIMARY KEY, geom geometry(POINT));
```

Create a table with 3D (XYZ) POINTs and an explicit SRID of 3005:

Code

```
CREATE TABLE pts(gid serial PRIMARY KEY, geom geometry(POINTZ, 3005));
```

Create a table with 4D (XYZM) LINESTRING geometry with the default SRID:

Code

```
CREATE TABLE lines(gid serial PRIMARY KEY, geom geometry(LINESTRINGZM));
```

Create a table with 2D POLYGON geometry with the SRID 4267 (NAD 1927 long lat):

Code

```
CREATE TABLE polys(gid serial PRIMARY KEY, geom geometry(POLYGON, 4267));
```

It is possible to have more than one geometry column in a table. This can be specified when the table is created, or a column can be added using the ALTER TABLE SQL statement. This example adds a column that can hold 3D LineStrings:

Code

```
ALTER TABLE roads ADD COLUMN geom2 geometry(LINESTRINGZ, 4326);
```

4.6.2 GEOMETRY_COLUMNS View

The OGC *Simple Features Specification for SQL* defines the `GEOMETRY_COLUMNS` metadata table to describe geometry table structure. In PostGIS `geometry_columns` is a view reading from database system catalog tables. This ensures that the spatial metadata information is always consistent with the currently defined tables and views. The view structure is:

Code

```
\d geometry_columns
```

Output

```
             View "public.geometry_columns"
      Column       |          Type          | Modifiers
-------------------+------------------------+-----------
 f_table_catalog   | character varying(256) |
 f_table_schema    | character varying(256) |
 f_table_name      | character varying(256) |
 f_geometry_column | character varying(256) |
 coord_dimension   | integer                |
 srid              | integer                |
 type              | character varying(30)  |
```

The columns are:

f_table_catalog, f_table_schema, f_table_name The fully qualified name of the feature table containing the geometry column. There is no PostgreSQL analogue of "catalog" so that column is left blank. For "schema" the PostgreSQL schema name is used (`public` is the default).

f_geometry_column The name of the geometry column in the feature table.

coord_dimension The coordinate dimension (2, 3 or 4) of the column. The coordinate dimension is at least 2, representing X and Y coordinates.

srid The ID of the spatial reference system used for the coordinate geometry in this table. It is a foreign key reference to the `spatial_ref_sys` table (see Section 4.5.1).

type The type of the spatial object. To restrict the spatial column to a single type, use one of: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION or corresponding XYM versions POINTM, LINESTRINGM, POLYGONM, MULTIPOINTM, MULTILINESTRINGM, MULTIPOLYGONM, GEOMETRYCOLLECTIONM. For heterogeneous (mixed-type) collections, you can use "GEOMETRY" as the type.

4.6.3 Manually Registering Geometry Columns

Two of the cases where you may need this are the case of SQL Views and bulk inserts. For bulk insert case, you can correct the registration in the geometry_columns table by constraining the column or doing an alter table. For views, you could expose using a CAST operation. Note, if your column is typmod based, the creation process would register it correctly, so no need to do anything. Also views that have no spatial function applied to the geometry will register the same as the underlying table geometry column.

For example, suppose you have a view created like this.

Code

```
CREATE VIEW public.vwmytablemercator AS
SELECT gid, ST_Transform(geom, 3395) As geom, f_name
FROM public.mytable;
```

To register the geometry column correctly, cast the transformed geometry.

Code

```
DROP VIEW public.vwmytablemercator;
CREATE VIEW  public.vwmytablemercator AS
    SELECT gid, ST_Transform(geom, 3395)::geometry(Geometry, 3395) As geom, f_name
    FROM public.mytable;
```

If the geometry type is known to be a 2D Polygon, specify it in the cast.

Code

```
DROP VIEW public.vwmytablemercator;
CREATE VIEW  public.vwmytablemercator AS
    SELECT gid, ST_Transform(geom, 3395)::geometry(Polygon, 3395) As geom, f_name
    FROM public.mytable;
```

For a derivative table created by a bulk insert, create the table first.

Code

```
SELECT poi.gid, poi.geom, citybounds.city_name
INTO myschema.my_special_pois
FROM poi INNER JOIN citybounds ON ST_Intersects(citybounds.geom, poi.geom);
```

Create a 2D spatial index on the new table.

Code

```
CREATE INDEX idx_myschema_myspecialpois_geom_gist
ON myschema.my_special_pois USING gist(geom);
```

If the points are 3D or measured, an n-D index may be more appropriate.

Code

```
CREATE INDEX my_special_pois_geom_gist_nd
ON my_special_pois USING gist(geom gist_geometry_ops_nd);
```

Manually register the new table's geometry column in `geometry_columns`. This also changes the underlying table structure to make the column typmod-based.

Code

```
SELECT Populate_Geometry_Columns('myschema.my_special_pois'::regclass);
```

To retain constraint-based definition behavior, such as for inherited tables where all children do not have the same type and SRID, set the optional `use_typmod` argument to false.

Code

```
SELECT Populate_Geometry_Columns('myschema.my_special_pois'::regclass, false);
```

Although the old-constraint based method is still supported, a constraint-based geometry column used directly in a view, will not register correctly in geometry_columns, as will a typmod one. In this example we define a column using typmod and another using constraints.

Code

```
CREATE TABLE pois_ny(gid SERIAL PRIMARY KEY, poi_name text, cat text, geom geometry(POINT,  ↩
    4326));
SELECT AddGeometryColumn('pois_ny', 'geom_2160', 2160, 'POINT', 2, false);
```

If we run **\d pois_ny** in psql, we observe they are defined differently -- one is typmod, one is constraint.

Code

```
\d pois_ny
```

Output

```
                                Table "public.pois_ny"
  Column   |         Type          |                       Modifiers

-----------+-----------------------+-------------------------------------------------------
 gid       | integer               | not null default nextval('pois_ny_gid_seq'::regclass)
 poi_name  | text                  |
 cat       | character varying(20) |
 geom      | geometry(Point,4326)  |
 geom_2160 | geometry              |
Indexes:
    "pois_ny_pkey" PRIMARY KEY, btree (gid)
Check constraints:
    "enforce_dims_geom_2160" CHECK (st_ndims(geom_2160) = 2)
    "enforce_geotype_geom_2160" CHECK (geometrytype(geom_2160) = 'POINT'::text
        OR geom_2160 IS NULL)
    "enforce_srid_geom_2160" CHECK (st_srid(geom_2160) = 2160)
```

In geometry_columns, they both register correctly

Code

```
SELECT f_table_name, f_geometry_column, srid, type
FROM geometry_columns
WHERE f_table_name = 'pois_ny';
```

Output

```
f_table_name | f_geometry_column | srid | type
-------------+-------------------+------+-------
pois_ny      | geom              | 4326 | POINT
pois_ny      | geom_2160         | 2160 | POINT
```

However -- if we were to create a view like this

Code

```
CREATE VIEW vw_pois_ny_parks AS
SELECT *
  FROM pois_ny
  WHERE cat='park';

SELECT f_table_name, f_geometry_column, srid, type
    FROM geometry_columns
    WHERE f_table_name = 'vw_pois_ny_parks';
```

The typmod based geom view column registers correctly, but the constraint based one does not.

Output

```
   f_table_name   | f_geometry_column | srid |   type
------------------+-------------------+------+----------
 vw_pois_ny_parks | geom              | 4326 | POINT
 vw_pois_ny_parks | geom_2160         |    0 | GEOMETRY
```

This may change in future versions of PostGIS, but for now to force the constraint-based view column to register correctly, you need to do this:

Code

```
DROP VIEW vw_pois_ny_parks;
CREATE VIEW vw_pois_ny_parks AS
SELECT gid, poi_name, cat,
  geom,
  geom_2160::geometry(POINT,2160) As geom_2160
  FROM pois_ny
  WHERE cat = 'park';
SELECT f_table_name, f_geometry_column, srid, type
    FROM geometry_columns
    WHERE f_table_name = 'vw_pois_ny_parks';
```

Output

```
   f_table_name   | f_geometry_column | srid | type
------------------+-------------------+------+-------
 vw_pois_ny_parks | geom              | 4326 | POINT
 vw_pois_ny_parks | geom_2160         | 2160 | POINT
```

4.7 Loading Spatial Data

Once you have created a spatial table, you are ready to upload spatial data to the database. There are two built-in ways to get spatial data into a PostGIS/PostgreSQL database: using formatted SQL statements or using the Shapefile loader.

4.7.1 Using SQL to Load Data

If spatial data can be converted to a text representation (as either WKT or WKB), then using SQL might be the easiest way to get data into PostGIS. Data can be bulk-loaded into PostGIS/PostgreSQL by loading a text file of SQL `INSERT` statements using the `psql` SQL utility.

A SQL load file (`roads.sql` for example) might look like this:

Code

```
BEGIN;
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (1, 'LINESTRING(191232 243118,191108 243242)', 'Jeff Rd');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (2, 'LINESTRING(189141 244158,189265 244817)', 'Geordie Rd');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (3, 'LINESTRING(192783 228138,192612 229814)', 'Paul St');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (4, 'LINESTRING(189412 252431,189631 259122)', 'Graeme Ave');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (5, 'LINESTRING(190131 224148,190871 228134)', 'Phil Tce');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (6, 'LINESTRING(198231 263418,198213 268322)', 'Dave Cres');
COMMIT;
```

The SQL file can be loaded into PostgreSQL using `psql`:

Code

```
psql -d [database] -f roads.sql
```

4.7.2 Using the Shapefile Loader

The `shp2pgsql` data loader converts Shapefiles into SQL suitable for insertion into a PostGIS/PostgreSQL database either in geometry or geography format. The loader has several operating modes selected by command line flags.

The input must be an unpacked shapefile path. The loader does not read `.zip` archives directly; unpack the `.shp`, `.shx`, and `.dbf` files before loading.

There is also a `shp2pgsql-gui` graphical interface with most of the options as the command-line loader. This may be easier to use for one-off non-scripted loading or if you are new to PostGIS. It can also be configured as a plugin to PgAdminIII.

(c|a|d|p) These are mutually exclusive options:

-c Creates a new table and populates it from the Shapefile. *This is the default mode.*

-a Appends data from the Shapefile into the database table. Note that to use this option to load multiple files, the files must have the same attributes and same data types.

-d Drops the database table before creating a new table with the data in the Shapefile.

-p Only produces the table creation SQL code, without adding any actual data. This can be used if you need to completely separate the table creation and data loading steps.

-? Display help screen.

-D Use the PostgreSQL "dump" format for the output data. This can be combined with -a, -c and -d. It is much faster to load than the default "insert" SQL format. Use this for very large data sets.

-f <fid_column>, --feature-id-column <fid_column> Specify the name of the feature id column. The default is `gid`. The value must be a simple PostgreSQL identifier and cannot use a PostgreSQL system column name.

-s [<FROM_SRID>:]<SRID> Creates and populates the geometry tables with the specified SRID. Optionally specifies that the input shapefile uses the given FROM_SRID, in which case the geometries will be reprojected to the target SRID.

-k Keep identifiers' case (column, schema and attributes). Note that attributes in Shapefile are all UPPERCASE.

-i Coerce all integers to standard 32-bit integers, do not create 64-bit bigints, even if the DBF header signature appears to warrant it.

-I Create a GiST index on the geometry column.

-u Create the target table as `UNLOGGED`. This can speed up loading transient staging data, but PostgreSQL does not preserve unlogged table contents after a crash or unclean shutdown.

-m Specify a file containing a set of mappings of (long) column names to 10 character DBF column names. The content of the file is one or more lines of two names separated by white space and no trailing or leading space. For example:

```
COLUMNNAME DBFFIELD1
AVERYLONGCOLUMNNAME DBFFIELD2
```

-S Generate simple geometries instead of MULTI geometries. Will only succeed if all the geometries are actually single (I.E. a MULTIPOLYGON with a single shell, or or a MULTIPOINT with a single vertex).

-t <dimensionality> Force the output geometry to have the specified dimensionality. Use the following strings to indicate the dimensionality: 2D, 3DZ, 3DM, 4D.

If the input has fewer dimensions that specified, the output will have those dimensions filled in with zeroes. If the input has more dimensions that specified, the unwanted dimensions will be stripped.

-w Output WKT format, instead of WKB. Note that this can introduce coordinate drifts due to loss of precision.

-e Execute each statement on its own, without using a transaction. This allows loading of the majority of good data when there are some bad geometries that generate errors. Note that this cannot be used with the -D flag as the "dump" format always uses a transaction.

-W <encoding> Specify encoding of the input data (dbf file). When used, all attributes of the dbf are converted from the specified encoding to UTF8. The resulting SQL output will contain a `SET CLIENT_ENCODING to UTF8` command, so that the backend will be able to reconvert from UTF8 to whatever encoding the database is configured to use internally.

-N <policy> NULL geometries handling policy (insert*,skip,abort)

-n -n Only import DBF file. If your data has no corresponding shapefile, it will automatically switch to this mode and load just the dbf. So setting this flag is only needed if you have a full shapefile set, and you only want the attribute data and no geometry.

-G Use geography type instead of geometry (requires lon/lat data) in WGS84 long lat (SRID=4326)

-T <tablespace> Specify the tablespace for the new table. Indexes will still use the default tablespace unless the -X parameter is also used. The PostgreSQL documentation has a good description on when to use custom tablespaces.

-X <tablespace> Specify the tablespace for the new table's indexes. This applies to the primary key index, and the GIST spatial index if -I is also used.

-Z When used, this flag will prevent the generation of `ANALYZE` statements. Without the -Z flag (default behavior), the `ANALYZE` statements will be generated.

An example session using the loader to create an input file and loading it might look like this:

Code

```
shp2pgsql -c -D -s 4269 -i -I shaperoads.shp myschema.roadstable > roads.sql
psql -d roadsdb -f roads.sql
```

A conversion and load can be done in one step using UNIX pipes:

Code

```
shp2pgsql shaperoads.shp myschema.roadstable | psql -d roadsdb
```

To load a directory of Shapefiles, run the loader once per file and derive the table name from the file name. On Windows `cmd.exe`, use a single percent sign interactively and double percent signs in a batch file:

Code

```
REM In a .bat file, use %%F. At an interactive cmd.exe prompt, use %F.
for %%F in ("C:\data\shapefiles\*.shp") do (
  shp2pgsql -d -D -s 4269 -I -W UTF-8 "%%~fF" "myschema.%%~nF" | psql -d roadsdb
)
```

If every Shapefile has the same attributes and data types and should be appended into one table, create the table from the first file and use `-a` for the remaining files.

4.8 Extracting Spatial Data

Spatial data can be extracted from the database using either SQL or the Shapefile dumper. The section on SQL presents some of the functions available to do comparisons and queries on spatial tables.

4.8.1 Using SQL to Extract Data

The most straightforward way of extracting spatial data out of the database is to use a SQL `SELECT` query to define the data set to be extracted and dump the resulting columns into a parsable text file:

Code

```
SELECT road_id,road_geom AS geom, road_name FROM roads;
```

Output

```
road_id | geom                                      | road_name
--------+-------------------------------------------+-----------
      1 | LINESTRING(191232 243118,191108 243242) | Jeff Rd
      2 | LINESTRING(189141 244158,189265 244817) | Geordie Rd
      3 | LINESTRING(192783 228138,192612 229814) | Paul St
      4 | LINESTRING(189412 252431,189631 259122) | Graeme Ave
      5 | LINESTRING(190131 224148,190871 228134) | Phil Tce
      6 | LINESTRING(198231 263418,198213 268322) | Dave Cres
      7 | LINESTRING(218421 284121,224123 241231) | Chris Way
(6 rows)
```

There will be times when some kind of restriction is necessary to cut down the number of records returned. In the case of attribute-based restrictions, use the same SQL syntax as used with a non-spatial table. In the case of spatial restrictions, the following functions are useful:

ST_Intersects This function tells whether two geometries share any space.

= This tests whether two geometries are geometrically identical. For example, if 'POLYGON((0 0,1 1,1 0,0 0))' is the same as 'POLYGON((0 0,1 1,1 0,0 0))' (it is).

Next, you can use these operators in queries. Note that when specifying geometries and boxes on the SQL command line, you must explicitly turn the string representations into geometries function. The 312 is a fictitious spatial reference system that matches our data. So, for example:

Code

```
SELECT road_id, road_name
FROM roads
WHERE roads_geom='SRID=312;LINESTRING(191232 243118,191108 243242)'::geometry;
```

The above query would return the single record from the "ROADS_GEOM" table in which the geometry was equal to that value.

To check whether some of the roads passes in the area defined by a polygon:

Code

```
SELECT road_id, road_name
FROM roads
WHERE ST_Intersects(roads_geom, 'SRID=312;POLYGON((...))');
```

The most common spatial query will probably be a "frame-based" query, used by client software, like data browsers and web mappers, to grab a "map frame" worth of data for display.

When using the "&&" operator, you can specify either a BOX3D as the comparison feature or a GEOMETRY. When you specify a GEOMETRY, however, its bounding box will be used for the comparison.

Using a "BOX3D" object for the frame, such a query looks like this:

Code

```
SELECT roads_geom AS geom
FROM roads
WHERE
  roads_geom && ST_MakeEnvelope(191232, 243117,191232, 243119,312);
```

Note the use of the SRID 312, to specify the projection of the envelope.

4.8.2 Using the Shapefile Dumper

The `pgsql2shp` table dumper connects to the database and converts a table (possibly defined by a query) into a shape file. The basic syntax is:

Code

```
pgsql2shp [<options>] <database> [<schema>.]<table>
```

Code

```
pgsql2shp [<options>] <database> <query>
```

The commandline options are:

-f <filename> Write the output to a particular filename.

-h <host> The database host to connect to.

-p <port> The port to connect to on the database host.

-P <password> The password to use when connecting to the database.

-u <user> The username to use when connecting to the database.

-g <geometry column> In the case of tables with multiple geometry columns, the geometry column to use when writing the shape file.

-b Use a binary cursor. This will make the operation faster, but will not work if any NON-geometry attribute in the table lacks a cast to text.

-r Raw mode. Do not drop the `gid` field, or escape column names.

-m filename Remap identifiers to ten character names. The content of the file is lines of two symbols separated by a single white space and no trailing or leading space: VERYLONGSYMBOL SHORTONE ANOTHERVERYLONGSYMBOL SHORTER etc.

4.9 Spatial Indexes

Spatial indexes make using a spatial database for large data sets possible. Without indexing, a search for features requires a sequential scan of every record in the database. Indexing speeds up searching by organizing the data into a structure which can be quickly traversed to find matching records.

The B-tree index method commonly used for attribute data is not very useful for spatial data, since it only supports storing and querying data in a single dimension. Data such as geometry (which has 2 or more dimensions) requires an index method that supports range query across all the data dimensions. One of the key advantages of PostgreSQL for spatial data handling is that it offers several kinds of index methods which work well for multi-dimensional data: GiST, BRIN and SP-GiST indexes.

- **GiST (Generalized Search Tree)** indexes break up data into "things to one side", "things which overlap", "things which are inside" and can be used on a wide range of data-types, including GIS data. PostGIS uses an R-Tree index implemented on top of GiST to index spatial data. GiST is the most commonly-used and versatile spatial index method, and offers very good query performance.
- **BRIN (Block Range Index)** indexes operate by summarizing the spatial extent of ranges of table records. Search is done via a scan of the ranges. BRIN is only appropriate for use for some kinds of data (spatially sorted, with infrequent or no update). But it provides much faster index create time, and much smaller index size.
- **SP-GiST (Space-Partitioned Generalized Search Tree)** is a generic index method that supports partitioned search trees such as quad-trees, k-d trees, and radix trees (tries).

Spatial indexes store only the bounding box of geometries. Spatial queries use the index as a **primary filter** to quickly determine a set of geometries potentially matching the query condition. Most spatial queries require a **secondary filter** that uses a spatial predicate function to test a more specific spatial condition. For more information on queying with spatial predicates see Section 5.2.

See also the PostGIS Workshop section on spatial indexes, and the PostgreSQL manual.

4.9.1 GiST Indexes

GiST stands for "Generalized Search Tree" and is a generic form of indexing for multi-dimensional data. PostGIS uses an R-Tree index implemented on top of GiST to index spatial data. GiST is the most commonly-used and versatile spatial index method, and offers very good query performance. Other implementations of GiST are used to speed up searches on all kinds of irregular data structures (integer arrays, spectral data, etc) which are not amenable to normal B-Tree indexing. For more information see the PostgreSQL manual.

Once a spatial data table exceeds a few thousand rows, you will want to build an index to speed up spatial searches of the data (unless all your searches are based on attributes, in which case you'll want to build a normal index on the attribute fields).

The syntax for building a GiST index on a "geometry" column is as follows:

Code

```
CREATE INDEX [indexname] ON [tablename] USING GIST ( [geometryfield]);
```

The above syntax will always build a 2D-index. To get the an n-dimensional index for the geometry type, you can create one using this syntax:

Code

```
CREATE INDEX [indexname] ON [tablename] USING GIST ([geometryfield] gist_geometry_ops_nd);
```

Building a spatial index is a computationally intensive exercise. It also blocks write access to your table for the time it creates, so on a production system you may want to do in in a slower CONCURRENTLY-aware way:

Code

```
CREATE INDEX CONCURRENTLY [indexname] ON [tablename] USING GIST ( [geometryfield]);
```

After building an index, it is sometimes helpful to force PostgreSQL to collect table statistics, which are used to optimize query plans:

Code

```
VACUUM ANALYZE [table_name] [(column_name)];
```

4.9.2 BRIN Indexes

BRIN stands for "Block Range Index". It is a general-purpose index method provided by PostgreSQL. BRIN is a *lossy* index method, meaning that a secondary check is required to confirm that a record matches a given search condition (which is the case for all provided spatial indexes). It provides much faster index creation and much smaller index size, with reasonable read performance. Its primary purpose is to support indexing very large tables on columns which have a correlation with their physical location within the table. In addition to spatial indexing, BRIN can speed up searches on various kinds of attribute data structures (integer, arrays etc). For more information see the PostgreSQL manual.

Once a spatial table exceeds a few thousand rows, you will want to build an index to speed up spatial searches of the data. GiST indexes are very performant as long as their size doesn't exceed the amount of RAM available for the database, and as long as you can afford the index storage size, and the cost of index update on write. Otherwise, for very large tables BRIN index can be considered as an alternative.

A BRIN index stores the bounding box enclosing all the geometries contained in the rows in a contiguous set of table blocks, called a *block range*. When executing a query using the index the block ranges are scanned to find the ones that intersect the query extent. This is efficient only if the data is physically ordered so that the bounding boxes for block ranges have minimal overlap (and ideally are mutually exclusive). The resulting index is very small in size, but is typically less performant for read than a GiST index over the same data.

Building a BRIN index is much less CPU-intensive than building a GiST index. It's common to find that a BRIN index is ten times faster to build than a GiST index over the same data. And because a BRIN index stores only one bounding box for each range of table blocks, it's common to use up to a thousand times less disk space than a GiST index.

You can choose the number of blocks to summarize in a range. If you decrease this number, the index will be bigger but will probably provide better performance.

For BRIN to be effective, the table data should be stored in a physical order which minimizes the amount of block extent overlap. It may be that the data is already sorted appropriately (for instance, if it is loaded from another dataset that is already sorted in spatial order). Otherwise, this can be accomplished by sorting the data by a one-dimensional spatial key. One way to do this is to create a new table sorted by the geometry values (which in recent PostGIS versions uses an efficient Hilbert curve ordering):

Code

```
CREATE TABLE table_sorted AS
SELECT * FROM table  ORDER BY geom;
```

Alternatively, data can be sorted in-place by using a GeoHash as a (temporary) index, and clustering on that index:

Code

```
CREATE INDEX idx_temp_geohash ON table
    USING btree (ST_GeoHash(ST_Transform(geom, 4326 ), 20));
CLUSTER table USING idx_temp_geohash;
```

The syntax for building a BRIN index on a `geometry` column is:

Code

```
CREATE INDEX [indexname] ON [tablename] USING BRIN ( [geome_col]);
```

The above syntax builds a 2D index. To build a 3D-dimensional index, use this syntax:

Code

```
CREATE INDEX [indexname] ON [tablename]
USING BRIN ([geome_col] brin_geometry_inclusion_ops_3d);
```

You can also get a 4D-dimensional index using the 4D operator class:

Code

```
CREATE INDEX [indexname] ON [tablename]
USING BRIN ([geome_col] brin_geometry_inclusion_ops_4d);
```

The above commands use the default number of blocks in a range, which is 128. To specify the number of blocks to summarise in a range, use this syntax

Code

```
CREATE INDEX [indexname] ON [tablename]
USING BRIN ( [geome_col] ) WITH (pages_per_range = [number]);
```

Keep in mind that a BRIN index only stores one index entry for a large number of rows. If your table stores geometries with a mixed number of dimensions, it's likely that the resulting index will have poor performance. You can avoid this performance penalty by choosing the operator class with the least number of dimensions of the stored geometries

The `geography` datatype is supported for BRIN indexing. The syntax for building a BRIN index on a geography column is:

Code

```
CREATE INDEX [indexname] ON [tablename] USING BRIN ( [geog_col]);
```

The above syntax builds a 2D-index for geospatial objects on the spheroid.

Currently, only "inclusion support" is provided, meaning that just the `&&`, `~` and `@` operators can be used for the 2D cases (for both `geometry` and `geography`), and just the `&&&` operator for 3D geometries. There is currently no support for kNN searches.

An important difference between BRIN and other index types is that the database does not maintain the index dynamically. Changes to spatial data in the table are simply appended to the end of the index. This will cause index search performance to degrade over time. The index can be updated by performing a `VACUUM`, or by using a special function `brin_summarize_new_values`. For this reason BRIN may be most appropriate for use with data that is read-only, or only rarely changing. For more information refer to the manual.

To summarize using BRIN for spatial data:

- Index build time is very fast, and index size is very small.
- Index query time is slower than GiST, but can still be very acceptable.
- Requires table data to be sorted in a spatial ordering.
- Requires manual index maintenance.
- Most appropriate for very large tables, with low or no overlap (e.g. points), which are static or change infrequently.
- More effective for queries which return relatively large numbers of data records.

4.9.3 SP-GiST Indexes

SP-GiST stands for "Space-Partitioned Generalized Search Tree" and is a generic form of indexing for multi-dimensional data types that supports partitioned search trees, such as quad-trees, k-d trees, and radix trees (tries). The common feature of these data structures is that they repeatedly divide the search space into partitions that need not be of equal size. In addition to spatial indexing, SP-GiST is used to speed up searches on many kinds of data, such as phone routing, ip routing, substring search, etc. For more information see the PostgreSQL manual.

As it is the case for GiST indexes, SP-GiST indexes are lossy, in the sense that they store the bounding box enclosing spatial objects. SP-GiST indexes can be considered as an alternative to GiST indexes.

Once a GIS data table exceeds a few thousand rows, an SP-GiST index may be used to speed up spatial searches of the data. The syntax for building an SP-GiST index on a "geometry" column is as follows:

Code

```
CREATE INDEX [indexname] ON [tablename] USING SPGIST ( [geometryfield]);
```

The above syntax will build a 2-dimensional index. A 3-dimensional index for the geometry type can be created using the 3D operator class:

Code

```
CREATE INDEX [indexname] ON [tablename] USING SPGIST ([geometryfield] spgist_geometry_ops_3d ↩
    );
```

Building a spatial index is a computationally intensive operation. It also blocks write access to your table for the time it creates, so on a production system you may want to do in in a slower CONCURRENTLY-aware way:

Code

```
CREATE INDEX CONCURRENTLY [indexname] ON [tablename] USING SPGIST ( [geometryfield]);
```

After building an index, it is sometimes helpful to force PostgreSQL to collect table statistics, which are used to optimize query plans:

Code

```
VACUUM ANALYZE [table_name] [(column_name)];
```

An SP-GiST index can accelerate queries involving the following operators:

- <<, &<, &>, >>, <<|, &<|, |&>, |>>, &&, @>, <@, and ~=, for 2-dimensional indexes,
- &/&, ~==, @>>, and <<@, for 3-dimensional indexes.

There is no support for kNN searches at the moment.

4.9.4 Tuning Index Usage

Ordinarily, indexes invisibly speed up data access: once an index is built, the PostgreSQL query planner automatically decides when to use it to improve query performance. But there are some situations where the planner does not choose to use existing indexes, so queries end up using slow sequential scans instead of a spatial index.

If you find your spatial indexes are not being used, there are a few things you can do:

- Examine the query plan and check your query actually computes the thing you need. An erroneous JOIN, either forgotten or to the wrong table, can unexpectedly retrieve table records multiple times. To get the query plan, execute with `EXPLAIN` in front of the query.
- Make sure statistics are gathered about the number and distributions of values in a table, to provide the query planner with better information to make decisions around index usage. **VACUUM ANALYZE** will compute both.

  You should regularly vacuum your databases anyways. Many PostgreSQL DBAs run **VACUUM** as an off-peak cron job on a regular basis.
- If vacuuming does not help, you can temporarily force the planner to use the index information by using the command **SET ENABLE_SEQSCAN TO OFF;**. This way you can check whether the planner is at all able to generate an index-accelerated query plan for your query. You should only use this command for debugging; generally speaking, the planner knows better than you do about when to use indexes. Once you have run your query, do not forget to run **SET ENABLE_SEQSCAN TO ON;** so that the planner will operate normally for other queries.
- If **SET ENABLE_SEQSCAN TO OFF;** helps your query to run faster, your Postgres is likely not tuned for your hardware. If you find the planner wrong about the cost of sequential versus index scans try reducing the value of `RANDOM_PAGE_COST` in `postgresql.conf`, or use **SET RANDOM_PAGE_COST TO 1.1;**. The default value for `RANDOM_PAGE_COST` is 4.0. Try setting it to 1.1 (for SSD) or 2.0 (for fast magnetic disks). Decreasing the value makes the planner more likely to use index scans.
- If **SET ENABLE_SEQSCAN TO OFF;** does not help your query, the query may be using a SQL construct that the Postgres planner is not yet able to optimize. It may be possible to rewrite the query in a way that the planner is able to handle. For example, a subquery with an inline SELECT may not produce an efficient plan, but could possibly be rewritten using a LATERAL JOIN.

For more information see the Postgres manual section on Query Planning.

Chapter 5

Spatial Queries

The *raison d'etre* of spatial databases is to perform queries inside the database which would ordinarily require desktop GIS functionality. Using PostGIS effectively requires knowing what spatial functions are available, how to use them in queries, and ensuring that appropriate indexes are in place to provide good performance.

5.1 Determining Spatial Relationships

Spatial relationships indicate how two geometries interact with one another. They are a fundamental capability for querying geometry.

5.1.1 Dimensionally Extended 9-Intersection Model

According to the OpenGIS Simple Features Implementation Specification for SQL, "the basic approach to comparing two geometries is to make pair-wise tests of the intersections between the Interiors, Boundaries and Exteriors of the two geometries and to classify the relationship between the two geometries based on the entries in the resulting 'intersection' matrix."

In the theory of point-set topology, the points in a geometry embedded in 2-dimensional space are categorized into three sets:

Boundary

The boundary of a geometry is the set of geometries of the next lower dimension. For `POINT`s, which have a dimension of 0, the boundary is the empty set. The boundary of a `LINESTRING` is the two endpoints. For `POLYGON`s, the boundary is the linework of the exterior and interior rings.

Interior

The interior of a geometry are those points of a geometry that are not in the boundary. For `POINT`s, the interior is the point itself. The interior of a `LINESTRING` is the set of points between the endpoints. For `POLYGON`s, the interior is the areal surface inside the polygon.

Exterior

The exterior of a geometry is the rest of the space in which the geometry is embedded; in other words, all points not in the interior or on the boundary of the geometry. It is a 2-dimensional non-closed surface.

The Dimensionally Extended 9-Intersection Model (DE-9IM) describes the spatial relationship between two geometries by specifying the dimensions of the 9 intersections between the above sets for each geometry. The intersection dimensions can be formally represented in a 3x3 **intersection matrix**.

For a geometry *g* the *Interior*, *Boundary*, and *Exterior* are denoted using the notation *I(g)*, *B(g)*, and *E(g)*. Also, *dim(s)* denotes the dimension of a set *s* with the domain of `{0,1,2,F}`:

- `0` => point
- `1` => line
- `2` => area
- `F` => empty set

Using this notation, the intersection matrix for two geometries *a* and *b* is:

	Interior	**Boundary**	**Exterior**
Interior	*dim(I(a) ∩ I(b))*	*dim(I(a) ∩ B(b))*	*dim(I(a) ∩ E(b))*
Boundary	*dim(B(a) ∩ I(b))*	*dim(B(a) ∩ B(b))*	*dim(B(a) ∩ E(b))*
Exterior	*dim(E(a) ∩ I(b))*	*dim(E(a) ∩ B(b))*	*dim(E(a) ∩ E(b))*

The following overlapping polygons provide a concrete example. ST_Relate computes the matrix `212101212`; it matches the overlap pattern `T*T***T**` used by ST_Overlaps.

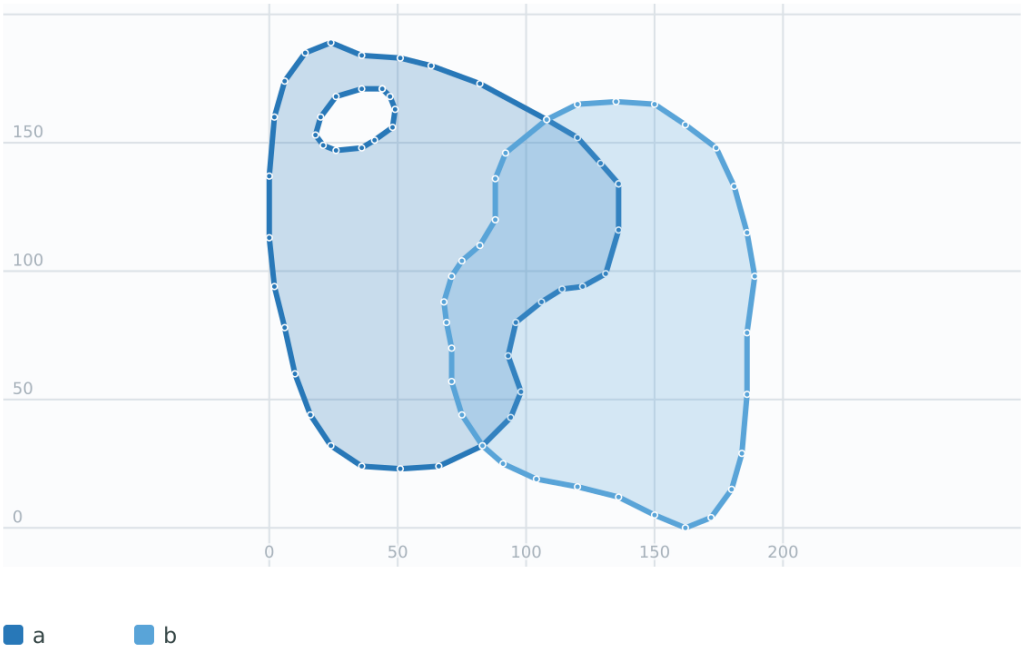
Code

```
WITH example AS (
  SELECT
    'POLYGON ((140 140,140 122,135 105,126 100,118 99,110 94,100 86,97 73,102 59,98 49,87  ↩
    38,70 30,55 29,40 30,28 38,20 50,14 66,10 84,6 100,4 119,4 143,6 166,10 180,18 191,28  ↩
    195,40 190,55 189,67 186,86 179,112 165,124 158,133 148,140 140),(48 177,40 177,30  ↩
    174,24 166,22 159,25 155,30 153,40 154,45 157,52 162,53 169,51 174,48 177))'::geometry  ↩
    AS a,
    'POLYGON ((75 63,79 50,87 38,95 31,108 25,124 22,140 18,154 11,166 6,176 10,184 21,188  ↩
    35,190 58,190 82,193 104,190 121,185 139,178 154,166 163,154 171,139 172,124 171,112  ↩
    165,96 152,92 142,92 126,86 116,79 110,75 104,72 94,73 86,75 76,75 63))'::geometry AS b
), relation AS (
  SELECT a, b, ST_Relate(a, b) AS matrix
  FROM example
)
SELECT matrix,
       ST_RelateMatch(matrix, 'T*T***T**') AS matches_overlap_pattern,
       ST_Overlaps(a, b) AS overlaps
FROM relation;
```

Output

```
  matrix   | matches_overlap_pattern | overlaps
-----------+-------------------------+----------
 212101212 | t                       | t
(1 row)
```

Figure



The executable query and generated overlay above are the visual source for the relation. The query below renders the nine cells of the same kind of matrix as a 3-by-3 set of examples. The final exterior/exterior cell is clipped to a finite viewport, since the true exterior/exterior intersection is unbounded.

Code

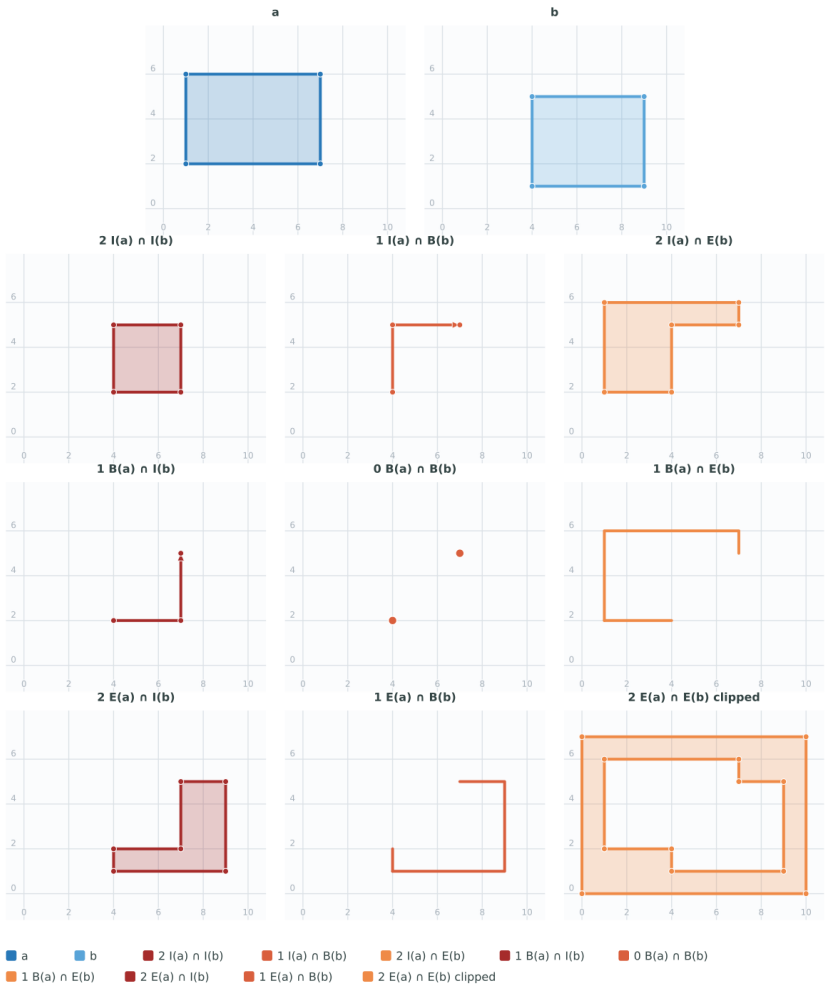
```
WITH example AS (
  SELECT
    ST_MakeEnvelope(0, 0, 6, 4) AS a,
    ST_MakeEnvelope(3, -1, 8, 3) AS b,
    ST_MakeEnvelope(-1, -2, 9, 5) AS viewport
), parts AS (
  SELECT
    a,
    b,
    ST_Boundary(a) AS ba,
    ST_Boundary(b) AS bb,
    viewport
  FROM example
)
SELECT
  ST_Relate(a, b) AS matrix,
  ST_AsText(ST_Normalize(a)) AS input_a,
  ST_AsText(ST_Normalize(b)) AS input_b,
  ST_AsText(ST_Normalize(ST_Intersection(a, b))) AS "2 I(a) ∩ I(b)",
  ST_AsText(ST_Normalize(ST_Intersection(a, bb))) AS "1 I(a) ∩ B(b)",
  ST_AsText(ST_Normalize(ST_Difference(a, b))) AS "2 I(a) ∩ E(b)",
  ST_AsText(ST_Normalize(ST_Intersection(ba, b))) AS "1 B(a) ∩ I(b)",
  ST_AsText(ST_Normalize(ST_Intersection(ba, bb))) AS "0 B(a) ∩ B(b)",
  ST_AsText(ST_Normalize(ST_Collect(
      'LINESTRING(0 0,0 4,6 4,6 3)'::geometry,
      'LINESTRING(0 0,3 0)'::geometry
  ))) AS "1 B(a) ∩ E(b)",
  ST_AsText(ST_Normalize(ST_Difference(b, a))) AS "2 E(a) ∩ I(b)",
  ST_AsText(ST_Normalize(ST_Collect(
      'LINESTRING(3 -1,8 -1,8 3,6 3)'::geometry,
      'LINESTRING(3 -1,3 0)'::geometry
```

```
  ))) AS "1 E(a) ∩ B(b)",
  ST_AsText(ST_Normalize(ST_Difference(viewport, ST_Union(a, b))))
    AS "2 E(a) ∩ E(b) clipped"
FROM parts;
```

Output

```
-[ RECORD 1 ]----------+----------------------------------------------------------------------------
matrix                 | 212101212
input_a                | POLYGON((0 0,0 4,6 4,6 0,0 0))
input_b                | POLYGON((3 -1,3 3,8 3,8 -1,3 -1))
2 I(a) ∩ I(b)          | POLYGON((3 0,3 3,6 3,6 0,3 0))
1 I(a) ∩ B(b)          | LINESTRING(3 0,3 3,6 3)
2 I(a) ∩ E(b)          | POLYGON((0 0,0 4,6 4,6 3,3 3,3 0,0 0))
1 B(a) ∩ I(b)          | LINESTRING(3 0,6 0,6 3)
0 B(a) ∩ B(b)          | MULTIPOINT((6 3),(3 0))
1 B(a) ∩ E(b)          | MULTILINESTRING((0 0,0 4,6 4,6 3),(0 0,3 0))
2 E(a) ∩ I(b)          | POLYGON((3 -1,3 0,6 0,6 3,8 3,8 -1,3 -1))
1 E(a) ∩ B(b)          | MULTILINESTRING((3 -1,8 -1,8 3,6 3),(3 -1,3 0))
2 E(a) ∩ E(b) clipped  | POLYGON((-1 -2,-1 5,9 5,9 -2,-1 -2),(0 0,3 0,3 -1,8 -1,8 3,6 3,6 4,0 4,0 0))
```

Figure



Reading from left to right and top to bottom, the intersection matrix is represented as the text string '**212101212**'.

For more information, refer to:

- OpenGIS Simple Features Implementation Specification for SQL (version 1.1, section 2.1.13.2)
- Wikipedia: Dimensionally Extended Nine-Intersection Model (DE-9IM)
- GeoTools: Point Set Theory and the DE-9IM Matrix

5.1.2 Named Spatial Relationships

To make it easy to determine common spatial relationships, the OGC SFS defines a set of *named spatial relationship predicates*. PostGIS provides these as the functions ST_Contains, ST_Crosses, ST_Disjoint, ST_Equals, ST_Intersects, ST_Overlaps, ST_Touches, ST_Within. It also defines the non-standard relationship predicates ST_Covers, ST_CoveredBy, and ST_ContainsProperly.

Spatial predicates are usually used as conditions in SQL `WHERE` or `JOIN` clauses. The named spatial predicates automatically use a spatial index if one is available, so there is no need to use the bounding box operator `&&` as well. For example:

Code

```
SELECT city.name, state.name, city.geom
FROM city JOIN state ON ST_Intersects(city.geom, state.geom);
```

For more details and illustrations, see the PostGIS Workshop.

5.1.3 General Spatial Relationships

In some cases the named spatial relationships are insufficient to provide a desired spatial filter condition. These requirements can be expressed by computing the full DE-9IM intersection matrix with ST_Relate.

To test a particular spatial relationship, an **intersection matrix pattern** is used. This is the matrix representation augmented with the additional symbols `{T,*}`:

- `T` => intersection dimension is non-empty; i.e. is in `{0,1,2}`
- `*` => don't care

Using intersection matrix patterns, specific spatial relationships can be evaluated succinctly. The ST_Relate and ST_RelateMatch functions can both test these patterns.

These road segments share a line segment. ST_Crosses does not find this relationship for linear features, because it only returns `true` when their interiors intersect at a point. The pattern `1*1***1**` tests for a one-dimensional intersection instead.

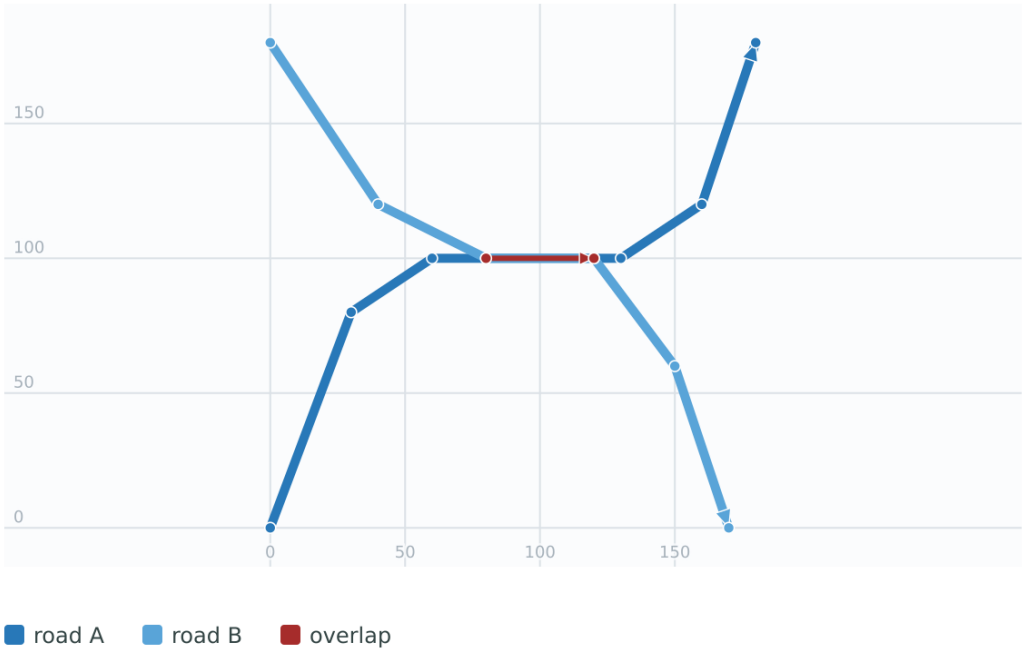
Code

```
WITH roads AS (
  SELECT
    'LINESTRING(10 10,40 90,70 110,140 110,170 130,190 190)'::geometry AS "road A",
    'LINESTRING(10 190,50 130,90 110,130 110,160 70,180 10)'::geometry AS "road B"
), relation AS (
  SELECT "road A", "road B", ST_Relate("road A", "road B") AS matrix
  FROM roads
)
SELECT
  ST_Intersection("road A", "road B") AS overlap,
  matrix,
  ST_Crosses("road A", "road B") AS "ST_Crosses",
  ST_RelateMatch(matrix, '1*1***1**') AS "matches 1*1***1**"
FROM relation;
```

Output

```
          overlap           |  matrix   | ST_Crosses | matches 1*1***1**
----------------------------+-----------+------------+-------------------
 LINESTRING(90 110,130 110) | 1F1FF0102 | f          | t
(1 row)
```

Figure



The next example finds a wharf which runs from the lake interior onto its shoreline, with one endpoint on the shoreline. The other wharves provide nearby counterexamples in the figure. The pattern `102101FF2` expresses the complete required relationship in one test.

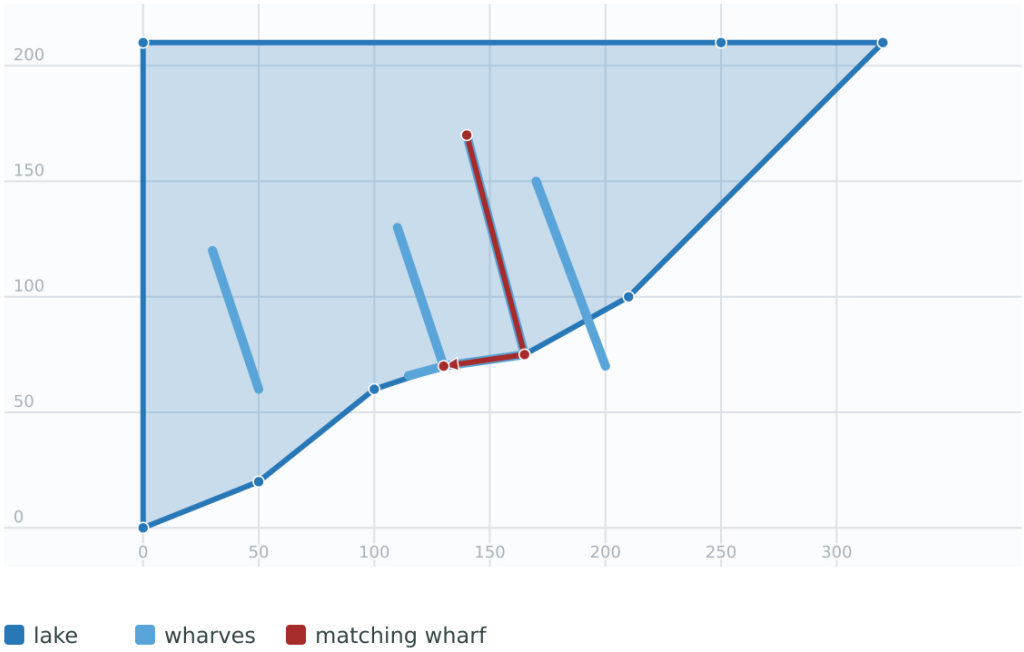
Code

```
WITH data AS (
  SELECT
    'POLYGON((-20 10,30 30,80 70,110 80,145 85,190 110,300 220,230 220,-20 220,-20 10))':: ↩
   geometry AS lake,
    'MULTILINESTRING((120 180,145 85,110 80),(10 130,30 70),(90 140,110 80,94.81118881118876 ↩
     75.83496503496498),(150 160,180 80))'::geometry AS wharves
), wharf AS (
  SELECT lake, (item).path[1] AS id, (item).geom
  FROM data CROSS JOIN LATERAL ST_Dump(wharves) AS item
), relation AS (
  SELECT id, geom, ST_Relate(lake, geom) AS matrix
  FROM wharf
)
SELECT
  id AS wharf,
  geom AS "matching wharf",
  matrix
FROM relation
WHERE ST_RelateMatch(matrix, '102101FF2')
ORDER BY id;
```

Output

```
 wharf |           matching wharf           |  matrix
-------+------------------------------------+-----------
     1 | LINESTRING(120 180,145 85,110 80) | 102101FF2
(1 row)
```

Figure



5.2 Using Spatial Indexes

When constructing queries using spatial conditions, for best performance it is important to ensure that a spatial index is used, if one exists (see Section 4.9). To do this, a spatial operator or index-aware function must be used in a `WHERE` or `ON` clause of the query.

Spatial operators include the bounding box operators (of which the most commonly used is &&; see Section 7.10.1 for the full list) and the distance operators used in nearest-neighbor queries (the most common being <->; see Section 7.10.2 for the full list.)

Index-aware functions automatically add a bounding box operator to the spatial condition. Index-aware functions include the named spatial relationship predicates ST_Contains, ST_ContainsProperly, ST_CoveredBy, ST_Covers, ST_Crosses, ST_Intersects, ST_Overlaps, ST_Touches, ST_Within, ST_Within, and ST_3DIntersects, and the distance predicates ST_DWithin, ST_DFullyWithin, ST_3DDFullyWithin, and ST_3DDWithin .)

Functions such as ST_Distance do *not* use indexes to optimize their operation. For example, the following query would be quite slow on a large table:

Code

```
SELECT geom
FROM geom_table
WHERE ST_Distance(geom, 'SRID=312;POINT(100000 200000)') < 100
```

This query selects all the geometries in `geom_table` which are within 100 units of the point (100000, 200000). It will be slow because it is calculating the distance between each point in the table and the specified point, ie. one `ST_Distance()` calculation is computed for **every** row in the table.

The number of rows processed can be reduced substantially by using the index-aware function ST_DWithin:

Code

```
SELECT geom
FROM geom_table
WHERE ST_DWithin(geom, 'SRID=312;POINT(100000 200000)', 100)
```

This query selects the same geometries, but it does it in a more efficient way. This is enabled by `ST_DWithin()` using the `&&` operator internally on an expanded bounding box of the query geometry. If there is a spatial index on `geom`, the query planner will recognize that it can use the index to reduce the number of rows scanned before calculating the distance. The spatial index allows retrieving only records with geometries whose bounding boxes overlap the expanded extent and hence which *might* be within the required distance. The actual distance is then computed to confirm whether to include the record in the result set.

For more information and examples see the PostGIS Workshop.

5.3 Examples of Spatial SQL

The examples in this section make use of a table of linear roads, and a table of polygonal municipality boundaries. The definition of the `bc_roads` table is:

Output

```
Column    | Type              | Description
----------+-------------------+-------------------
gid       | integer           | Unique ID
name      | character varying | Road Name
geom      | geometry          | Location Geometry (Linestring)
```

The definition of the `bc_municipality` table is:

Output

```
Column   | Type              | Description
---------+-------------------+-------------------
gid      | integer           | Unique ID
code     | integer           | Unique ID
name     | character varying | City / Town Name
geom     | geometry          | Location Geometry (Polygon)
```

1. *What is the total length of all roads, expressed in kilometers?*

   You can answer this question with a very simple piece of SQL:

   Code

   ```
   SELECT sum(ST_Length(geom))/1000 AS km_roads FROM bc_roads;
   ```

   Output

   ```
   km_roads
   ------------------
   70842.1243039643
   ```

2. *How large is the city of Prince George, in hectares?*

   This query combines an attribute condition (on the municipality name) with a spatial calculation (of the polygon area):

   Code

   ```
   SELECT
     ST_Area(geom)/10000 AS hectares
   FROM bc_municipality
   WHERE name = 'PRINCE GEORGE';
   ```

Output

```
hectares
------------------
32657.9103824927
```

3. *What is the largest municipality in the province, by area?*

   This query uses a spatial measurement as an ordering value. There are several ways of approaching this problem, but the most efficient is below:

   Code

   ```
   SELECT
     name,
     ST_Area(geom)/10000 AS hectares
   FROM bc_municipality
   ORDER BY hectares DESC
   LIMIT 1;
   ```

   Output

   ```
   name           | hectares
   ---------------+-----------------
   TUMBLER RIDGE  | 155020.02556131
   ```

   Note that in order to answer this query we have to calculate the area of every polygon. If we were doing this a lot it would make sense to add an area column to the table that could be indexed for performance. By ordering the results in a descending direction, and them using the PostgreSQL "LIMIT" command we can easily select just the largest value without using an aggregate function like MAX().

4. *What is the length of roads fully contained within each municipality?*

   This is an example of a "spatial join", which brings together data from two tables (with a join) using a spatial interaction ("contained") as the join condition (rather than the usual relational approach of joining on a common key):

   Code

   ```
   SELECT
     m.name,
     sum(ST_Length(r.geom))/1000 as roads_km
   FROM bc_roads AS r
   JOIN bc_municipality AS m
     ON ST_Contains(m.geom, r.geom)
   GROUP BY m.name
   ORDER BY roads_km;
   ```

   Output

   ```
   name                         | roads_km
   -----------------------------+----------
   SURREY                       | 1539.476
   VANCOUVER                    | 1450.331
   LANGLEY DISTRICT             |  833.793
   BURNABY                      |  773.769
   PRINCE GEORGE                |  694.376
   ...
   ```

   This query takes a while, because every road in the table is summarized into the final result (about 250K roads for the example table). For smaller datasets (several thousand records on several hundred) the response can be very fast.

5. *Create a new table with all the roads within the city of Prince George.*

   This is an example of an "overlay", which takes in two tables and outputs a new table that consists of spatially clipped or cut resultants. Unlike the "spatial join" demonstrated above, this query creates new geometries. An overlay is like a turbo-charged spatial join, and is useful for more exact analysis work:

Code

```
CREATE TABLE pg_roads as
SELECT
  ST_Intersection(r.geom, m.geom) AS intersection_geom,
  ST_Length(r.geom) AS rd_orig_length,
  r.*
FROM bc_roads AS r
JOIN bc_municipality AS m
  ON ST_Intersects(r.geom, m.geom)
WHERE
  m.name = 'PRINCE GEORGE';
```

6. *What is the length in kilometers of "Douglas St" in Victoria?*

Code

```
SELECT
  sum(ST_Length(r.geom))/1000 AS kilometers
FROM bc_roads r
JOIN bc_municipality m
  ON ST_Intersects(m.geom, r.geom
WHERE
  r.name = 'Douglas St'
  AND m.name = 'VICTORIA';
```

Output

```
kilometers
------------------
4.89151904172838
```

7. *What is the largest municipality polygon that has a hole?*

Code

```
SELECT gid, name, ST_Area(geom) AS area
FROM bc_municipality
WHERE ST_NRings(geom) > 1
ORDER BY area DESC LIMIT 1;
```

Output

```
gid  | name         | area
-----+--------------+------------------
12   | SPALLUMCHEEN | 257374619.430216
```

Chapter 6

Performance Tips

6.1 Small tables of large geometries

6.1.1 Problem description

Current PostgreSQL versions suffer from a query optimizer weakness regarding TOAST tables. TOAST tables are a kind of "extension room" used to store large (in the sense of data size) values that do not fit into normal data pages (like long texts, images or complex geometries with lots of vertices), see the PostgreSQL Documentation for TOAST for more information).

The problem appears if you happen to have a table with rather large geometries, but not too many rows of them (like a table containing the boundaries of all European countries in high resolution). Then the table itself is small, but it uses lots of TOAST space. In our example case, the table itself had about 80 rows and used only 3 data pages, but the TOAST table used 8225 pages.

Now issue a query where you use the geometry operator && to search for a bounding box that matches only very few of those rows. Now the query optimizer sees that the table has only 3 pages and 80 rows. It estimates that a sequential scan on such a small table is much faster than using an index. And so it decides to ignore the GIST index. Usually, this estimation is correct. But in our case, the && operator has to fetch every geometry from disk to compare the bounding boxes, thus reading all TOAST pages, too.

To see whether your suffer from this issue, use the "EXPLAIN ANALYZE" postgresql command. For more information and the technical details, you can read the thread on the PostgreSQL performance mailing list: http://archives.postgresql.org/pgsql-performance/2005-02/msg00030.php

and newer thread on PostGIS https://lists.osgeo.org/pipermail/postgis-devel/2017-June/026209.html

6.1.2 Workarounds

The PostgreSQL people are trying to solve this issue by making the query estimation TOAST-aware. For now, here are two workarounds:

The first workaround is to force the query planner to use the index. Send "SET enable_seqscan TO off;" to the server before issuing the query. This basically forces the query planner to avoid sequential scans whenever possible. So it uses the GIST index as usual. But this flag has to be set on every connection, and it causes the query planner to make misestimations in other cases, so you should "SET enable_seqscan TO on;" after the query.

The second workaround is to make the sequential scan as fast as the query planner thinks. This can be achieved by creating an additional column that "caches" the bbox, and matching against this. In our example, the commands are like:

Code

```
SELECT AddGeometryColumn('myschema', 'mytable', 'bbox', '4326', 'GEOMETRY', '2');
UPDATE mytable SET bbox = ST_Envelope(ST_Force2D(geom));
```

Now change your query to use the && operator against bbox instead of geom_column, like:

Code

```
SELECT geom_column
FROM mytable
WHERE bbox && ST_SetSRID('BOX3D(0 0,1 1)'::box3d,4326);
```

Of course, if you change or add rows to mytable, you have to keep the bbox "in sync". The most transparent way to do this would be triggers, but you also can modify your application to keep the bbox column current or run the UPDATE query above after every modification.

6.2 CLUSTERing on geometry indices

The PostgreSQL CLUSTER command physically reorders table rows according to an index. For a mostly read-only table whose queries frequently use the same spatial index, this can reduce I/O and improve cache locality.

PostgreSQL supports clustering on GiST indices, including indices on nullable geometry columns. To cluster a table using a geometry index:

Code

```
CLUSTER my_table USING my_geom_index;
```

Output

```
CLUSTER
```

6.3 Avoiding dimension conversion

Sometimes, you happen to have 3D or 4D data in your table, but always access it using OpenGIS compliant ST_AsText() or ST_AsBinary() functions that only output 2D geometries. They do this by internally calling the ST_Force2D() function, which introduces a significant overhead for large geometries. To avoid this overhead, it may be feasible to pre-drop those additional dimensions once and forever:

Code

```
UPDATE mytable SET geom = ST_Force2D(geom);
VACUUM FULL ANALYZE mytable;
```

Note that if you added your geometry column using AddGeometryColumn() there'll be a constraint on geometry dimension. To bypass it you will need to drop the constraint. Remember to update the entry in the geometry_columns table and recreate the constraint afterwards.

In case of large tables, it may be wise to divide this UPDATE into smaller portions by constraining the UPDATE to a part of the table via a WHERE clause and your primary key or another feasible criteria, and running a simple "VACUUM;" between your UPDATEs. This drastically reduces the need for temporary disk space. Additionally, if you have mixed dimension geometries, restricting the UPDATE by "WHERE dimension(geom)>2" skips re-writing of geometries that already are in 2D.

Chapter 7

PostGIS Reference

The functions given below are the ones which a user of PostGIS is likely to need. There are other functions which are required support functions to the PostGIS objects which are not of use to a general user.



Note

PostGIS has begun a transition from the existing naming convention to an SQL-MM-centric convention. As a result, most of the functions that you know and love have been renamed using the standard spatial type (ST) prefix. Previous functions are still available, though are not listed in this document where updated functions are equivalent. The non ST_ functions not listed in this documentation are deprecated and will be removed in a future release so STOP USING THEM.

7.1 PostGIS Geometry/Geography/Box Data Types

7.1.1 box2d

box2d — The type representing a 2-dimensional bounding box.

Description

`box2d` is a spatial data type used to represent the two-dimensional bounding box enclosing a geometry or collection of geometries. For example, the ST_Extent aggregate function returns a `box2d` object.

The representation contains the values `xmin, ymin, xmax, ymax`. These are the minimum and maximum values of the X and Y extents.

`box2d` objects have a text representation which looks like `BOX(1 2,5 6)`.

Casting Behavior

This table lists the automatic and explicit casts allowed for this data type:

Cast To	Behavior
box3d	automatic
geometry	automatic

See Also

Section 13.7

7.1.2 box3d

box3d — The type representing a 3-dimensional bounding box.

Description

`box3d` is a PostGIS spatial data type used to represent the three-dimensional bounding box enclosing a geometry or collection of geometries. For example, the ST_3DExtent aggregate function returns a `box3d` object.

The representation contains the values `xmin, ymin, zmin, xmax, ymax, zmax`. These are the minimum and maximum values of the X, Y and Z extents.

`box3d` objects have a text representation which looks like `BOX3D(1 2 3,5 6 5)`.

Casting Behavior

This table lists the automatic and explicit casts allowed for this data type:

Cast To	Behavior
box	automatic
box2d	automatic
geometry	automatic

See Also

Section 13.7

7.1.3 geometry

geometry — The type representing spatial features with planar coordinate systems.

Description

`geometry` is a fundamental PostGIS spatial data type used to represent a feature in planar (Euclidean) coordinate systems.

All spatial operations on geometry use the units of the Spatial Reference System the geometry is in.

Casting Behavior

This table lists the automatic and explicit casts allowed for this data type:

Cast To	Behavior
box	automatic
box2d	automatic
box3d	automatic
bytea	automatic
geography	automatic
text	automatic

See Also

Section 4.1, Section 13.3

7.1.4 geometry_dump

geometry_dump — A composite type used to describe the parts of complex geometry.

Description

`geometry_dump` is a composite data type containing the fields:

- `geom` - a geometry representing a component of the dumped geometry. The geometry type depends on the originating function.
- `path[]` - an integer array that defines the navigation path within the dumped geometry to the `geom` component. The path array is 1-based (i.e. `path[1]` is the first element.)

It is used by the `ST_Dump*` family of functions as an output type to explode a complex geometry into its constituent parts.

See Also

Section 13.6

7.1.5 geography

geography — The type representing spatial features with geodetic (ellipsoidal) coordinate systems.

Description

`geography` is a spatial data type used to represent a feature in geodetic coordinate systems. Geodetic coordinate systems model the earth using an ellipsoid.

Spatial operations on the geography type provide more accurate results by taking the ellipsoidal model into account.

Casting Behavior

This table lists the automatic and explicit casts allowed for this data type:

Cast To	Behavior
geometry	explicit

See Also

Section 4.3, Section 13.4

7.2 Table Management Functions

7.2.1 AddGeometryColumn

AddGeometryColumn — Adds a geometry column to an existing table.

Synopsis

text **AddGeometryColumn**(varchar table_name, varchar column_name, integer srid, varchar type, integer dimension, boolean use_typmod=true);
text **AddGeometryColumn**(varchar schema_name, varchar table_name, varchar column_name, integer srid, varchar type, integer dimension, boolean use_typmod=true);
text **AddGeometryColumn**(varchar catalog_name, varchar schema_name, varchar table_name, varchar column_name, integer srid, varchar type, integer dimension, boolean use_typmod=true);

Description

Adds a geometry column to an existing table of attributes. The `schema_name` is the name of the table schema. The `srid` must be an integer value reference to an entry in the SPATIAL_REF_SYS table. The `type` must be a string corresponding to the geometry type, eg, 'POLYGON' or 'MULTILINESTRING' . An error is thrown if the schemaname doesn't exist (or not visible in the current search_path) or the specified SRID, geometry type, or dimension is invalid.

Changed: 2.0.0 This function no longer updates geometry_columns since geometry_columns is a view that reads from system catalogs. It by default also does not create constraints, but instead uses the built in type modifier behavior of PostgreSQL. So for example building a wgs84 POINT column with this function is now equivalent to: `ALTER TABLE some_table ADD COLUMN geom geometry(Point,4326);`

Changed: 2.0.0 If you require the old behavior of constraints use the default `use_typmod`, but set it to false.

Changed: 2.0.0 Views can no longer be manually registered in geometry_columns, however views built against geometry typmod tables geometries and used without wrapper functions will register themselves correctly because they inherit the typmod behavior of their parent table column. Views that use geometry functions that output other geometries will need to be cast to typmod geometries for these view geometry columns to be registered correctly in geometry_columns. Refer to Section 4.6.3.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Enhanced: 2.0.0 use_typmod argument introduced. Defaults to creating typmod geometry column instead of constraint-based.

Examples

This example creates a schema to hold data.

Code

```
CREATE SCHEMA my_schema;
```

Create a simple PostgreSQL table in the schema.

Code

```
CREATE TABLE my_schema.my_spatial_table (id serial);
```

Describing the table shows a single `id` column.

Code

```
\d my_schema.my_spatial_table
```

Output

```
                                                    Table "my_schema.my_spatial_table"
 Column |  Type   |                                  Modifiers
--------+---------+-----------------------------------------------------------------------------
 id     | integer | not null default nextval('my_schema.my_spatial_table_id_seq'::regclass)
```

Add a spatial column to the table.

Code

```
SELECT AddGeometryColumn('my_schema', 'my_spatial_table', 'geom', 4326, 'POINT', 2);
```

Add a point column using the old constraint-based behavior.

Code

```
SELECT AddGeometryColumn('my_schema', 'my_spatial_table', 'geom_c', 4326, 'POINT', 2, false) ↩
    ;
```

Add a CurvePolygon column using the old constraint-based behavior.

Code

```
SELECT AddGeometryColumn('my_schema', 'my_spatial_table', 'geomcp_c', 4326, 'CURVEPOLYGON',  ↩
    2, false);
```

Output

```
my_schema.my_spatial_table.geomcp_c SRID:4326 TYPE:CURVEPOLYGON DIMS:2
```

Describe the table again to see the new geometry columns.

Code

```
\d my_schema.my_spatial_table
```

Output

```
Table "my_schema.my_spatial_table"

Columns:
  id       integer              not null default nextval('my_schema.my_spatial_table_id_seq'::regclass)
  geom     geometry(Point,4326)
  geom_c   geometry
  geomcp_c geometry

Check constraints:
    "enforce_dims_geom_c" CHECK (st_ndims(geom_c) = 2)
    "enforce_dims_geomcp_c" CHECK (st_ndims(geomcp_c) = 2)
    "enforce_geotype_geom_c" CHECK (geometrytype(geom_c) = 'POINT'::text OR geom_c IS NULL)
    "enforce_geotype_geomcp_c" CHECK (geometrytype(geomcp_c) = 'CURVEPOLYGON'::text OR geomcp_c IS NULL)
    "enforce_srid_geom_c" CHECK (st_srid(geom_c) = 4326)
    "enforce_srid_geomcp_c" CHECK (st_srid(geomcp_c) = 4326)
```

The `geometry_columns` view also registers the new columns.

Code

```
SELECT f_geometry_column As col_name, type, srid, coord_dimension As ndims
FROM geometry_columns
WHERE f_table_name = 'my_spatial_table' AND f_table_schema = 'my_schema';
```

Output

```
 col_name |     type     | srid | ndims
----------+--------------+------+-------
 geom     | Point        | 4326 |     2
 geom_c   | Point        | 4326 |     2
 geomcp_c | CurvePolygon | 4326 |     2
```

See Also

DropGeometryColumn, DropGeometryTable, Section 4.6.2, Section 4.6.3

7.2.2 DropGeometryColumn

DropGeometryColumn — Removes a geometry column from a spatial table.

Synopsis

text **DropGeometryColumn**(varchar table_name, varchar column_name);
text **DropGeometryColumn**(varchar schema_name, varchar table_name, varchar column_name);
text **DropGeometryColumn**(varchar catalog_name, varchar schema_name, varchar table_name, varchar column_name);

Description

Removes a geometry column from a spatial table. Note that schema_name will need to match the f_table_schema field shown by the `geometry_columns` view.



Warning
`DropGeometryColumn` is retained for backward compatibility and is not recommended for new applications. Use the standard **ALTER TABLE ... DROP COLUMN** command instead. The `geometry_columns` view updates automatically from the system catalogs.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.

Changed: 2.0.0. Retained for backward compatibility after `geometry_columns` became a view of the system catalogs.

Examples

Use standard SQL to drop the geometry column. The `geometry_columns` view updates automatically.

Code

```
ALTER TABLE my_schema.my_spatial_table DROP COLUMN geom;
```

The legacy compatibility function performs the equivalent operation.

Code

```
SELECT DropGeometryColumn ('my_schema', 'my_spatial_table', 'geom');
```

Output

```
my_schema.my_spatial_table.geom effectively removed.
```

See Also

AddGeometryColumn, DropGeometryTable, Section 4.6.2

7.2.3 DropGeometryTable

DropGeometryTable — Drops a table and all its references in geometry_columns.

Synopsis

boolean **DropGeometryTable**(varchar table_name);
boolean **DropGeometryTable**(varchar schema_name, varchar table_name);
boolean **DropGeometryTable**(varchar catalog_name, varchar schema_name, varchar table_name);

Description

Drops a table and all its references in geometry_columns. Note: uses current_schema() on schema-aware pgsql installations if schema is not provided.



Warning
`DropGeometryTable` is retained for backward compatibility and is not recommended for new applications. Use the standard **DROP TABLE** command instead. The `geometry_columns` view updates automatically from the system catalogs.

Changed: 2.0.0. Retained for backward compatibility after `geometry_columns` became a view of the system catalogs.

Examples

Use standard SQL to drop a table containing geometry columns. The `geometry_columns` view updates automatically.

Code

```
DROP TABLE my_schema.my_spatial_table;
```

The legacy compatibility function performs the equivalent operation.

Code

```
SELECT DropGeometryTable ('my_schema', 'my_spatial_table');
```

Output

```
my_schema.my_spatial_table dropped.
```

See Also

AddGeometryColumn, DropGeometryColumn, Section 4.6.2

7.2.4 Find_SRID

Find_SRID — Returns the SRID defined for a geometry column.

Synopsis

integer **Find_SRID**(varchar a_schema_name, varchar a_table_name, varchar a_geomfield_name);

Description

Returns the integer SRID of the specified geometry column by searching through the GEOMETRY_COLUMNS table. If the geometry column has not been properly added (e.g. with the AddGeometryColumn function), this function will not work.

Examples

Code

```
SELECT Find_SRID('public', 'tiger_us_state_2007', 'geom_4269');
```

Output

```
4269
```

See Also

ST_SRID

7.2.5 Populate_Geometry_Columns

Populate_Geometry_Columns — Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.

Synopsis

text **Populate_Geometry_Columns**(boolean use_typmod=true);
int **Populate_Geometry_Columns**(oid relation_oid, boolean use_typmod=true);

Description

Ensures geometry columns have appropriate type modifiers or spatial constraints to ensure they are registered correctly in the `geometry_columns` view. By default will convert all geometry columns with no type modifier to ones with type modifiers.

For backwards compatibility and for spatial needs such as table inheritance where each child table may have different geometry type, the old check constraint behavior is still supported. If you need the old behavior, you need to pass in the new optional argument as false `use_typmod=false`. When this is done geometry columns will be created with no type modifiers but will have 3 constraints defined. In particular, this means that every geometry column belonging to a table has at least three constraints:

- `enforce_dims_geom` - ensures every geometry has the same dimension (see ST_NDims)
- `enforce_geotype_geom` - ensures every geometry is of the same type (see GeometryType)
- `enforce_srid_geom` - ensures every geometry is in the same projection (see ST_SRID)

If a table `oid` is provided, this function tries to determine the srid, dimension, and geometry type of all geometry columns in the table, adding constraints as necessary. If successful, an appropriate row is inserted into the geometry_columns table, otherwise, the exception is caught and an error notice is raised describing the problem.

If the `oid` of a view is provided, as with a table oid, this function tries to determine the srid, dimension, and type of all the geometries in the view, inserting appropriate entries into the `geometry_columns` table, but nothing is done to enforce constraints.

The parameterless variant is a simple wrapper for the parameterized variant that first truncates and repopulates the geometry_columns table for every spatial table and view in the database, adding spatial constraints to tables where appropriate. It returns a summary of the number of geometry columns detected in the database and the number that were inserted into the `geometry_columns` table. The parameterized version simply returns the number of rows inserted into the `geometry_columns` table.

Availability: 1.4.0

Changed: 2.0.0 By default, now uses type modifiers instead of check constraints to constrain geometry types. You can still use check constraint behavior instead by using the new `use_typmod` and setting it to false.

Enhanced: 2.0.0 `use_typmod` optional argument was introduced that allows controlling if columns are created with typmodifiers or with check constraints.

Examples

This example populates geometry column metadata using typmod behavior. For this to work, there must be existing data.

Code

```
CREATE TABLE public.myspatial_table(gid serial, geom geometry);
INSERT INTO myspatial_table(geom) VALUES(ST_GeomFromText('LINESTRING(1 2,3 4)', 4326) );
SELECT Populate_Geometry_Columns('public.myspatial_table'::regclass);
```

Output

```
1
```

Describe the table to see the typmod-based geometry column.

Code

```
\d myspatial_table
```

Output

```
Table "public.myspatial_table"

Columns:
  gid  integer                    not null default nextval('myspatial_table_gid_seq':: ↩
    regclass)
  geom geometry(LineString,4326)
```

This example changes the geometry columns to use constraints if they are not typmod or already constrained. For this to work, there must be existing data.

Code

```
CREATE TABLE public.myspatial_table_cs(gid serial, geom geometry);
INSERT INTO myspatial_table_cs(geom) VALUES(ST_GeomFromText('LINESTRING(1 2,3 4)', 4326) );
SELECT Populate_Geometry_Columns('public.myspatial_table_cs'::regclass, false);
```

Output

```
1
```

Describe the table to see the constraint-based geometry column.

Code

```
\d myspatial_table_cs
```

Output

```
                         Table "public.myspatial_table_cs"
 Column |   Type   |                            Modifiers
--------+----------+-----------------------------------------------------------------
 gid    | integer  | not null default nextval('myspatial_table_cs_gid_seq'::regclass)
 geom   | geometry |
Check constraints:
    "enforce_dims_geom" CHECK (st_ndims(geom) = 2)
    "enforce_geotype_geom" CHECK (geometrytype(geom) = 'LINESTRING'::text OR geom IS NULL)
    "enforce_srid_geom" CHECK (st_srid(geom) = 4326)
```

7.2.6 UpdateGeometrySRID

UpdateGeometrySRID — Updates the SRID of all features in a geometry column, and the table metadata.

Synopsis

text **UpdateGeometrySRID**(varchar table_name, varchar column_name, integer srid);
text **UpdateGeometrySRID**(varchar schema_name, varchar table_name, varchar column_name, integer srid);
text **UpdateGeometrySRID**(varchar catalog_name, varchar schema_name, varchar table_name, varchar column_name, integer srid);

Description

Updates the SRID of all features in a geometry column, updating constraints and reference in geometry_columns. If the column was enforced by a type definition, the type definition will be changed. Note: uses current_schema() on schema-aware pgsql installations if schema is not provided.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Insert geometries into roads table with a SRID set already using EWKT format:

Code

```
COPY roads (geom) FROM STDIN;
SRID=4326;LINESTRING(0 0, 10 10)
SRID=4326;LINESTRING(10 10, 15 0)
\.
```

This will change the srid of the roads table to 4326 from whatever it was before:

Code

```
SELECT UpdateGeometrySRID('roads', 'geom', 4326);
```

The prior example is equivalent to this DDL statement:

Code

```
ALTER TABLE roads
ALTER COLUMN geom TYPE geometry(MULTILINESTRING, 4326)
  USING ST_SetSRID(geom, 4326);
```

If you got the projection wrong (or brought it in as unknown) in load and you wanted to transform to web mercator all in one shot you can do this with DDL but there is no equivalent PostGIS management function to do so in one go.

Code

```
ALTER TABLE roads
ALTER COLUMN geom TYPE geometry(MULTILINESTRING, 3857) USING ST_Transform(ST_SetSRID(geom, ↩
    4326), 3857) ;
```

See Also

UpdateRasterSRID, ST_SetSRID, ST_Transform, ST_GeomFromEWKT

7.3 Geometry Constructors

7.3.1 ST_Collect

ST_Collect — Creates a GeometryCollection or Multi* geometry from a set of geometries.

Synopsis

geometry **ST_Collect**(geometry g1, geometry g2);
geometry **ST_Collect**(geometry[] g1_array);
geometry **ST_Collect**(geometry set g1field);

Description

Collects geometries into a geometry collection. The result is either a Multi* or a GeometryCollection, depending on whether the input geometries have the same or different types (homogeneous or heterogeneous). The input geometries are left unchanged within the collection.

Variant 1: accepts two input geometries

Variant 2: accepts an array of geometries

Variant 3: aggregate function accepting a rowset of geometries.



Warning
`ST_Collect` does not dissolve boundaries or resolve overlaps. Collecting overlapping polygons can produce an invalid MultiPolygon. If the result must be a valid polygonal geometry with overlaps merged, use ST_Union instead. `ST_Union` also splits linestrings at intersections and may return a single geometry after dissolving boundaries.

If any input geometry is a collection (Multi* or GeometryCollection), `ST_Collect` returns a GeometryCollection, since that is the only type which can contain nested collections. To prevent this, use ST_Dump in a subquery to expand input collections to their atomic elements (see example below).

Availability: 1.4.0 - ST_Collect(geomarray) was introduced. ST_Collect was enhanced to handle more geometries faster.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Two-input variant.

Collect 2D points.

Code

```
SELECT ST_Collect(
    'POINT(1 2)',
    'POINT(-2 3)' );
```

Output

```
MULTIPOINT((1 2),(-2 3))
```

Figure



Collect 3D points.

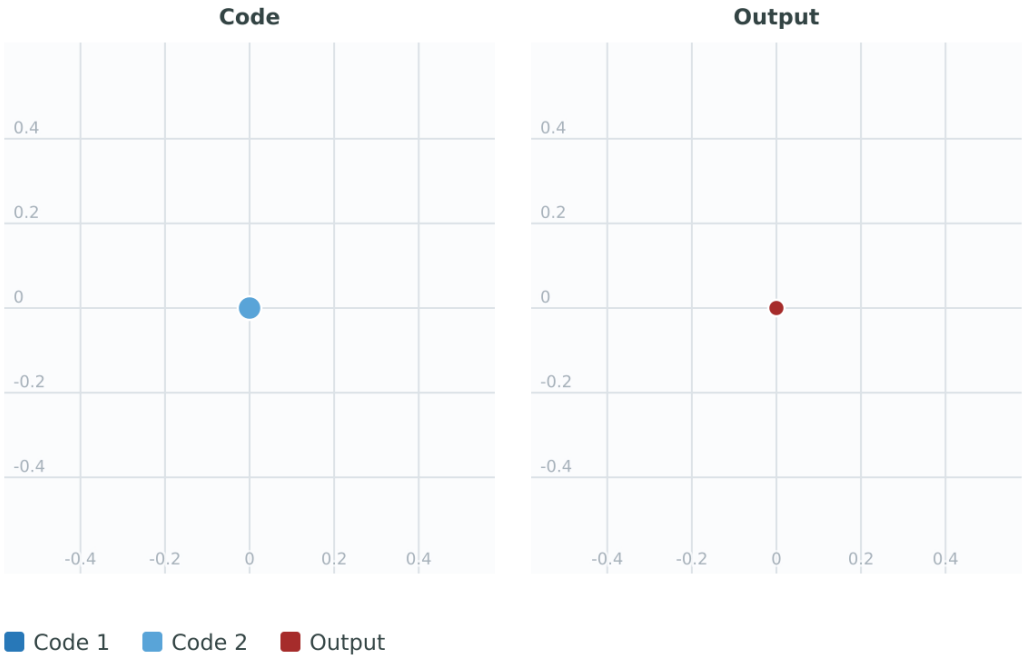
Code

```
SELECT ST_Collect(
        'POINT(1 2 3)',
        'POINT(1 2 4)' );
```

Output

```
MULTIPOINT(1 2 3,1 2 4)
```

Figure

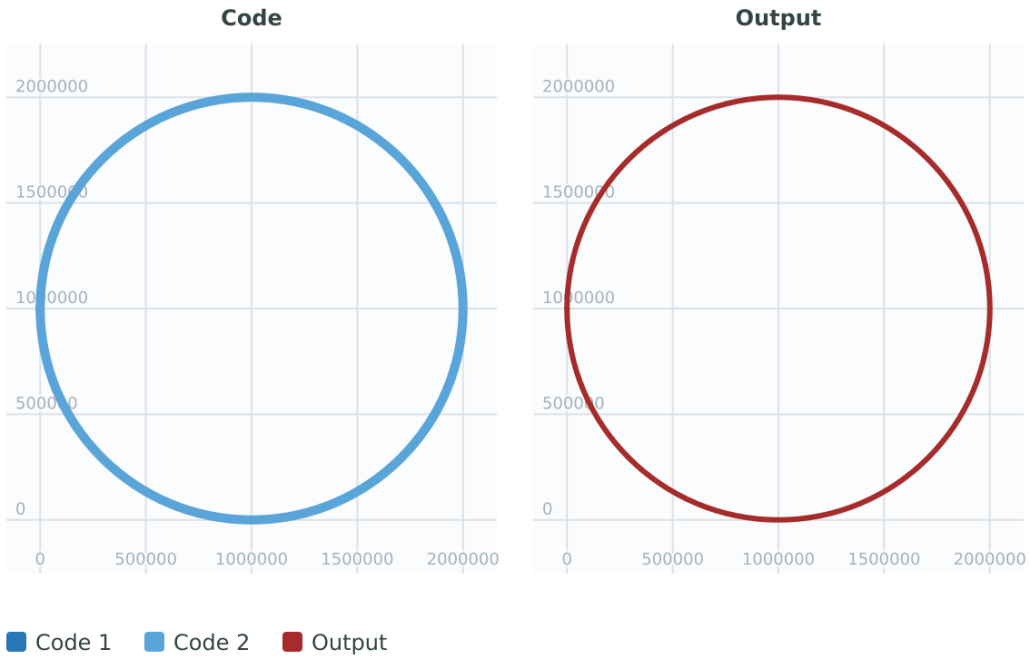


Collect curves.

Code

```
SELECT ST_Collect(
        'CIRCULARSTRING(220268 150415,220227 150505,220227 150406)',
        'CIRCULARSTRING(220227 150406,2220227 150407,220227 150406)');
```

Figure



Array variant.

Using an array constructor for a subquery.

Code

```
SELECT ST_Collect(ARRAY(SELECT geom FROM sometable));
```

Using an array constructor for values.

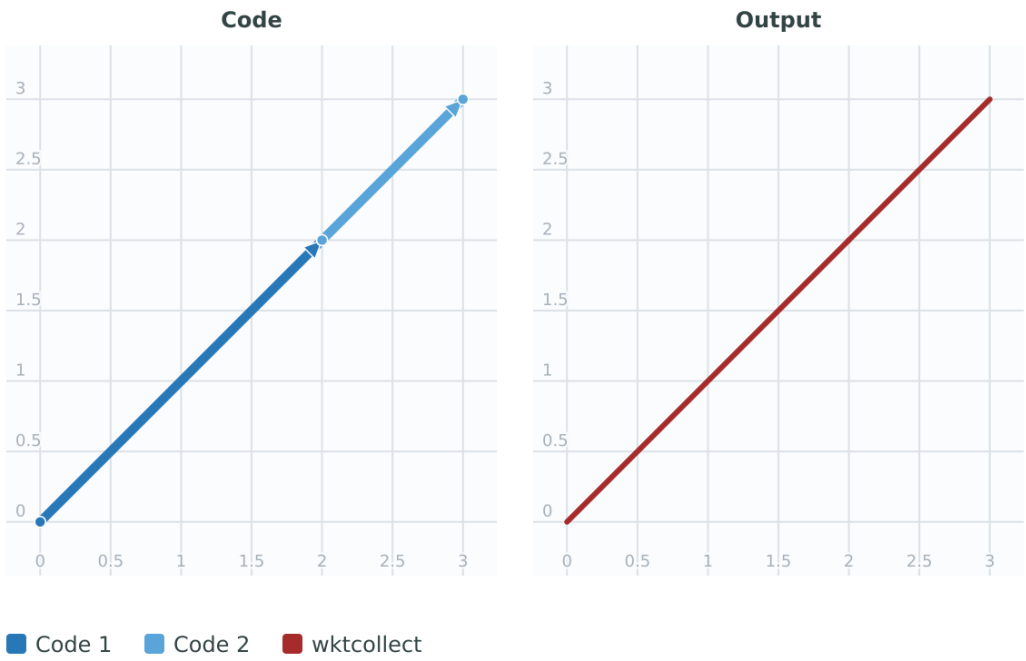
Code

```
SELECT ST_Collect(
        ARRAY[ 'LINESTRING(1 2,3 4)'::geometry,
            'LINESTRING(3 4,4 5)'::geometry ] ) As wktcollect;
```

Output

```
MULTILINESTRING((1 2,3 4),(3 4,4 5))
```

Figure



Aggregate variant.

Creating multiple collections by grouping geometries in a table.

Code

```
SELECT stusps, ST_Collect(geom) AS geom
FROM (
  SELECT stusps, (ST_Dump(geom)).geom AS geom
  FROM somestatetable
) AS f
GROUP BY stusps;
```

See Also

ST_Dump, ST_Union

7.3.2 ST_LineFromMultiPoint

ST_LineFromMultiPoint — Creates a LineString from a MultiPoint geometry.

Synopsis

geometry **ST_LineFromMultiPoint**(geometry aMultiPoint);

Description

Creates a LineString from a MultiPoint geometry.

Use ST_MakeLine to create lines from Point or LineString inputs.

 This function supports 3d and will not drop the z-index.

Examples

Create a 3D line string from a 3D MultiPoint

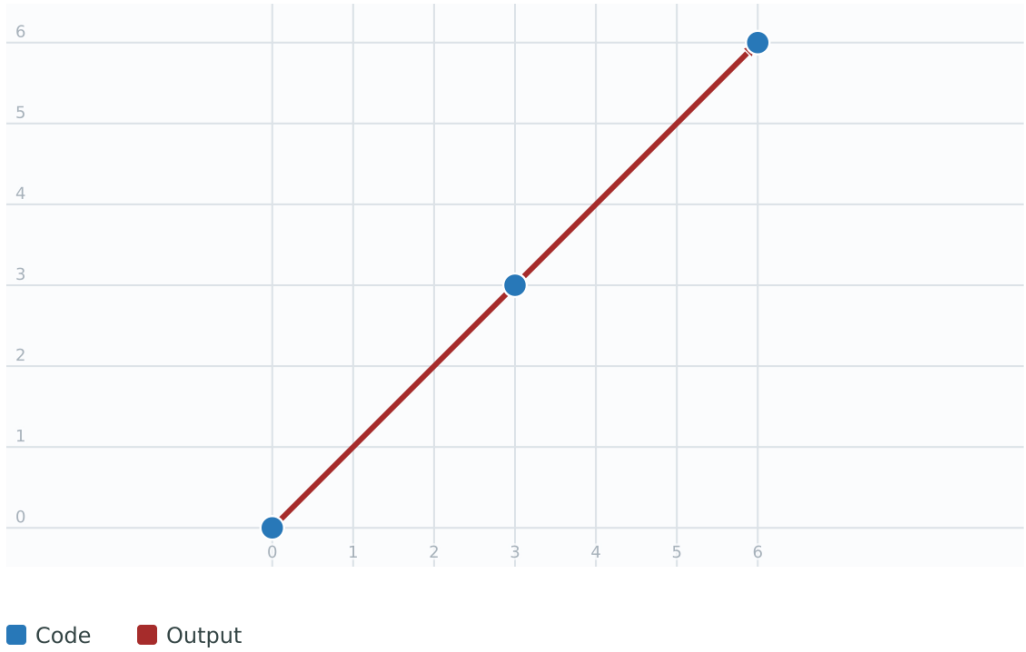
Code

```
SELECT ST_LineFromMultiPoint('MULTIPOINT(1 2 3,4 5 6,7 8 9)');
```

Output

```
LINESTRING(1 2 3,4 5 6,7 8 9)
```

Figure



See Also

ST_AsEWKT, ST_MakeLine

7.3.3 ST_MakeEnvelope

ST_MakeEnvelope — Creates a rectangular Polygon from minimum and maximum coordinates.

Synopsis

geometry **ST_MakeEnvelope**(float xmin, float ymin, float xmax, float ymax, integer srid=unknown);

Description

Creates a rectangular Polygon from the minimum and maximum values for X and Y. Input values must be in the spatial reference system specified by the SRID. If no SRID is specified the unknown spatial reference system (SRID 0) is used.

Availability: 1.5

Enhanced: 2.0: Ability to specify an envelope without specifying an SRID was introduced.

Examples

Building a bounding box polygon.

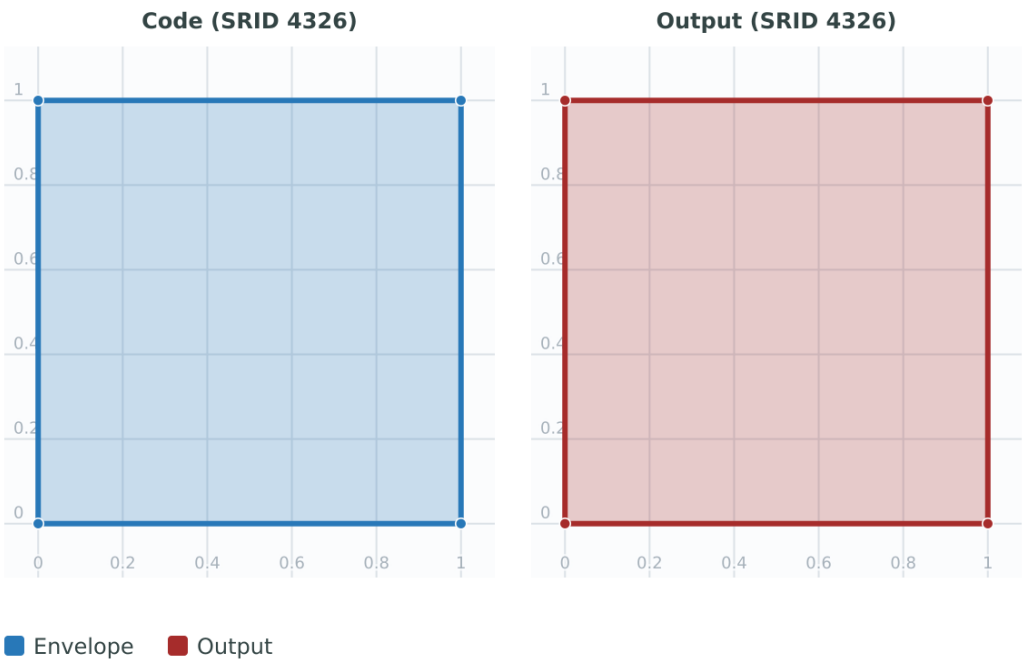
Code

```
SELECT ST_MakeEnvelope(10, 10, 11, 11, 4326);
```

Output

```
POLYGON((10 10, 10 11, 11 11, 11 10, 10 10))
```

Figure



See Also

ST_MakePoint, ST_MakeLine, ST_MakePolygon, ST_TileEnvelope

7.3.4 ST_MakeLine

ST_MakeLine — Creates a LineString from Point, MultiPoint, or LineString geometries.

Synopsis

geometry **ST_MakeLine**(geometry geom1, geometry geom2);
geometry **ST_MakeLine**(geometry[] geoms_array);
geometry **ST_MakeLine**(geometry set geoms);

Description

Creates a LineString containing the points of Point, MultiPoint, or LineString geometries. Other geometry types cause an error.

Variant 1: accepts two input geometries

Variant 2: accepts an array of geometries

Variant 3: aggregate function accepting a rowset of geometries. To ensure the order of the input geometries use `ORDER BY` in the function call, or a subquery with an `ORDER BY` clause.

Repeated nodes at the beginning of input LineStrings are collapsed to a single point. Repeated points in Point and MultiPoint inputs are not collapsed. Components of MultiLineString are handled in the order they appear in the collection. ST_RemoveRepeatedPoints can be used to collapse repeated points from the output LineString.

This function supports 3d and will not drop the z-index.

Availability: 3.7.0 - Support for MultiLineString input elements was introduced

Availability: 2.3.0 - Support for MultiPoint input elements was introduced

Availability: 2.0.0 - Support for LineString input elements was introduced

Availability: 1.4.0 - ST_MakeLine(geomarray) was introduced. ST_MakeLine aggregate functions was enhanced to handle more points faster.

Examples

Two-input variant.

Create a line composed of two points.

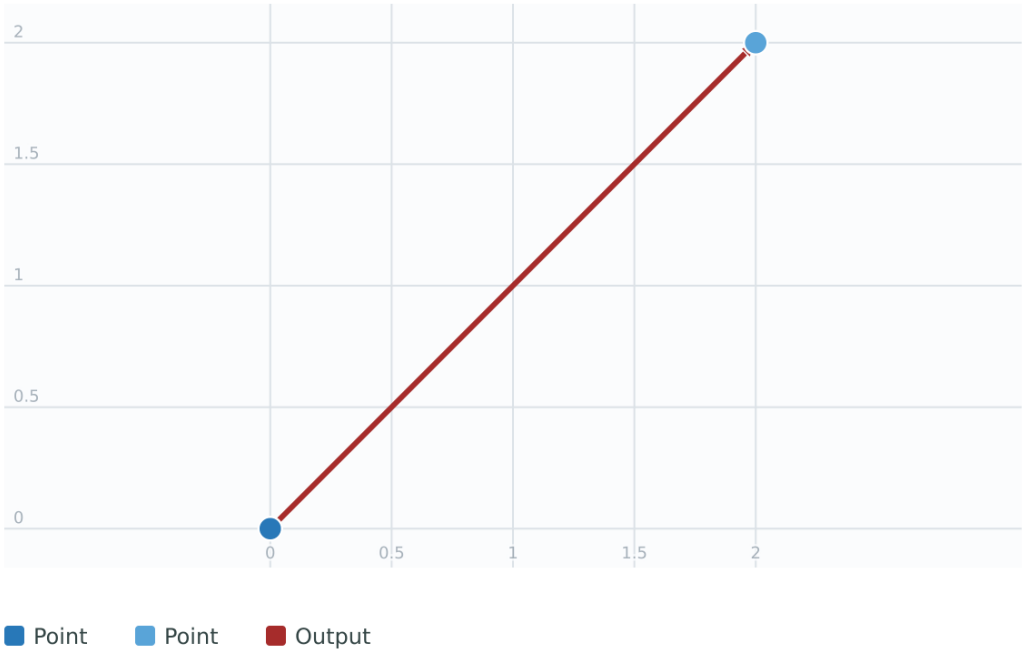
Code

```
SELECT ST_MakeLine(ST_Point(1, 2), ST_Point(3, 4));
```

Output

```
LINESTRING(1 2,3 4)
```

Figure



Create a 3D line from two 3D points.

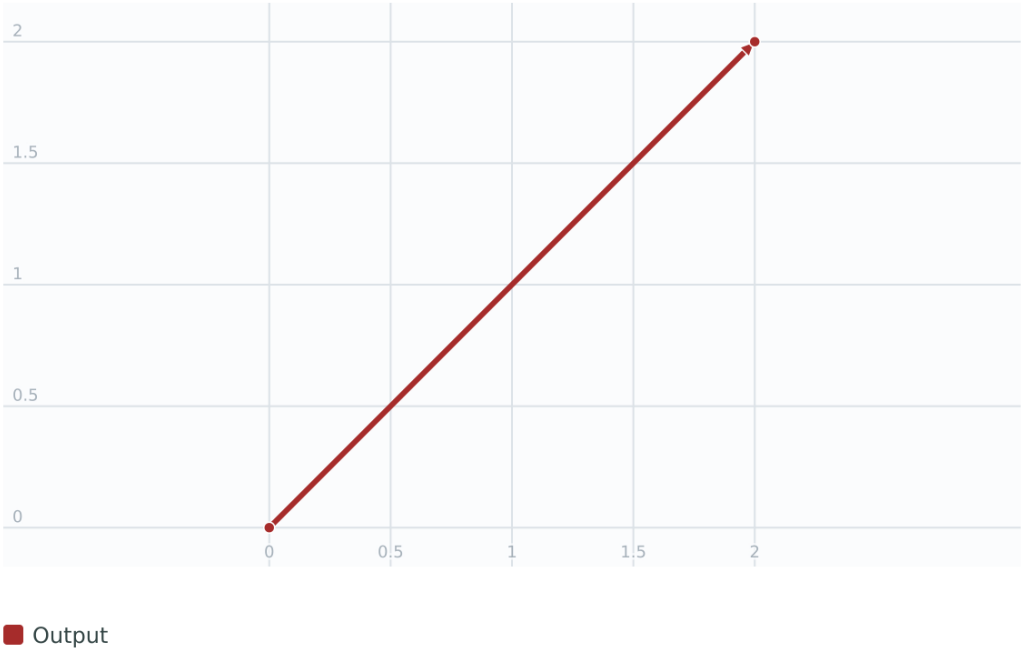
Code

```
SELECT ST_MakeLine(ST_MakePoint(1, 2, 3), ST_MakePoint(3, 4, 5));
```

Output

```
LINESTRING(1 2 3,3 4 5)
```

Figure



Create a line from two disjoint LineStrings.

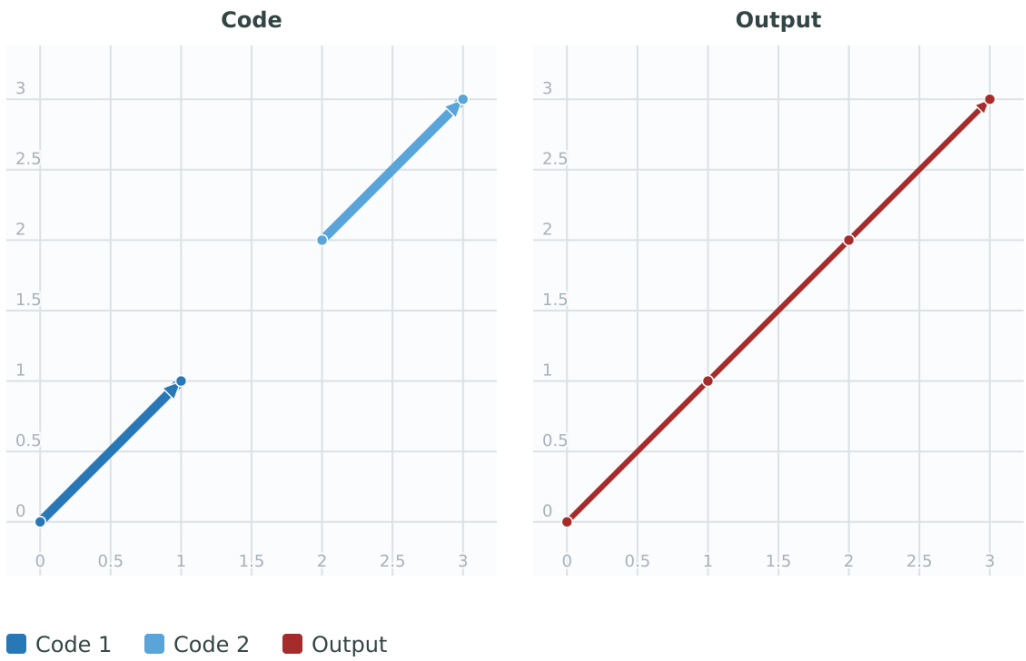
Code

```
SELECT ST_MakeLine('LINESTRING(0 0,1 1)', 'LINESTRING(2 2,3 3)');
```

Output

```
LINESTRING(0 0,1 1,2 2,3 3)
```

Figure



Array variant.

Create a line from an array formed by a subquery with ordering.

Code

```
SELECT ST_MakeLine(ARRAY(SELECT ST_Centroid(geom) FROM visit_locations ORDER BY visit_time)) ↩
    ;
```

Create a 3D line from an array of 3D points

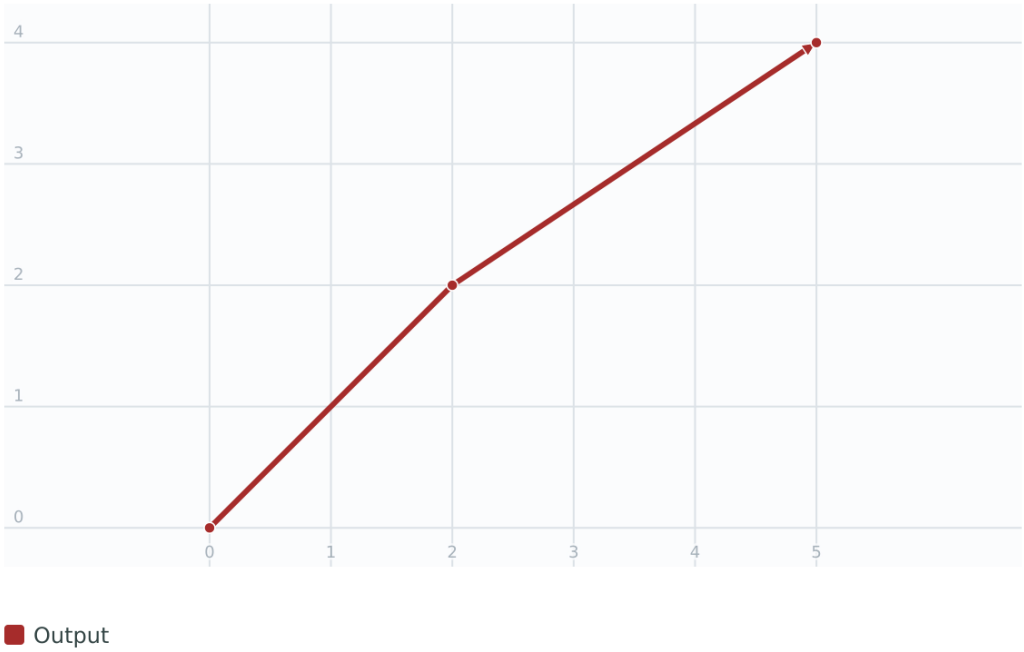
Code

```
SELECT ST_MakeLine(ARRAY[ST_MakePoint(1, 2, 3), ST_MakePoint(3, 4, 5), ST_MakePoint(6, 6, 6) ↩
    ]);
```

Output

```
LINESTRING(1 2 3,3 4 5,6 6 6)
```

Figure



Aggregate variant.

This example queries time-based sequences of GPS points from a set of tracks and creates one record for each track. The result geometries are LineStrings composed of the GPS track points in the order of travel.

Using aggregate `ORDER BY` provides a correctly-ordered LineString.

Code

```
SELECT gps.track_id, ST_MakeLine(gps.geom ORDER BY gps_time) AS geom
FROM gps_points AS gps
GROUP BY gps.track_id;
```

When ordering within the aggregate is not possible, ordering in a subquery can be used. However, sometimes the query plan may not respect the order of the subquery.

Code

```
SELECT gps.track_id, ST_MakeLine(gps.geom) AS geom
FROM (
  SELECT track_id, gps_time, geom
  FROM gps_points
  ORDER BY track_id, gps_time
) AS gps
GROUP BY gps.track_id;
```

See Also

ST_RemoveRepeatedPoints, ST_AsEWKT, ST_AsText, ST_GeomFromText, ST_MakePoint, ST_Point

7.3.5 ST_MakePoint

ST_MakePoint — Creates a 2D, 3DZ or 4D Point.

Synopsis

geometry **ST_MakePoint**(float x, float y);

geometry **ST_MakePoint**(float x, float y, float z);

geometry **ST_MakePoint**(float x, float y, float z, float m);

Description

Creates a 2D XY, 3D XYZ or 4D XYZM Point geometry. Use ST_MakePointM to make points with XYM coordinates.

Use ST_SetSRID to specify a SRID for the created point.

While not OGC-compliant, `ST_MakePoint` is faster than ST_GeomFromText and ST_PointFromText. It is also easier to use for numeric coordinate values.



Note
For geodetic coordinates, `X` is longitude and `Y` is latitude



Note
The functions ST_Point, ST_PointZ, ST_PointM, and ST_PointZM can be used to create points with a given SRID.



This function supports 3d and will not drop the z-index.

Examples

This example creates a point with an unknown SRID.

Code

```
SELECT ST_MakePoint(-71.1043443253471, 42.3150676015829);
```

This example creates a point in the WGS 84 geodetic CRS.

Code

```
SELECT ST_SetSRID(ST_MakePoint(-71.1043443253471, 42.3150676015829), 4326);
```

This example creates a 3D point with altitude.

Code

```
SELECT ST_MakePoint(1, 2, 1.5);
```

This example returns the Z coordinate of a point.

Code

```
SELECT ST_Z(ST_MakePoint(1, 2, 1.5));
```

Output

```
1.5
```

See Also

ST_GeomFromText, ST_PointFromText, ST_SetSRID, ST_MakePointM, ST_Point, ST_PointZ, ST_PointM, ST_PointZM

7.3.6 ST_MakePointM

ST_MakePointM — Creates a Point from X, Y and M values.

Synopsis

geometry **ST_MakePointM**(float x, float y, float m);

Description

Creates a point with X, Y and M (measure) ordinates. Use ST_MakePoint to make points with XY, XYZ, or XYZM coordinates. Use ST_SetSRID to specify a SRID for the created point.



Note
For geodetic coordinates, `X` is longitude and `Y` is latitude



Note
The functions ST_PointM, and ST_PointZM can be used to create points with an M value and a given SRID.

Examples



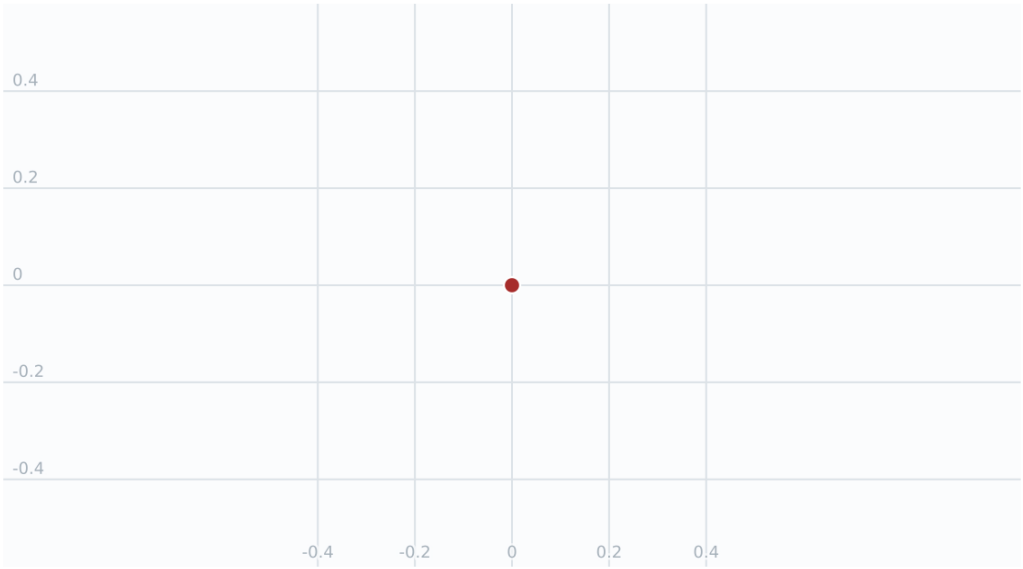
Note
ST_AsEWKT is used for text output because ST_AsText does not support M values.

Create point with unknown SRID.

Code

```
SELECT ST_MakePointM(-71.1043443253471, 42.3150676015829, 10);
```

Figure

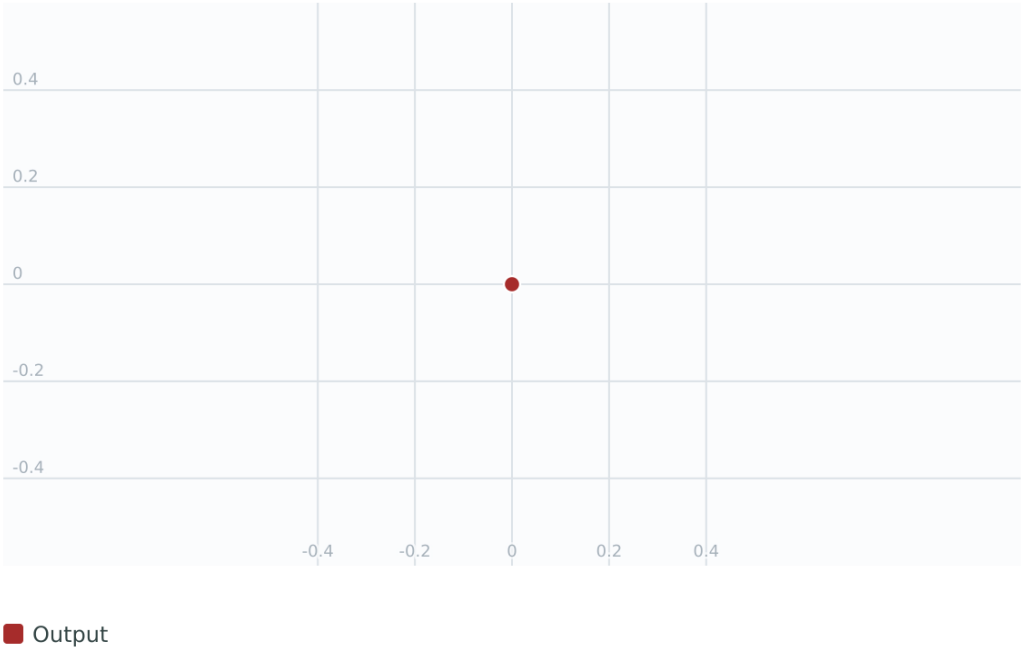


Create point with a measure in the WGS 84 geodetic coordinate system.

Code

```
SELECT ST_SetSRID(ST_MakePointM(-71.104, 42.315, 10), 4326);
```

Figure



Get measure of created point.

Code

```
SELECT ST_M(ST_MakePointM(-71.104, 42.315, 10));
```

Output

```
10
```

See Also

ST_MakePoint, ST_SetSRID, ST_PointM, ST_PointZM

7.3.7 ST_MakePolygon

ST_MakePolygon — Creates a Polygon or CurvePolygon from a shell and optional list of holes.

Synopsis

geometry **ST_MakePolygon**(geometry linestring);

geometry **ST_MakePolygon**(geometry outerlinestring, geometry[] interiorlinestrings);

Description

Creates a Polygon or CurvePolygon formed by the given shell and optional array of holes. Input geometries must be closed LineStrings, CircularStrings, CompoundCurves, or NURBSCurves (rings). If any input ring is curved, the result is a CurvePolygon.

Variant 1: Accepts one shell ring.

Variant 2: Accepts a shell ring and an array of inner (hole) rings. A geometry array can be constructed using the PostgreSQL array_agg(), ARRAY[] or ARRAY() constructs.



Note
This function does not accept MultiLineStrings. Use ST_LineMerge to generate a LineString, or ST_Dump to extract LineStrings.

 This function supports 3d and will not drop the z-index.

Enhanced: 3.7.0 - Support for curved input rings was introduced.

Examples

Single input variant.

Create a Polygon from a 2D LineString.

Code

```
SELECT ST_MakePolygon('LINESTRING(75 29,77 29,77 29,75 29)');
```

Create a Polygon from an open LineString, using ST_StartPoint and ST_AddPoint to close it.

Code

```
SELECT ST_MakePolygon(ST_AddPoint(foo.open_line, ST_StartPoint(foo.open_line)) )
FROM (
  SELECT 'LINESTRING(75 29,77 29,77 29,75 29)'::geometry As open_line) As foo;
```

Create a Polygon from a 3D LineString

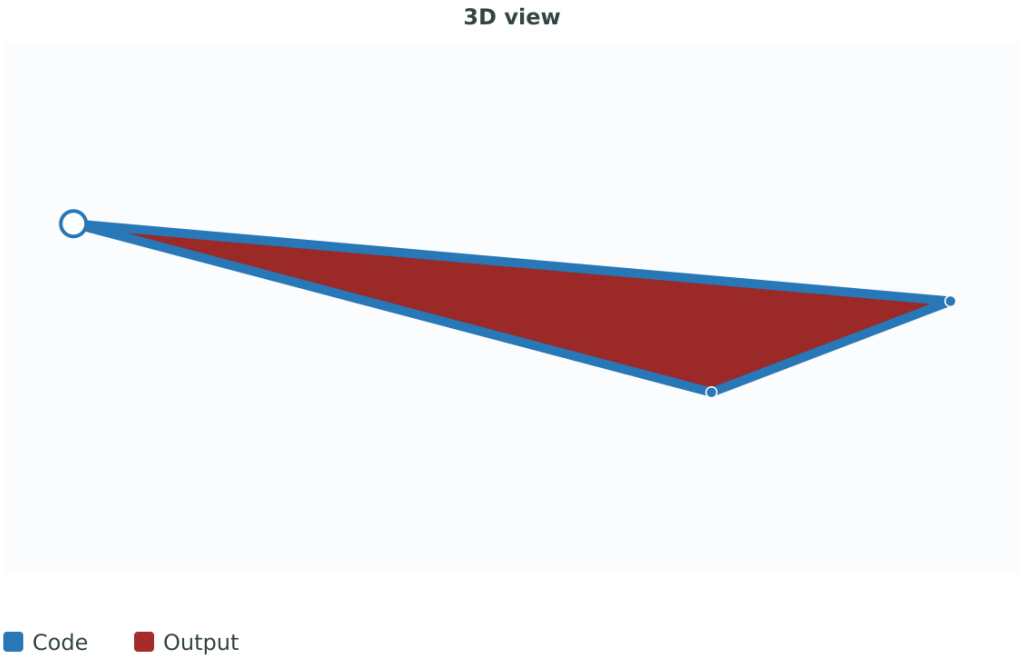
Code

```
SELECT ST_MakePolygon('LINESTRING(75.15 29.53 1,77 29 1,77.6 29.5 1,75.15 29.53 1)');
```

Output

```
POLYGON((75.15 29.53 1,77 29 1,77.6 29.5 1,75.15 29.53 1))
```

Figure

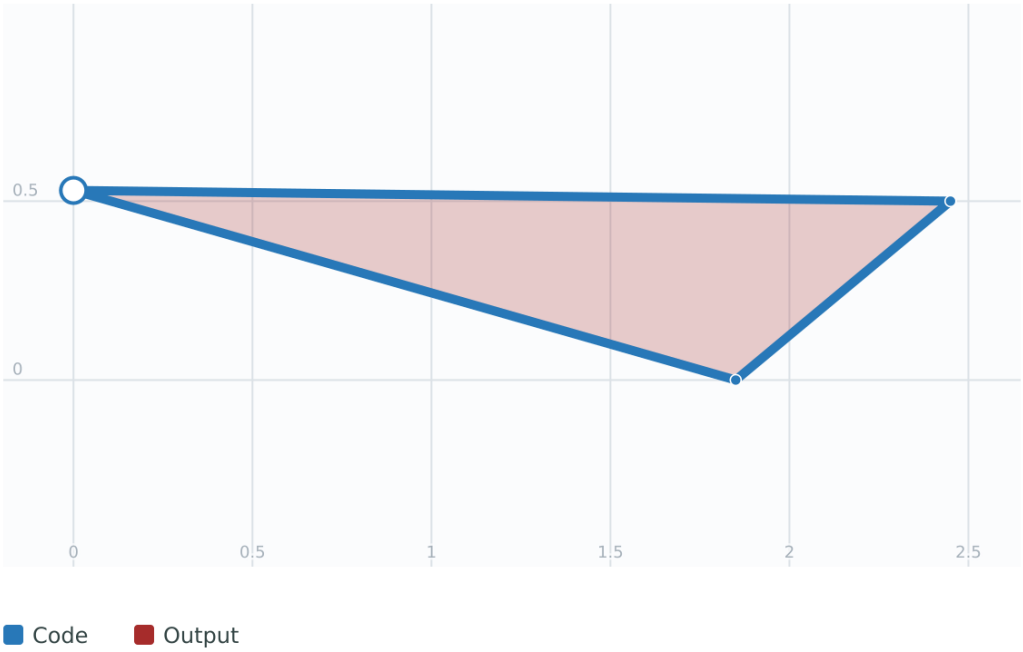


Create a Polygon from a LineString with measures

Code

```
SELECT ST_MakePolygon('LINESTRINGM(75.15 29.53 1,77 29 1,77.6 29.5 2, 75.15 29.53 2)');
```

Figure



Create a CurvePolygon from a CircularString shell.

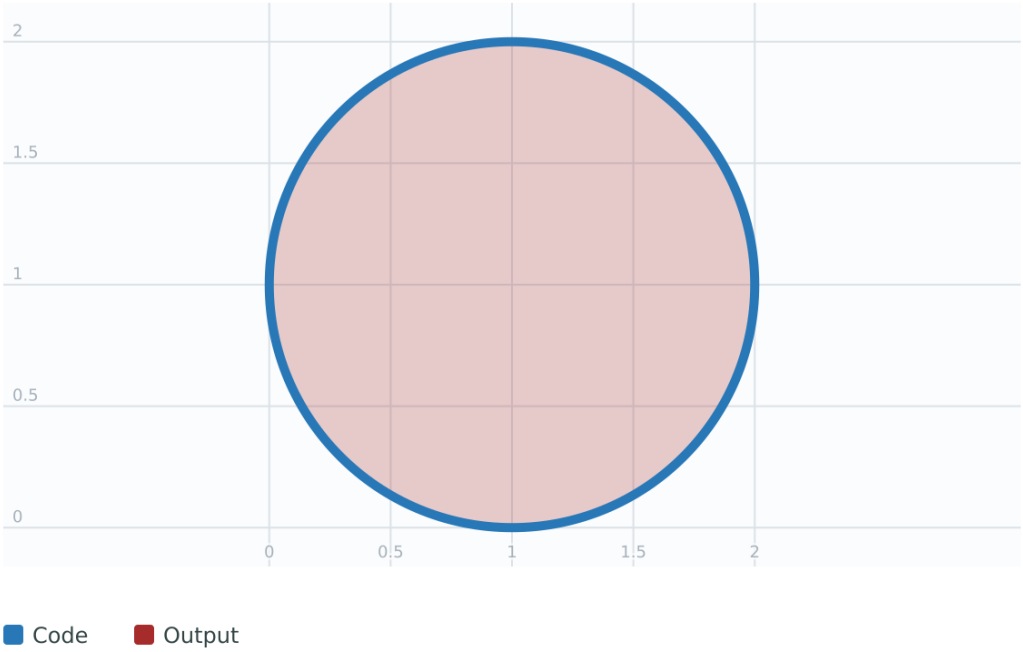
Code

```
SELECT ST_MakePolygon('CIRCULARSTRING(0 0,1 1,2 0,1 -1,0 0)');
```

Output

```
CURVEPOLYGON(CIRCULARSTRING(0 0,1 1,2 0,1 -1,0 0))
```

Figure



Outer shell with inner holes variant.

Create a donut Polygon with an extra hole

Code

```
SELECT ST_MakePolygon(
    ST_ExteriorRing(ST_Buffer(ring.line, 10)),
    ARRAY[  ST_Translate(ring.line, 1, 1),
        ST_ExteriorRing(ST_Buffer(ST_Point(20, 20), 1)) ]
    )
FROM (SELECT ST_ExteriorRing(ST_Buffer(ST_Point(10, 10), 10, 10)) AS line ) AS ring;
```

Create a set of province boundaries with holes representing lakes. The input is a table of province Polygons/MultiPolygons and a table of water linestrings. Lines forming lakes are determined by using ST_IsClosed. The province linework is extracted by using ST_Boundary. As required by `ST_MakePolygon`, the boundary is forced to be a single LineString by using ST_LineMerge. (However, note that if a province has more than one region or has islands this will produce an invalid polygon.) Using a LEFT JOIN ensures all provinces are included even if they have no lakes.



Note
The CASE construct is used because passing a null array into ST_MakePolygon results in a NULL return value.

Code

```
SELECT p.gid, p.province_name,
  CASE
    WHEN array_agg(w.geom) IS NULL THEN p.geom
    ELSE ST_MakePolygon(
      ST_LineMerge(ST_Boundary(p.geom)),
      array_agg(w.geom)
    )
  END AS geom
FROM provinces p
LEFT JOIN waterlines w
  ON ST_Within(w.geom, p.geom)
  AND ST_IsClosed(w.geom)
GROUP BY p.gid, p.province_name, p.geom;
```

Another technique is to utilize a correlated subquery and the ARRAY() constructor that converts a row set to an array.

Code

```
SELECT p.gid, p.province_name,
  CASE
    WHEN EXISTS (
      SELECT w.geom
      FROM waterlines w
      WHERE ST_Within(w.geom, p.geom)
      AND ST_IsClosed(w.geom)
    )
    THEN ST_MakePolygon(
      ST_LineMerge(ST_Boundary(p.geom)),
      ARRAY(
        SELECT w.geom
        FROM waterlines w
        WHERE ST_Within(w.geom, p.geom)
        AND ST_IsClosed(w.geom)
      )
```

```
    )
    ELSE p.geom
  END AS geom
FROM provinces p;
```

See Also

ST_BuildArea ST_Polygon

7.3.8 ST_Point

ST_Point — Creates a Point with X, Y and SRID values.

Synopsis

geometry **ST_Point**(float x, float y);

geometry **ST_Point**(float x, float y, integer srid=unknown);

Description

Returns a Point with the given X and Y coordinate values. This is the SQL-MM equivalent for ST_MakePoint that takes just X and Y.



Note
For geodetic coordinates, `X` is longitude and `Y` is latitude

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

 This method implements the SQL/MM specification. SQL-MM 3: 6.1.2

Examples

Geometry.

Code

```
SELECT ST_Point(-71.104, 42.315);
```

Creating a point with SRID specified:

Code

```
SELECT ST_Point(-71.104, 42.315, 4326);
```

Alternative way of specifying SRID:

Code

```
SELECT ST_SetSRID(ST_Point(-71.104, 42.315), 4326);
```

Create geography points using the `::` cast syntax:

Code

```
SELECT ST_Point(-71.104, 42.315, 4326)::geography;
```

Pre-PostGIS 3.2 code, using `CAST`:

Code

```
SELECT CAST(ST_SetSRID(ST_Point(-71.104, 42.315), 4326) AS geography);
```

If the point coordinates are not in a geodetic coordinate system (such as WGS84), then they must be reprojected before casting to a geography. In this example a point in Pennsylvania State Plane feet (SRID 2273) is projected to WGS84 (SRID 4326).

Code

```
SELECT ST_Transform(ST_Point(3637510, 3014852, 2273), 4326)::geography;
```

See Also

ST_MakePoint, ST_PointZ, ST_PointM, ST_PointZM, ST_SetSRID, ST_Transform

7.3.9 ST_PointZ

ST_PointZ — Creates a Point with X, Y, Z and SRID values.

Synopsis

geometry **ST_PointZ**(float x, float y, float z, integer srid=unknown);

Description

Returns an Point with the given X, Y and Z coordinate values, and optionally an SRID number.

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

Examples

Code

```
SELECT ST_PointZ(-71.104, 42.315, 3.4, 4326)
```

Code

```
SELECT ST_PointZ(-71.104, 42.315, 3.4, srid => 4326)
```

Code

```
SELECT ST_PointZ(-71.104, 42.315, 3.4)
```

See Also

ST_MakePoint, ST_Point, ST_PointM, ST_PointZM

7.3.10 ST_PointM

ST_PointM — Creates a Point with X, Y, M and SRID values.

Synopsis

geometry **ST_PointM**(float x, float y, float m, integer srid=unknown);

Description

Returns an Point with the given X, Y and M coordinate values, and optionally an SRID number.

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

Examples

Code

```
SELECT ST_PointM(-71.104, 42.315, 3.4, 4326)
```

Code

```
SELECT ST_PointM(-71.104, 42.315, 3.4, srid => 4326)
```

Code

```
SELECT ST_PointM(-71.104, 42.315, 3.4)
```

See Also

ST_MakePoint, ST_Point, ST_PointZ, ST_PointZM

7.3.11 ST_PointZM

ST_PointZM — Creates a Point with X, Y, Z, M and SRID values.

Synopsis

geometry **ST_PointZM**(float x, float y, float z, float m, integer srid=unknown);

Description

Returns an Point with the given X, Y, Z and M coordinate values, and optionally an SRID number.

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

Examples

Code

```
SELECT ST_PointZM(-71.104, 42.315, 3.4, 4.5, 4326)
```

Code

```
SELECT ST_PointZM(-71.104, 42.315, 3.4, 4.5, srid => 4326)
```

Code

```
SELECT ST_PointZM(-71.104, 42.315, 3.4, 4.5)
```

See Also

ST_MakePoint, ST_Point, ST_PointM, ST_PointZ, ST_SetSRID

7.3.12 ST_Polygon

ST_Polygon — Creates a Polygon from a LineString with a specified SRID.

Synopsis

geometry **ST_Polygon**(geometry lineString, integer srid);

Description

Returns a polygon built from the given LineString and sets the spatial reference system from the `srid`.

ST_Polygon is similar to ST_MakePolygon Variant 1 with the addition of setting the SRID.

To create polygons with holes use ST_MakePolygon Variant 2 and then ST_SetSRID.



Note
This function does not accept MultiLineStrings. Use ST_LineMerge to generate a LineString, or ST_Dump to extract LineStrings.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1.


This method implements the SQL/MM specification. SQL-MM 3: 8.3.2


This function supports 3d and will not drop the z-index.

Examples

Create a 2D polygon.

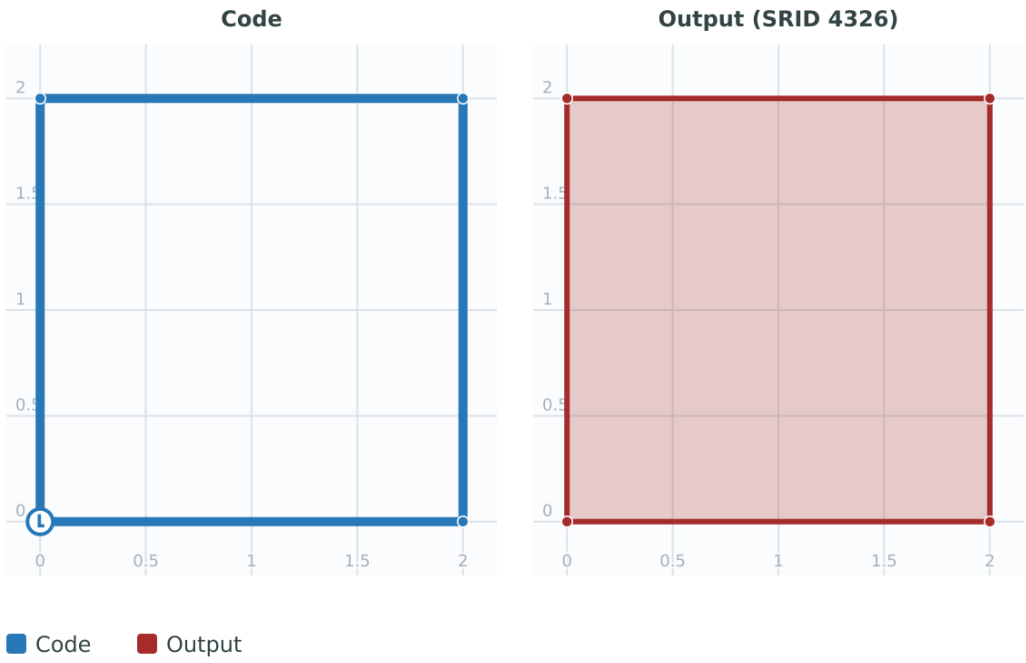
Code

```
SELECT ST_Polygon('LINESTRING(75 29,77 29,77 31,75 31,75 29)'::geometry, 4326);
```

Output

```
POLYGON((75 29,77 29,77 31,75 31,75 29))
```

Figure



Create a 3D polygon.

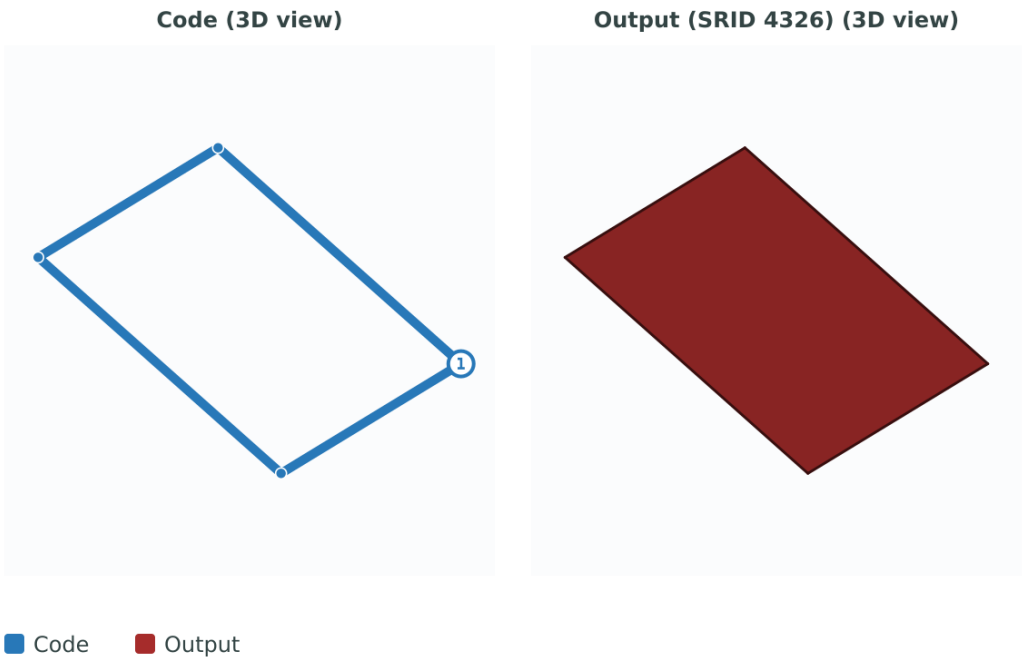
Code

```
SELECT ST_Polygon('LINESTRING(75 29 1,77 29 2,77 31 2,75 31 1,75 29 1)'::geometry, 4326);
```

Output

```
SRID=4326;POLYGON((75 29 1,77 29 2,77 31 2,75 31 1,75 29 1))
```

Figure



See Also

ST_AsEWKT, ST_AsText, ST_GeomFromEWKT, ST_GeomFromText, ST_LineMerge, ST_MakePolygon

7.3.13 ST_TileEnvelope

ST_TileEnvelope — Creates a rectangular Polygon in Web Mercator (SRID:3857) using the XYZ tile system.

Synopsis

geometry **ST_TileEnvelope**(integer tileZoom, integer tileX, integer tileY, geometry bounds=SRID=3857;LINESTRING(-20037508.342 -20037508.342789,20037508.342789 20037508.342789), float margin=0.0);

Description

Creates a rectangular Polygon giving the extent of a tile in the XYZ tile system. The tile is specified by the zoom level Z and the XY index of the tile in the grid at that level. Can be used to define the tile bounds required by ST_AsMVTGeom to convert geometry into the MVT tile coordinate space.

By default, the tile envelope is in the Web Mercator coordinate system (SRID:3857) using the standard range of the Web Mercator system (-20037508.342789, 20037508.342789). This is the most common coordinate system used for MVT tiles. The optional `bounds` parameter can be used to generate tiles in any coordinate system. It is a geometry that has the SRID and extent of the "Zoom Level zero" square within which the XYZ tile system is inscribed.

The optional `margin` parameter can be used to expand a tile by the given percentage. E.g. `margin=0.125` expands the tile by 12.5%, which is equivalent to buffer=512 when the tile extent size is 4096, as used in ST_AsMVTGeom. This is useful to create a tile buffer to include data lying outside of the tile's visible area, but whose existence affects the tile rendering. For example, a city name (a point) could be near an edge of a tile, so its label should be rendered on two tiles, even though the point is located in the visible area of just one tile. Using expanded tiles in a query will include the city point in both tiles. Use a negative value to shrink the tile instead. Values less than -0.5 are prohibited because that would eliminate the tile completely. Do not specify a margin when using with `ST_AsMVTGeom`. See the example for ST_AsMVT.

Enhanced: 3.1.0 Added margin parameter.

Availability: 3.0.0

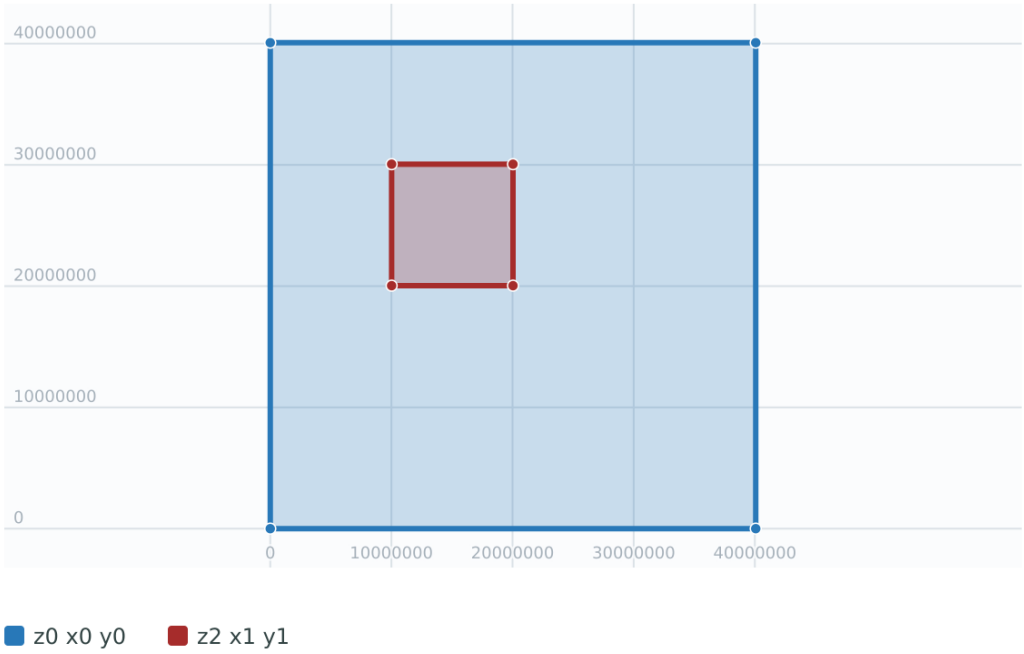
Examples

Show tile (2, 1, 1) within the root tile (0, 0, 0).

Code

```
SELECT ST_TileEnvelope(0, 0, 0) AS input_z0_x0_y0, ST_TileEnvelope(2, 1, 1) AS z2_x1_y1;
```

Figure



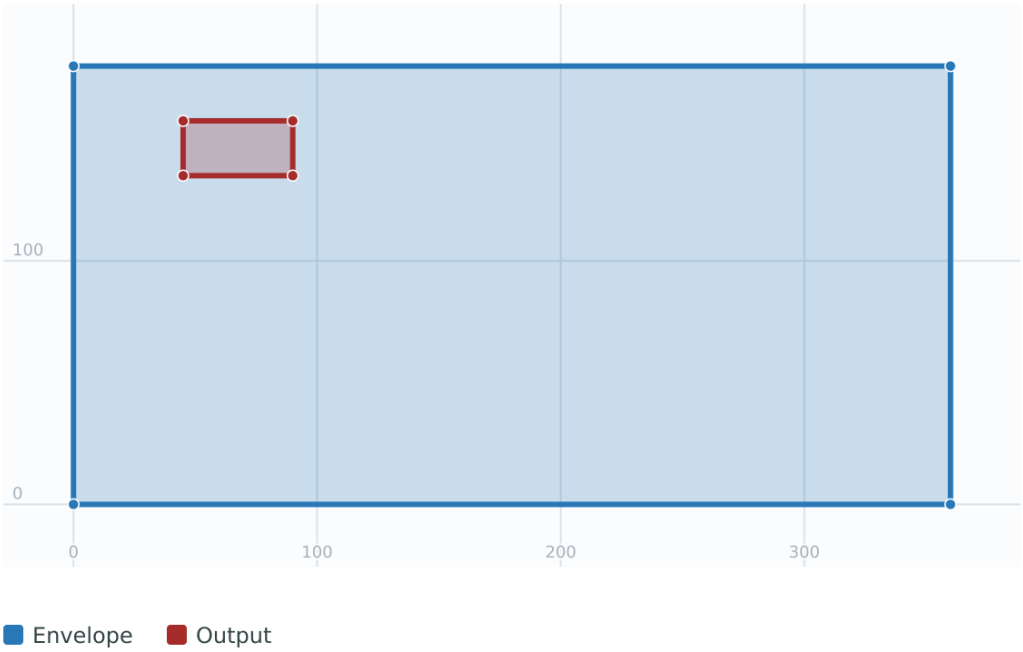
Code

```
SELECT ST_TileEnvelope(3, 1, 1, ST_MakeEnvelope(-180, -90, 180, 90, 4326));
```

Output

```
POLYGON((-135 45,-135 67.5,-90 67.5,-90 45,-135 45))
```

Figure



See Also

ST_MakeEnvelope

7.3.14 ST_HexagonGrid

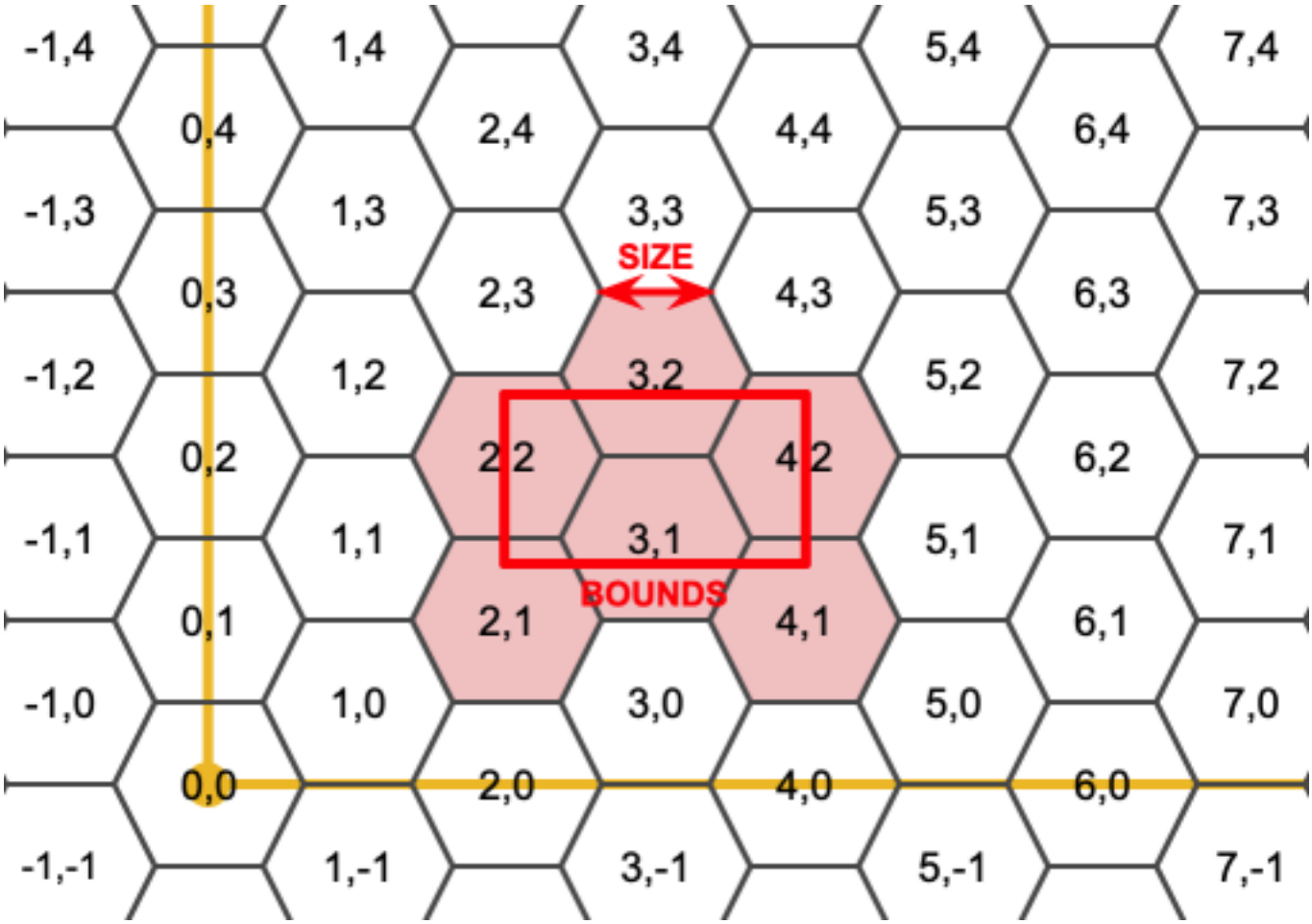
ST_HexagonGrid — Returns a set of hexagons and cell indices that completely cover the bounds of the geometry argument.

Synopsis

setof record **ST_HexagonGrid**(float8 size, geometry bounds);

Description

Starts with the concept of a hexagon tiling of the plane. (Not a hexagon tiling of the globe; for H3 use the h3-pg extension.) For a given planar SRS, and a given edge size, starting at the origin of the SRS, there is one unique hexagonal tiling of the plane, Tiling(SRS, Size). This function answers the question: what hexagons in a given Tiling(SRS, Size) overlap with a given bounds.



The SRS for the output hexagons is the SRS provided by the bounds geometry.

Doubling or tripling the edge size of the hexagon generates a new parent tiling that fits with the origin tiling. Unfortunately, it is not possible to generate parent hexagon tilings that the child tiles perfectly fit inside.

Availability: 3.1.0

Examples

Unit hexagons covering rectangular bounds.

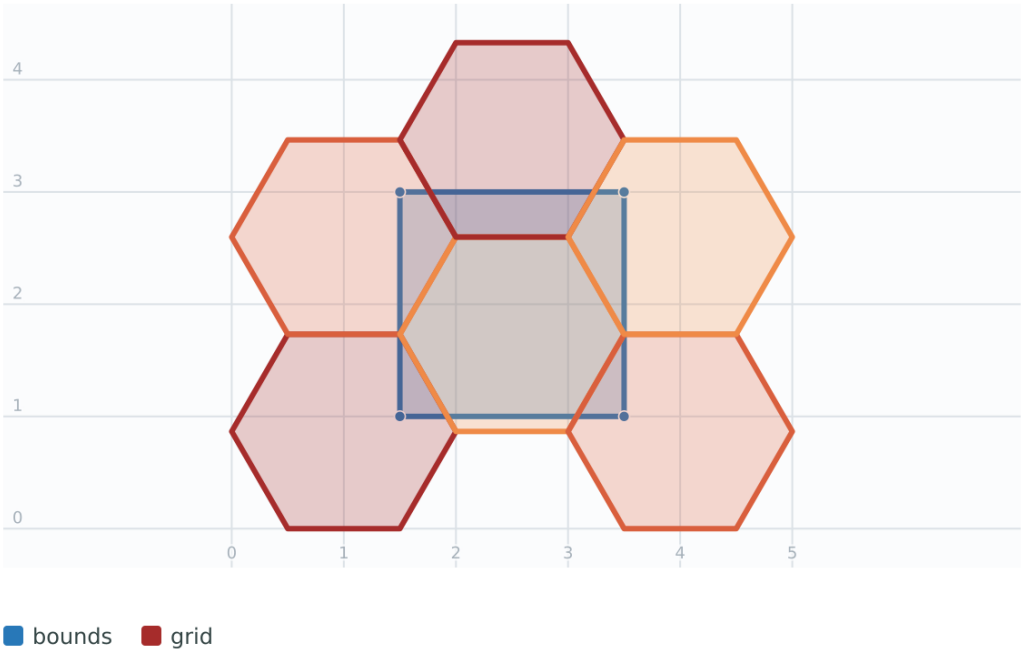
Code

```
WITH bounds AS (
    SELECT ST_MakeEnvelope(2, 1, 4, 3, 3857) AS geom
), grid AS (
    SELECT hex.*
    FROM bounds
    CROSS JOIN LATERAL ST_HexagonGrid(1, geom) AS hex
)
SELECT (SELECT geom FROM bounds) AS input_bounds,
    ST_Collect(geom ORDER BY i, j) AS grid
FROM grid;
```

Output

```
POLYGON((2 1,2 3,4 3,4 1,2 1)) | MULTIPOLYGON(((0.5 0.866,1 0,2 0,2.5 0.866,2 1.732,1 1.732,0.5 0.866)),((0.5 2.598,1 ↩
    1.732,2 1.732,2.5 2.598,2 3.464,1 3.464,0.5 2.598)),((2 1.732,2.5 0.866,3.5 0.866,4 1.732,3.5 2.598,2.5 2.598,2 ↩
    1.732)),((2 3.464,2.5 2.598,3.5 2.598,4 3.464,3.5 4.33,2.5 4.33,2 3.464)),((3.5 0.866,4 0,5 0,5.5 0.866,5 1.732,4 ↩
    1.732,3.5 0.866)),((3.5 2.598,4 1.732,5 1.732,5.5 2.598,5 3.464,4 3.464,3.5 2.598)))
```

Figure

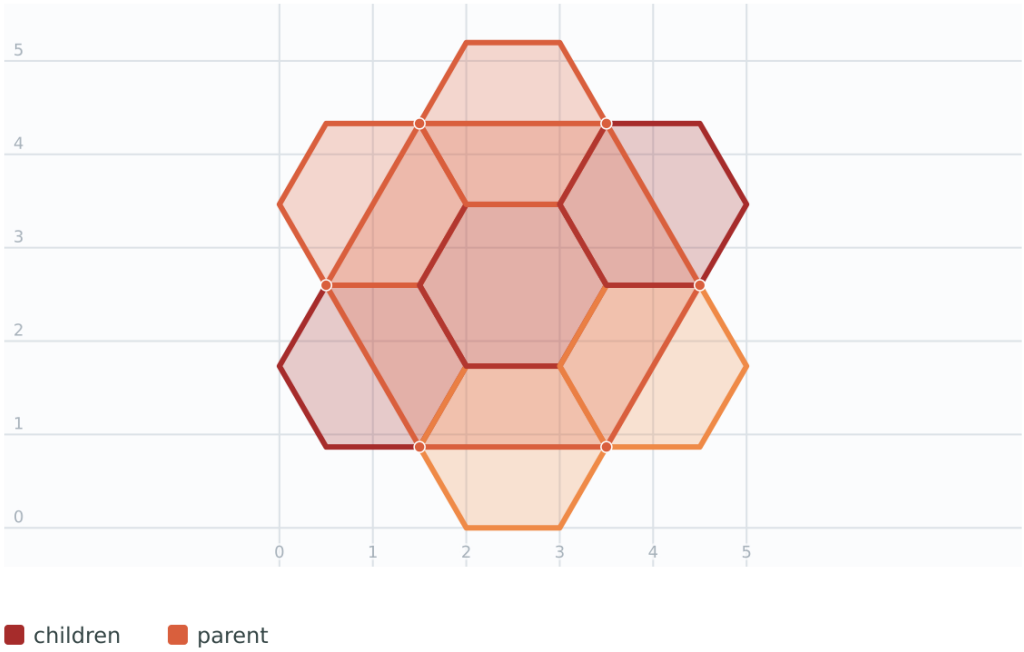


Child tiles that intersect a doubled-size parent hexagon share the tiling origin, but do not nest perfectly inside the parent.

Code

```
WITH parent AS (
    SELECT ST_Hexagon(2, 0, 0) AS geom
), children AS (
    SELECT child.*
    FROM parent
    CROSS JOIN LATERAL ST_HexagonGrid(1, geom) AS child
    WHERE ST_Area(ST_Intersection(child.geom, parent.geom)) > 0
)
SELECT ST_Collect(geom ORDER BY i, j) AS children,
    (SELECT geom FROM parent) AS parent
FROM children;
```

Figure



Counting points in hexagons.

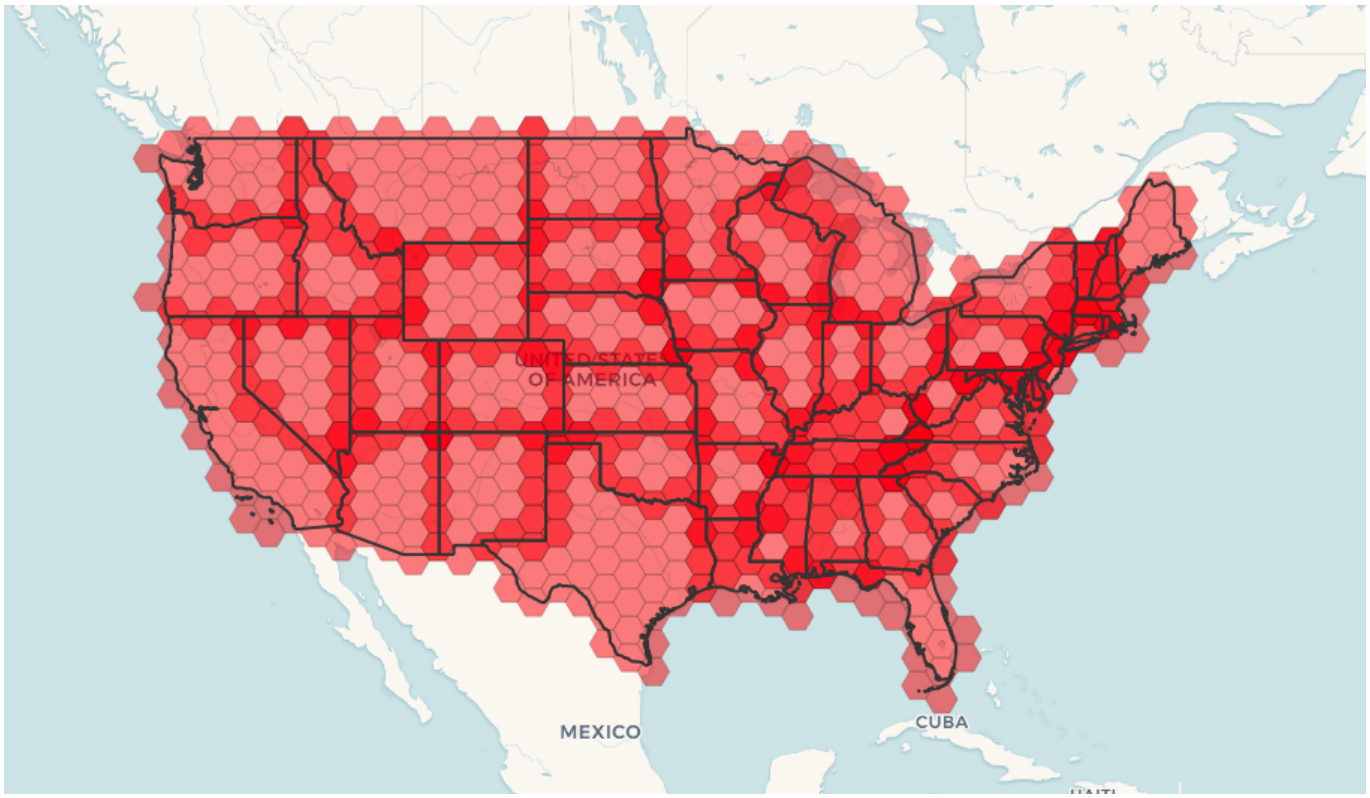
To do a point summary against a hexagonal tiling, generate a hexagon grid using the extent of the points as the bounds, then spatially join to that grid.

Code

```
SELECT COUNT(*), hexes.geom
FROM ST_HexagonGrid(
  10000,
  ST_SetSRID(ST_EstimatedExtent('pointtable', 'geom'), 3857)
) AS hexes
INNER JOIN pointtable AS pts
  ON ST_Intersects(pts.geom, hexes.geom)
GROUP BY hexes.geom;
```

Generating hex coverage of polygons.

If a grid is generated for each polygon boundary and filtered to intersecting cells, adjacent regions receive overlapping hexagons along their shared border. The build-time figure below uses two simple regions in place of an external administrative-boundary table, so the example is self-contained.



Code

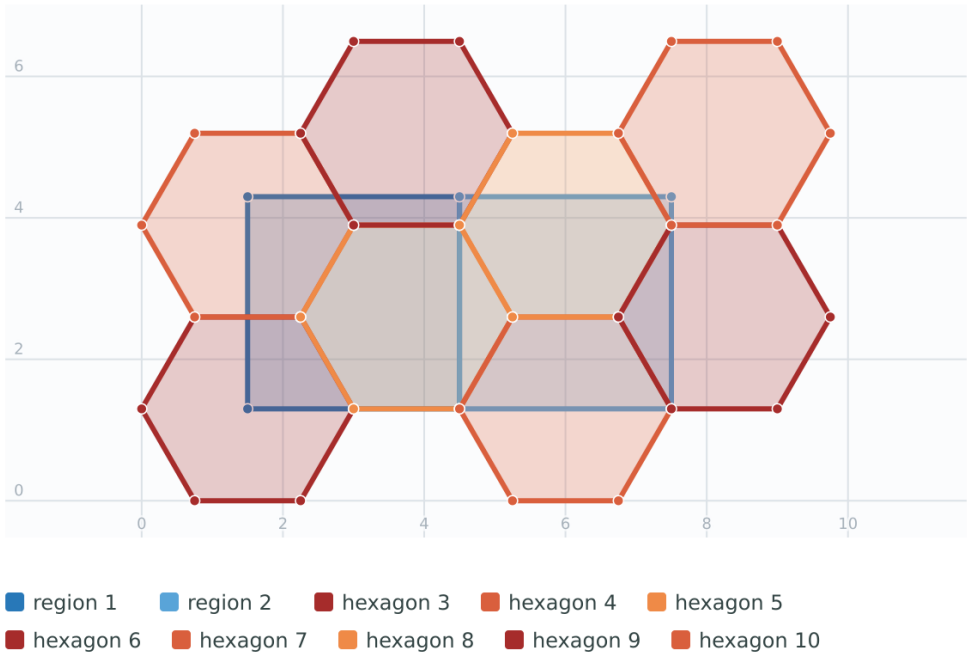
```
WITH admin1(gid, geom) AS (
    VALUES
        (1, ST_MakeEnvelope(0, 0, 3, 3, 3857)),
        (2, ST_MakeEnvelope(3, 0, 6, 3, 3857))
), coverage AS (
    SELECT hex.i, hex.j, hex.geom
    FROM admin1
    CROSS JOIN LATERAL ST_HexagonGrid(1.5, admin1.geom) AS hex
    WHERE ST_Intersects(admin1.geom, hex.geom)
    GROUP BY hex.i, hex.j, hex.geom
), layers AS (
    SELECT 'region' AS kind, gid AS position, geom
    FROM admin1
    UNION ALL
    SELECT
        'hexagon',
        1000 + row_number() OVER (ORDER BY i, j),
        geom
    FROM coverage
)
SELECT
    CASE kind WHEN 'region' THEN geom END AS input_region,
    CASE kind WHEN 'hexagon' THEN geom END AS hexagon
FROM layers
ORDER BY position;
```

Output

```
POLYGON((0 0,0 3,3 3,3 0,0 0)) | null
POLYGON((3 0,3 3,6 3,6 0,3 0)) | null
null | POLYGON((-1.5 0,-0.75 -1.3,0.75 -1.3,1.5 0,0.75 1.3,-0.75 1.3,-1.5 0))
null | POLYGON((-1.5 2.6,-0.75 1.3,0.75 1.3,1.5 2.6,0.75 3.9,-0.75 3.9,-1.5 2.6))
null | POLYGON((0.75 1.3,1.5 0,3 0,3.75 1.3,3 2.6,1.5 2.6,0.75 1.3))
null | POLYGON((0.75 3.9,1.5 2.6,3 2.6,3.75 3.9,3 5.2,1.5 5.2,0.75 3.9))
```

```
null | POLYGON((3 0,3.75 -1.3,5.25 -1.3,6 0,5.25 1.3,3.75 1.3,3 0))
null | POLYGON((3 2.6,3.75 1.3,5.25 1.3,6 2.6,5.25 3.9,3.75 3.9,3 2.6))
null | POLYGON((5.25 1.3,6 0,7.5 0,8.25 1.3,7.5 2.6,6 2.6,5.25 1.3))
null | POLYGON((5.25 3.9,6 2.6,7.5 2.6,8.25 3.9,7.5 5.2,6 5.2,5.25 3.9))
```

Figure



The same pattern applies to a real administrative-boundary table:



Note
The LATERAL keyword is implied for set-returning functions when referring to a prior table in the FROM list. So CROSS JOIN LATERAL, CROSS JOIN, or just plain , are equivalent constructs for this example.

Code

```
SELECT admin1.gid, hex.geom
FROM
    admin1
    CROSS JOIN
    ST_HexagonGrid(100000, admin1.geom) AS hex
WHERE
    adm0_a3 = 'USA'
    AND
    ST_Intersects(admin1.geom, hex.geom)
```

See Also

ST_EstimatedExtent, ST_SetSRID, ST_SquareGrid, ST_TileEnvelope

7.3.15 ST_Hexagon

ST_Hexagon — Returns a single hexagon, using the provided edge size and cell coordinate within the hexagon grid space.

Synopsis

geometry **ST_Hexagon**(float8 size, integer cell_i, integer cell_j, geometry origin);

Description

Uses the same hexagon tiling concept as ST_HexagonGrid, but generates just one hexagon at the desired cell coordinate. Optionally, can adjust origin coordinate of the tiling, the default origin is at 0,0.

Hexagons are generated with no SRID set, so use ST_SetSRID to set the SRID to the one you expect.

Availability: 3.1.0

Examples

Creating a hexagon at the origin.

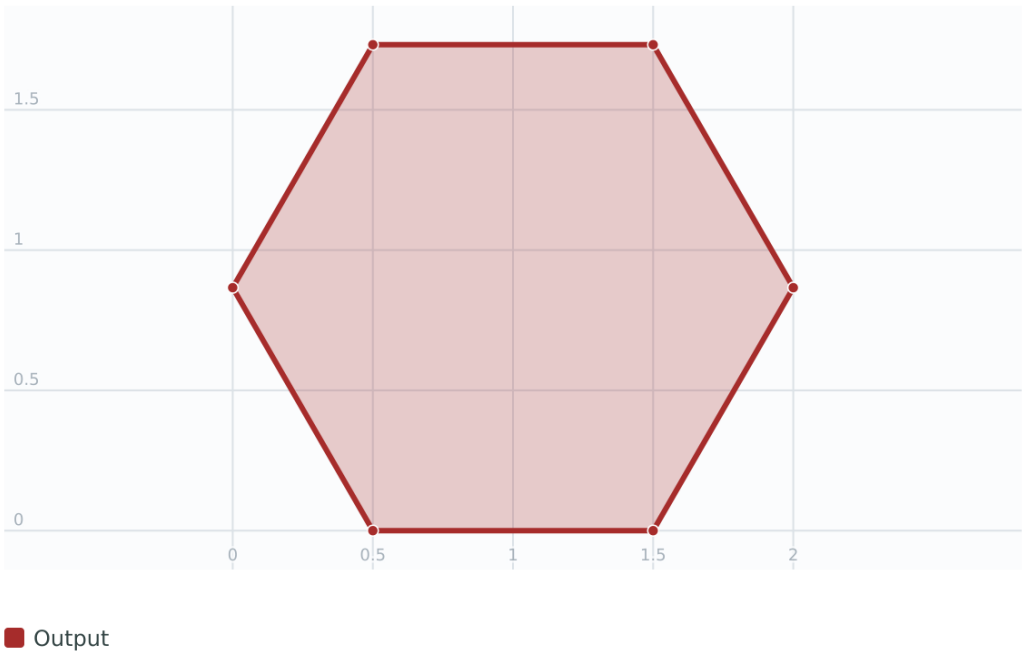
Code

```
SELECT ST_SetSRID(ST_Hexagon(1.0, 0, 0), 3857);
```

Output

```
POLYGON((-1 0,-0.5 -0.8660254037844387,0.5 -0.8660254037844387,1 0,0.5 0.8660254037844387,-0.5 0.8660254037844387,-1 0) ↩
    )
```

Figure



See Also

ST_TileEnvelope, ST_HexagonGrid, ST_Square

7.3.16 ST_SquareGrid

ST_SquareGrid — Returns a set of grid squares and cell indices that completely cover the bounds of the geometry argument.

Synopsis

setof record **ST_SquareGrid**(float8 size, geometry bounds);

Description

Starts with the concept of a square tiling of the plane. For a given planar SRS, and a given edge size, starting at the origin of the SRS, there is one unique square tiling of the plane, Tiling(SRS, Size). This function answers the question: what grids in a given Tiling(SRS, Size) overlap with a given bounds.

The SRS for the output squares is the SRS provided by the bounds geometry.

Doubling or edge size of the square generates a new parent tiling that perfectly fits with the original tiling. Standard web map tilings in mercator are just powers-of-two square grids in the mercator plane.

Availability: 3.1.0

Examples

Generating a 1 degree grid for a country.

The grid will fill the whole bounds of the country, so if you want just squares that touch the country you will have to filter afterwards with ST_Intersects.

Code

```
WITH grid AS (
SELECT (ST_SquareGrid(1, ST_Transform(geom, 4326))).*
FROM admin0 WHERE name = 'Canada'
)
  SELEcT geom
  FROM grid
```

Counting points in squares (using single chopped grid).

To do a point summary against a square tiling, generate a square grid using the extent of the points as the bounds, then spatially join to that grid. Note the estimated extent might be off from actual extent, so be cautious and at very least make sure you've analyzed your table.

Code

```
SELECT COUNT(*), squares.geom
FROM pointtable AS pts
INNER JOIN ST_SquareGrid(
  1000,
  ST_SetSRID(ST_EstimatedExtent('pointtable', 'geom'), 3857)
) AS squares
  ON ST_Intersects(pts.geom, squares.geom)
GROUP BY squares.geom;
```

Counting points in squares using set of grid per point.

This yields the same result as the first example but will be slower for a large number of points

Code

```
SELECT COUNT(*), squares.geom
FROM pointtable AS pts
INNER JOIN ST_SquareGrid(
  1000,
  pts.geom
) AS squares
  ON ST_Intersects(pts.geom, squares.geom)
GROUP BY squares.geom;
```

See Also

ST_TileEnvelope, ST_HexagonGrid , ST_EstimatedExtent , ST_SetSRID

7.3.17 ST_Square

ST_Square — Returns a single square, using the provided edge size and cell coordinate within the square grid space.

Synopsis

geometry **ST_Square**(float8 size, integer cell_i, integer cell_j, geometry origin='POINT(0 0)');

Description

Uses the same square tiling concept as ST_SquareGrid, but generates just one square at the desired cell coordinate. Optionally, can adjust origin coordinate of the tiling, the default origin is at 0,0.

Squares are generated with the SRID of the given origin. Use ST_SetSRID to set the SRID if the given origin has an unknown SRID (as is the case by default).

Availability: 3.1.0

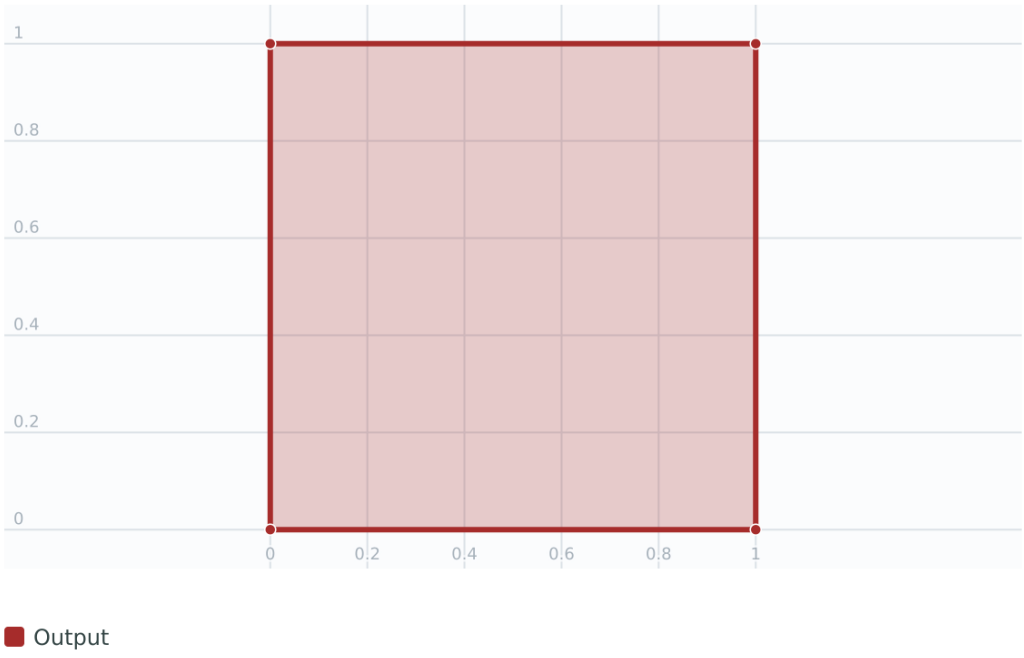
Examples

Creating a square at the origin.

Code

```
SELECT ST_SetSRID(ST_Square(1.0, 0, 0), 3857);
```

Figure



See Also

ST_TileEnvelope, ST_SquareGrid, ST_Hexagon

7.3.18 ST_Letters

ST_Letters — Returns the input letters rendered as geometry with a default start position at the origin and default text height of 100.

Synopsis

geometry **ST_Letters**(text letters, json font);

Description

Uses a built-in font to render out a string as a multipolygon geometry. The default text height is 100.0, the distance from the bottom of a descender to the top of a capital. The default start position places the start of the baseline at the origin. Over-riding the font involves passing in a json map, with a character as the key, and base64 encoded TWKB for the font shape, with the fonts having a height of 1000 units from the bottom of the descenders to the tops of the capitals.

The text is generated at the origin by default, so to reposition and resize the text, first apply the `ST_Scale` function and then apply the `ST_Translate` function.

Availability: 3.3.0

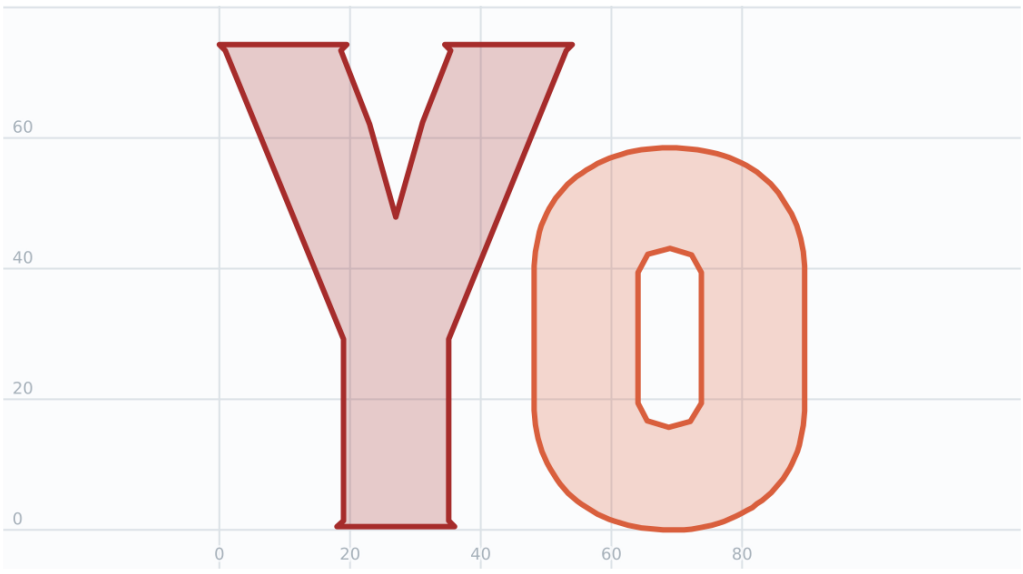
Examples

Generating the word 'Yo'.

Code

```
SELECT ST_Letters('Yo');
```

Figure



Generating the word 'PostGIS'.

Code

```
SELECT ST_Letters('PostGIS') AS letters;
```

Figure

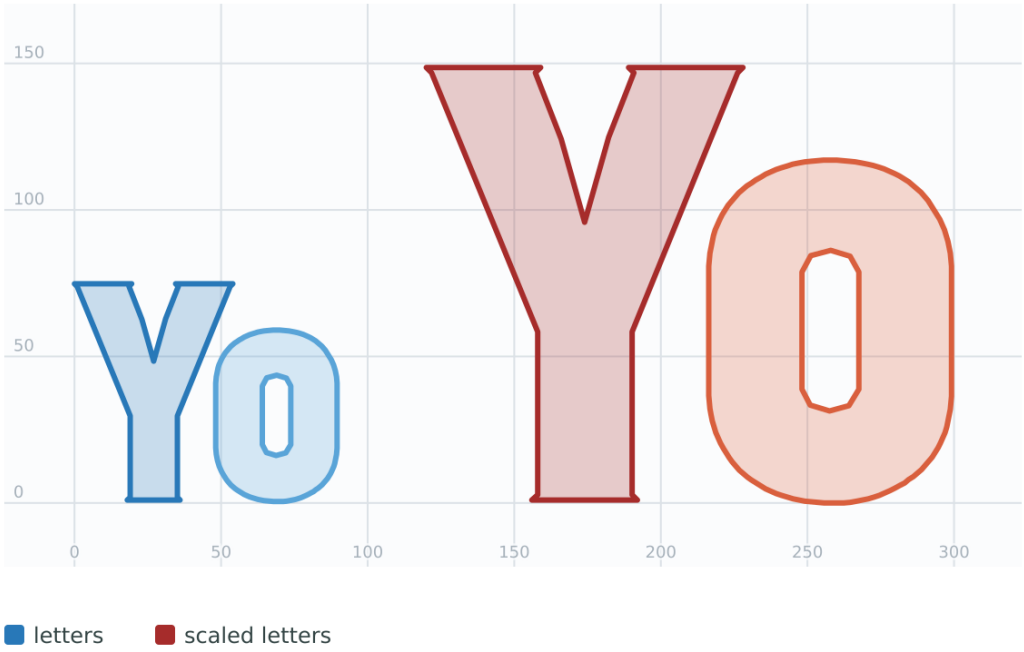


Scaling and moving words.

Code

```
WITH letters AS (
    SELECT ST_Letters('Yo') AS geom
)
SELECT geom AS input_letters,
       ST_Translate(ST_Scale(geom, 2, 2), 120, 0)
           AS scaled_letters
FROM letters;
```

Figure



See Also

ST_AsTWKB, ST_Scale, ST_Translate

7.3.19 ST_MakeNurbsCurve

ST_MakeNurbsCurve — Creates a NURBS (Non-Uniform Rational B-Spline) curve from control points, optional weights, and optional knot vector.

Synopsis

geometry **ST_MakeNurbsCurve**(integer degree, geometry control_points, float8[] weights, float8[] knots);

Description

Creates a NURBS (Non-Uniform Rational B-Spline) curve geometry from the specified parameters. NURBS curves are smooth parametric curves that provide precise control over curve shape through control points, weights, and knot values.

The `degree` parameter specifies the polynomial degree of the curve (must be between 1 and 10). Higher degrees produce smoother curves.

The `control_points` must be a LINESTRING geometry defining the control polygon. The number of control points must be at least `degree` + 1.

The optional `weights` array assigns a weight to each control point. All weights must be positive. If NULL or omitted, uniform weights (1.0) are used, resulting in a non-rational B-spline.

The optional `knots` array defines the parameter space. If NULL or omitted, a uniform clamped knot vector is automatically generated. If provided, the knot vector must have exactly (`npoints` + `degree` + 1) elements and must be non-decreasing.

Availability: 3.7.0

Examples

This example creates a degree-2 NURBS curve with uniform weights.

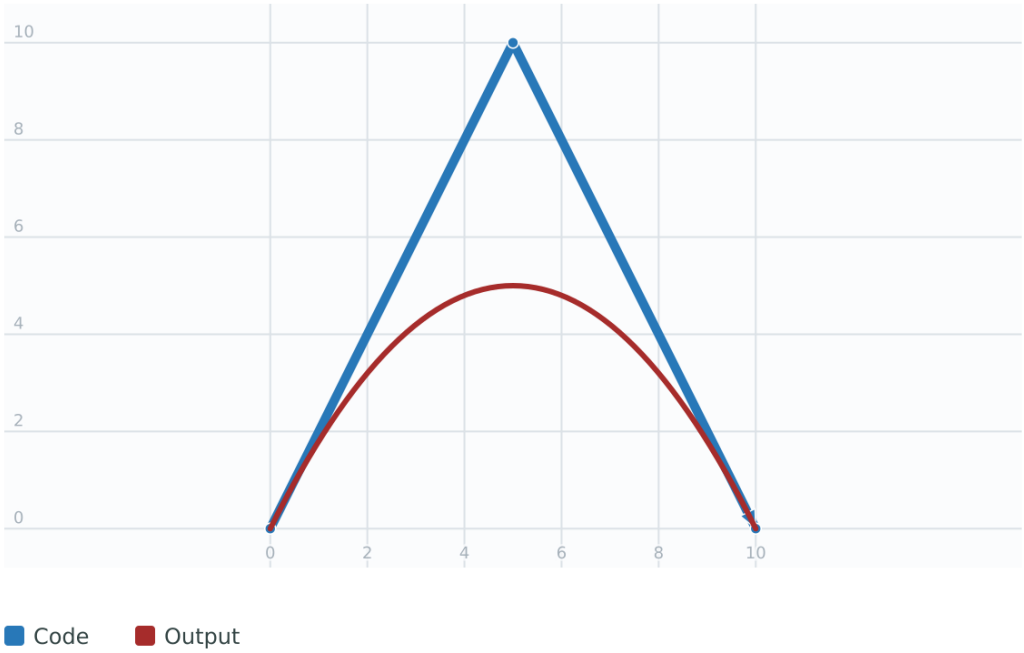
Code

```
SELECT ST_MakeNurbsCurve(
    2,
    'LINESTRING(0 0,5 10,10 0)'::geometry
);
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 10),WEIGHT 1),
    NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



This example creates a rational NURBS curve with custom weights.

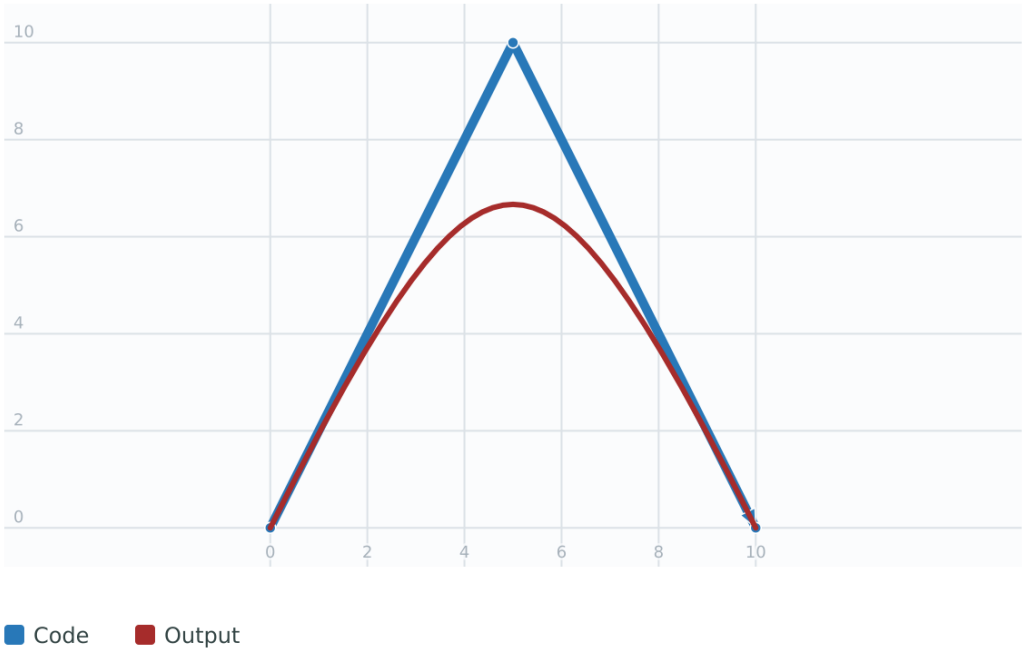
Code

```
SELECT ST_MakeNurbsCurve(
    2,
    'LINESTRING(0 0,5 10,10 0)'::geometry,
    ARRAY[1.0, 2.0, 1.0]
);
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 10),WEIGHT 2),
    NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



This example creates a NURBS curve with a non-uniform knot vector.

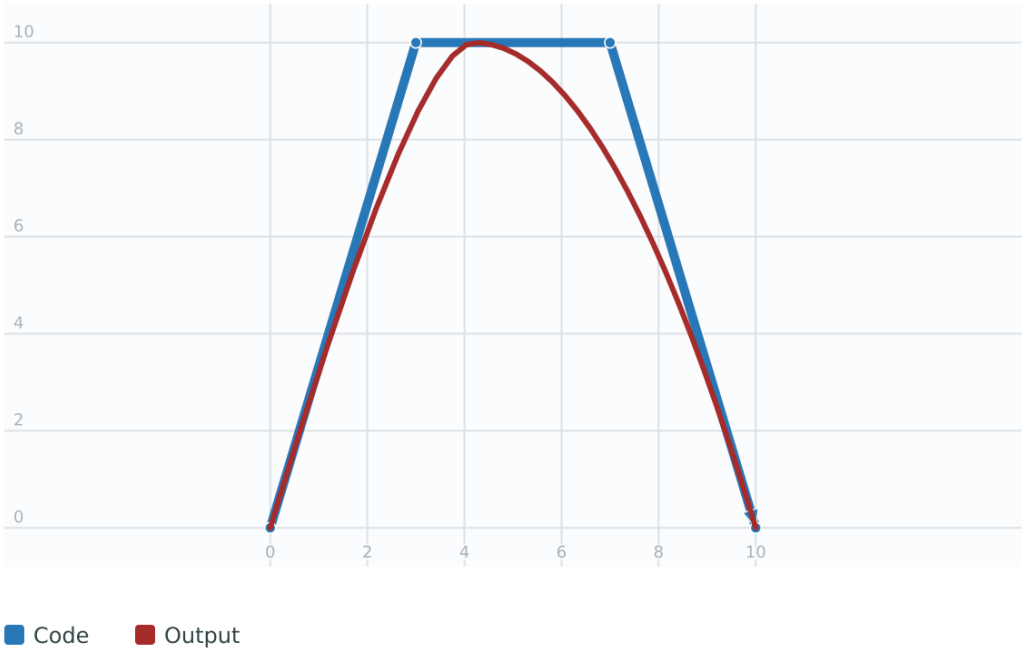
Code

```
SELECT ST_MakeNurbsCurve(
    2,
    'LINESTRING(0 0,3 10,7 10,10 0)'::geometry,
    ARRAY[1.0, 1.0, 1.0, 1.0],
    ARRAY[0.0, 0.0, 0.0, 0.3, 1.0, 1.0, 1.0]
);
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(3 10),WEIGHT 1),  ↩
    NURBSPOINT(WEIGHTEDPOINT(7 10),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(0.3,1), ↩
     KNOT(1,3)))
```

Figure



See Also

ST_NurbsToLineString, ST_Evaluate, ST_ControlPoints

7.4 Geometry Accessors

7.4.1 GeometryType

GeometryType — Returns the type of a geometry as text.

Synopsis

text **GeometryType**(geometry geomA);

Description

Returns the type of the geometry as a string. Eg: 'LINESTRING', 'POLYGON', 'MULTIPOINT', etc.

OGC SPEC s2.1.1.1 - Returns the name of the instantiable subtype of Geometry of which this Geometry instance is a member. The name of the instantiable subtype of Geometry is returned as a string.



Note
This function also indicates if the geometry is measured, by returning a string of the form 'POINTM'.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

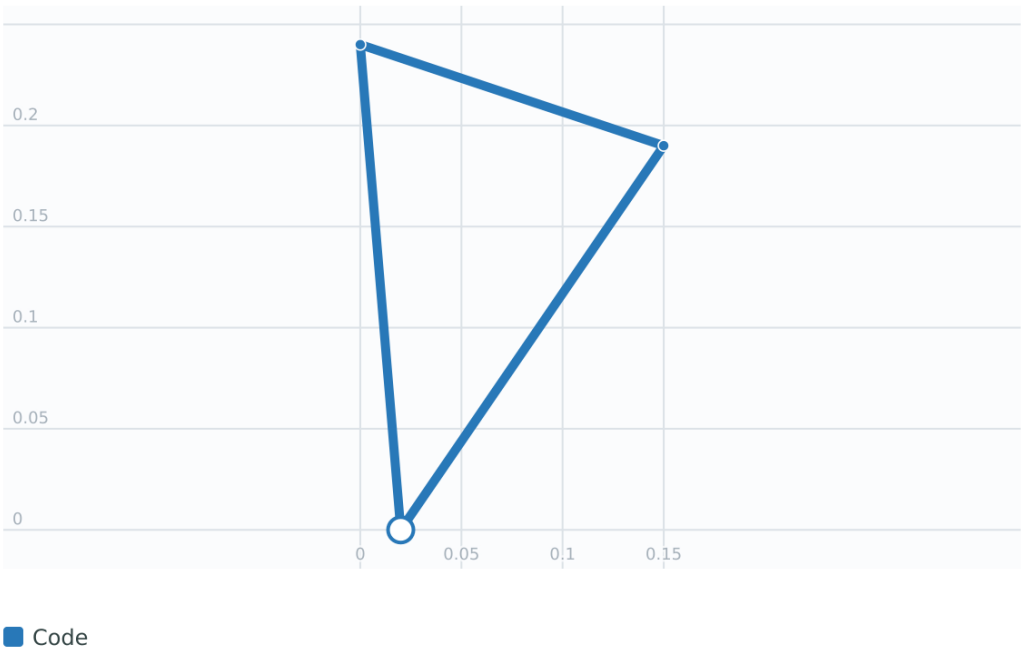
Code

```
SELECT GeometryType('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)'::geometry) ↩
    ;
```

Output

```
LINESTRING
```

Figure



Code

```
SELECT ST_GeometryType('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )');
    POLYHEDRALSURFACE
```

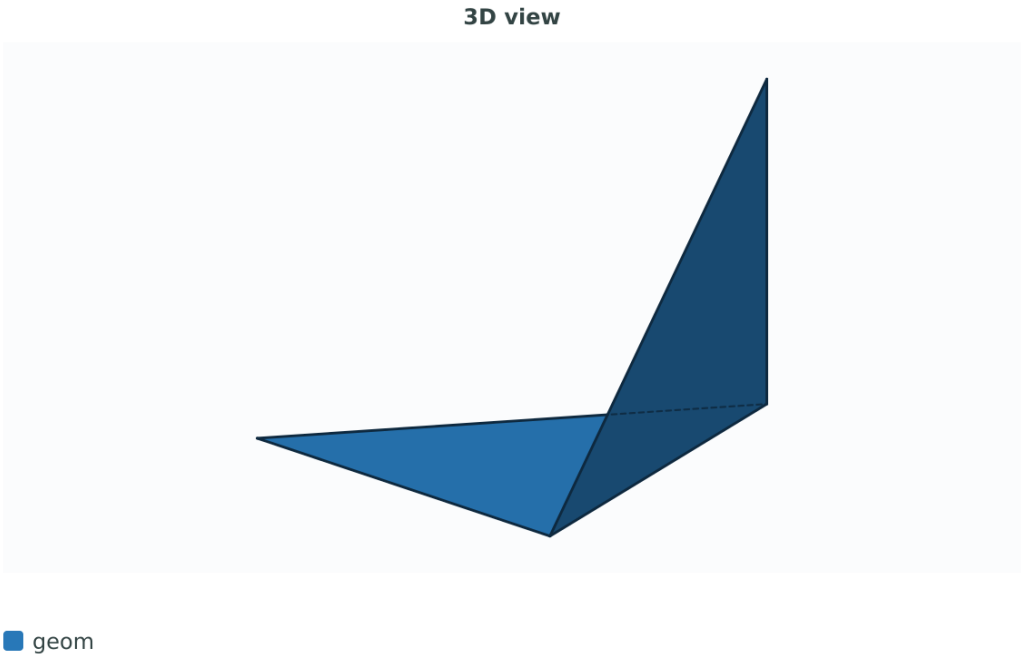
Code

```
SELECT GeometryType(geom) as result
FROM
  (SELECT
     'TIN (((
               0 0 0,
               0 0 1,
               0 1 0,
               0 0 0
           )),((
               0 0 0,
               0 1 0,
               1 1 0,
               0 0 0
           ))
           )'::geometry  AS geom
  ) AS g;
```

Output

```
TIN
```

Figure



See Also

ST_GeometryType

7.4.2 ST_Boundary

ST_Boundary — Returns the boundary of a geometry.

Synopsis

geometry **ST_Boundary**(geometry geomA);

Description

Returns the closure of the combinatorial boundary of this Geometry. The combinatorial boundary is defined as described in section 3.12.3.2 of the OGC SPEC. Because the result of this function is a closure, and hence topologically closed, the resulting boundary can be represented using representational geometry primitives as discussed in the OGC SPEC, section 3.12.2.

Performed by the GEOS module



Note

Prior to 2.0.0, this function throws an exception if used with `GEOMETRYCOLLECTION`. From 2.0.0 up it will return NULL instead (unsupported input).



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. OGC SPEC s2.1.1.1



This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.17



This function supports 3d and will not drop the z-index.

Enhanced: 2.1.0 support for Triangle was introduced

Changed: 3.2.0 support for TIN, does not use geos, does not linearize curves

Examples

LineString with boundary points overlaid.

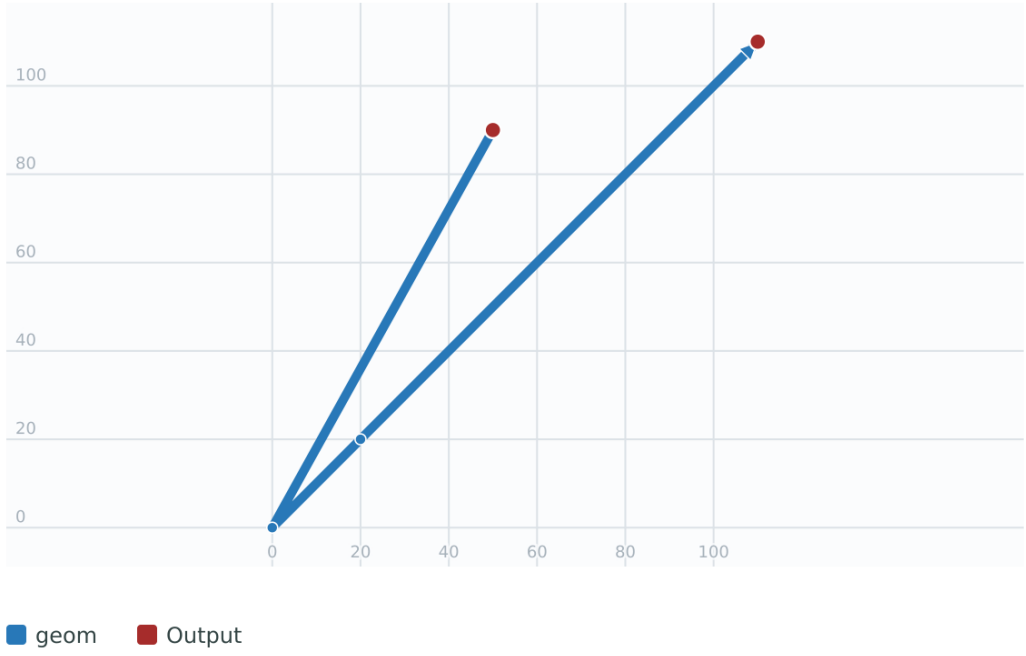
Code

```
SELECT ST_Boundary(geom)
FROM (SELECT 'LINESTRING(100 150,50 60,70 80,160 170)'::geometry As geom) As f;
```

Output

```
MULTIPOINT((100 150),(160 170))
```

Figure



Polygon holes with the boundary MultiLineString overlaid.

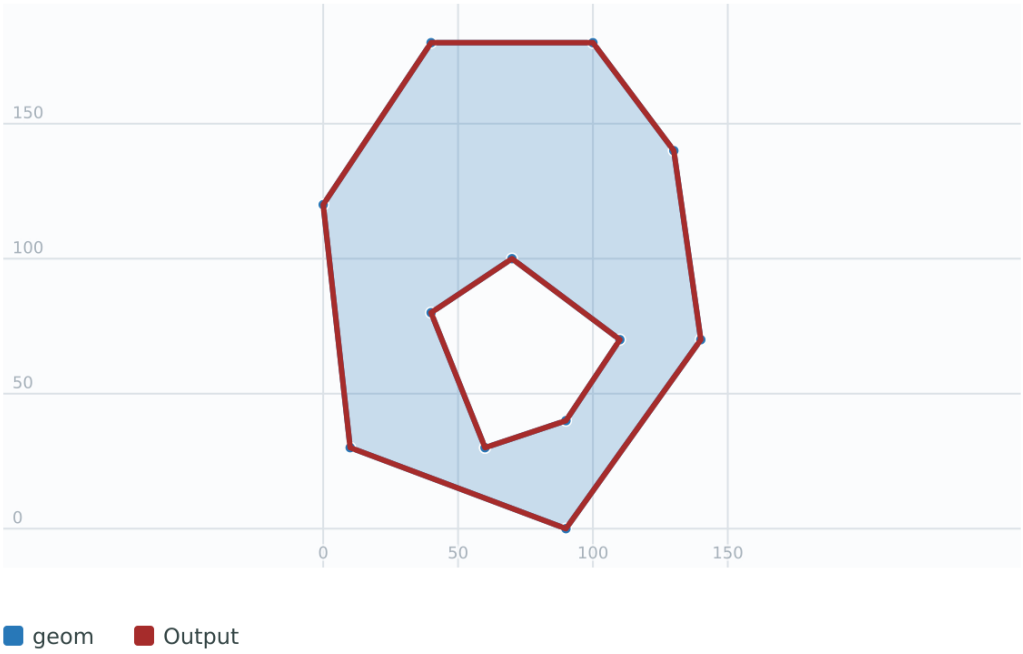
Code

```
SELECT ST_Boundary(geom)
FROM (SELECT
'POLYGON (( 10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130 ),
    ( 70 40,100 50,120 80,80 110,50 90,70 40 ))'::geometry As geom) As f;
```

Output

```
MULTILINESTRING((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130),
        (70 40,100 50,120 80,80 110,50 90,70 40))
```

Figure



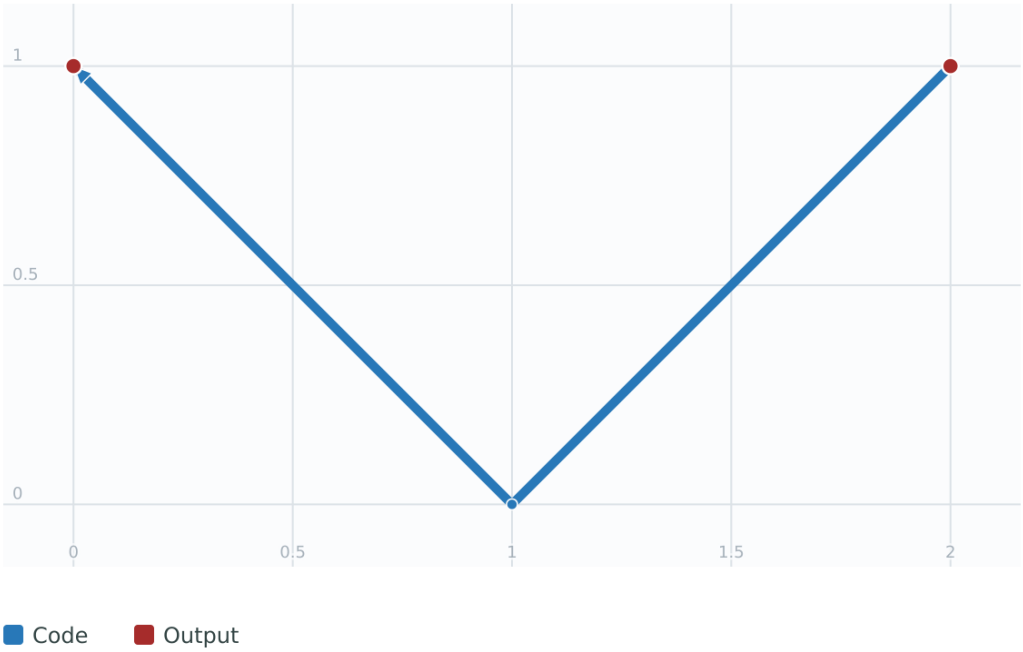
Code

```
SELECT ST_Boundary('LINESTRING(1 1,0 0,-1 1)');
```

Output

```
MULTIPOINT((1 1),(-1 1))
```

Figure



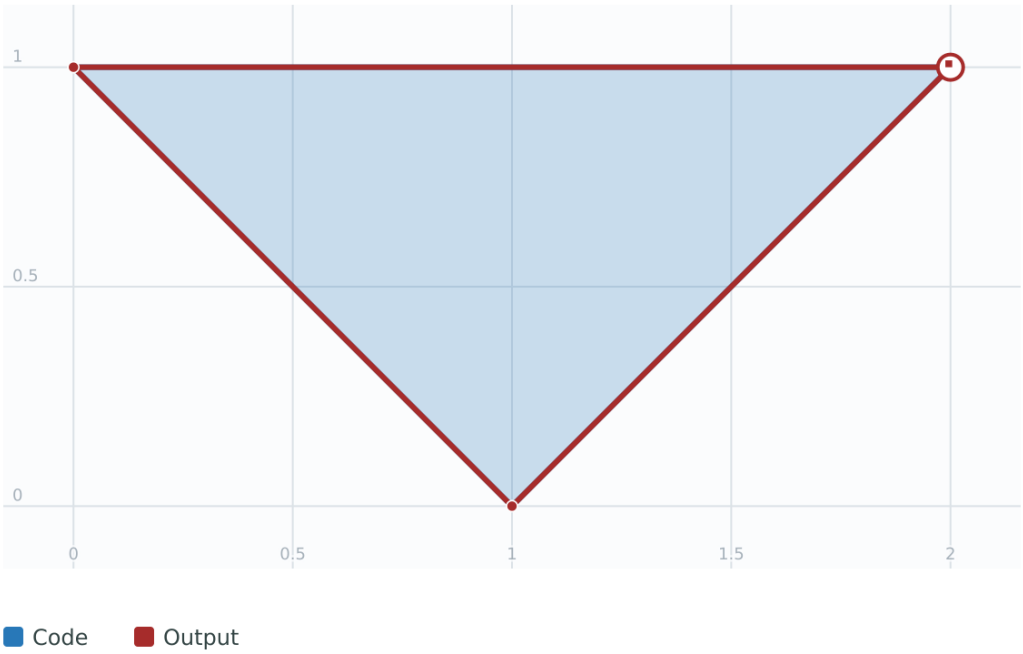
Code

```
SELECT ST_Boundary('POLYGON((1 1,0 0,-1 1,1 1))');
```

Output

```
LINESTRING(1 1,0 0,-1 1,1 1)
```

Figure



This example uses a 3D polygon.

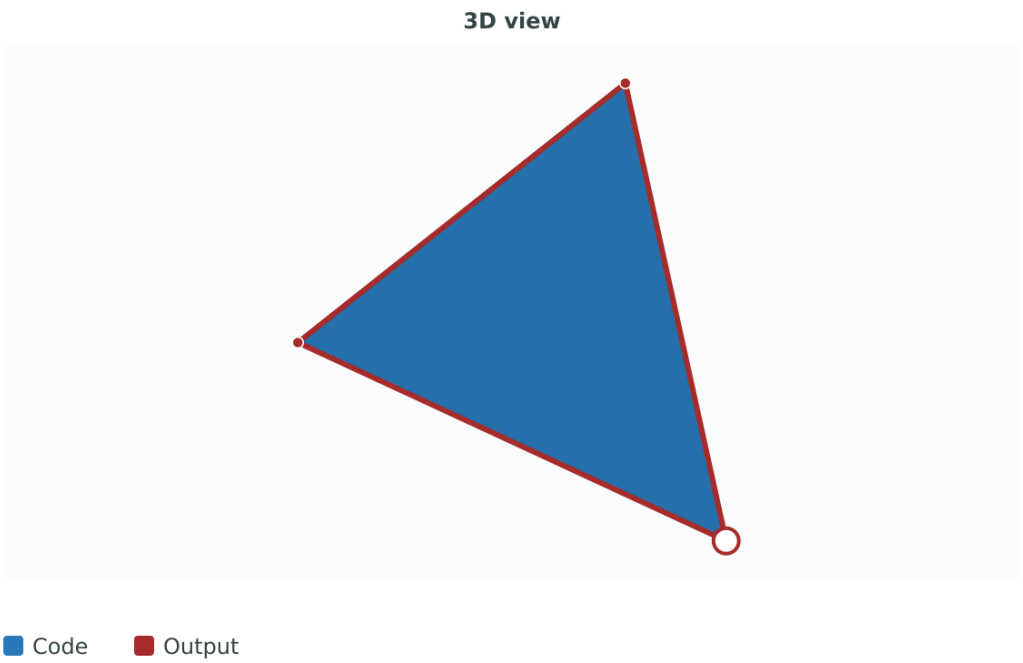
Code

```
SELECT ST_Boundary('POLYGON((1 1 1,0 0 1,-1 1 1,1 1 1))');
```

Output

```
LINESTRING(1 1 1,0 0 1,-1 1 1,1 1 1)
```

Figure



This example uses a 3D MultiLineString.

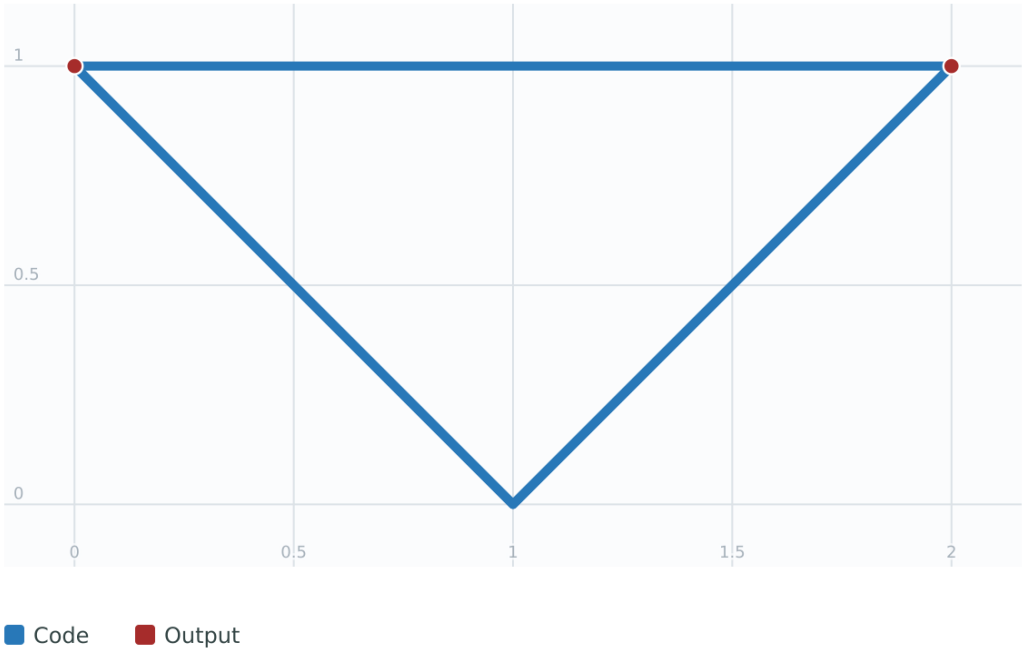
Code

```
SELECT ST_Boundary('MULTILINESTRING((1 1 1,0 0 0.5,-1 1 1),(1 1 0.5,0 0 0.5,-1 1 0.5,1 1 ↩
    0.5) )');
```

Output

```
MULTIPOINT(1 1 1,-1 1 1)
```

Figure



See Also

ST_AsText, ST_ExteriorRing, ST_MakePolygon

7.4.3 ST_BoundingDiagonal

ST_BoundingDiagonal — Returns the diagonal of a geometry's bounding box.

Synopsis

geometry **ST_BoundingDiagonal**(geometry geom, boolean fits=false);

Description

Returns the diagonal of the supplied geometry's bounding box as a LineString. The diagonal is a 2-point LineString with the minimum values of each dimension in its start point and the maximum values in its end point. If the input geometry is empty, the diagonal line is a LINESTRING EMPTY.

The optional `fits` parameter specifies if the best fit is needed. If false, the diagonal of a somewhat larger bounding box can be accepted (which is faster to compute for geometries with many vertices). In either case, the bounding box of the returned diagonal line always covers the input geometry.

The returned geometry retains the SRID and dimensionality (Z and M presence) of the input geometry.



Note
In degenerate cases (i.e. a single vertex in input) the returned linestring will be formally invalid (no interior). The result is still topologically valid.

Availability: 2.2.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Examples

This example shows a polygon together with the diagonal of its bounding box.

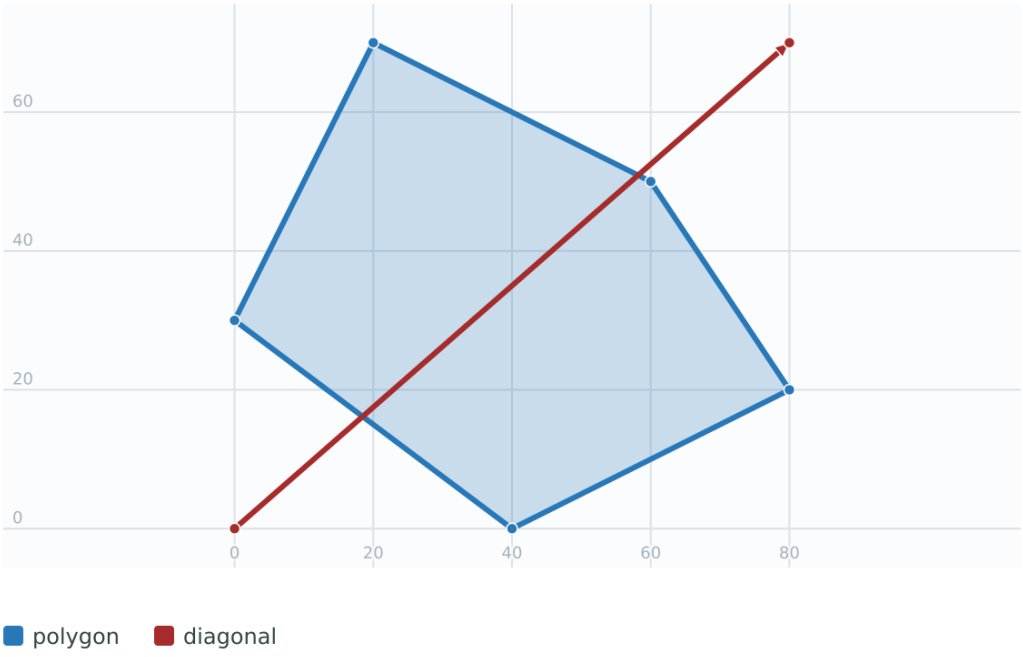
Code

```
WITH input AS (
  SELECT 'POLYGON((10 20,30 60,70 40,90 10,50 -10,10 20))'::geometry AS geom
)
SELECT geom AS input_polygon,
       ST_BoundingDiagonal(geom) AS diagonal
FROM input;
```

Output

```
POLYGON((10 20,30 60,70 40,90 10,50 -10,10 20)) | LINESTRING(10 -10,90 60)
```

Figure



This example returns the minimum X coordinate of a buffer around a point.

Code

```
SELECT ST_X(ST_StartPoint(ST_BoundingDiagonal(
  ST_Buffer(ST_Point(0, 0), 10)
)));
```

Output

```
-10
```

See Also

ST_StartPoint, ST_EndPoint, ST_X, ST_Y, ST_Z, ST_M, ST_Envelope

7.4.4 ST_CoordDim

ST_CoordDim — Return the coordinate dimension of a geometry.

Synopsis

integer **ST_CoordDim**(geometry geomA);

Description

Return the coordinate dimension of the ST_Geometry value.

This is the MM compliant alias name for ST_NDims

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.3

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Code

```
SELECT ST_CoordDim('CIRCULARSTRING(1 2 3,1 3 4,5 6 7,8 9 10,11 12 13)');
```

Output

```
3
```

Code

```
SELECT ST_CoordDim(ST_Point(1, 2));
```

Output

```
2
```

See Also

ST_NDims

7.4.5 ST_Dimension

ST_Dimension — Returns the topological dimension of a geometry.

Synopsis

integer **ST_Dimension**(geometry g);

Description

Return the topological dimension of this Geometry object, which must be less than or equal to the coordinate dimension. OGC SPEC s2.1.1.1 - returns 0 for `POINT`, 1 for `LINESTRING`, 2 for `POLYGON`, and the largest dimension of the components of a `GEOMETRYCOLLECTION`. If the dimension is unknown (e.g. for an empty `GEOMETRYCOLLECTION`) 0 is returned.

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.2



Note
A `TIN`, including a closed TIN surface, is a surface geometry and has dimension 2. Use CG_MakeSolid to make the solid representation explicit; the resulting 3D solid has dimension 3. Solid geometry semantics, such as volume, are handled separately by the SQL/MM 3D solid functions (SQL-MM IEC 13249-3: 9.1).

Enhanced: 2.0.0 support for Polyhedral surfaces and TINs was introduced. No longer throws an exception if given empty geometry.



Note
Prior to 2.0.0, this function throws an exception if used with empty geometry.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

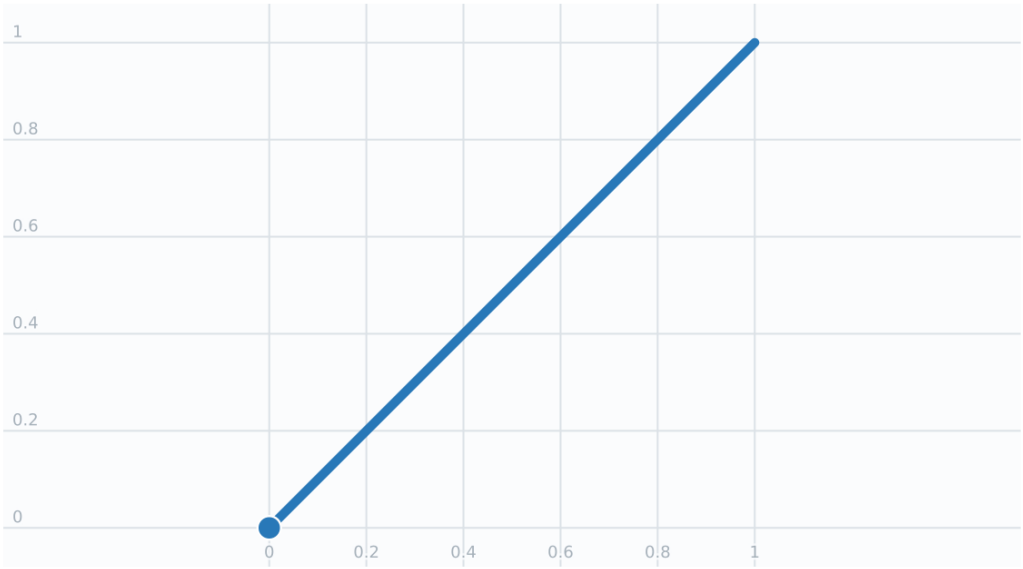
Code

```
SELECT ST_Dimension('GEOMETRYCOLLECTION(LINESTRING(1 1,0 0), POINT(0 0))');
```

Output

```
1
```

Figure



Code

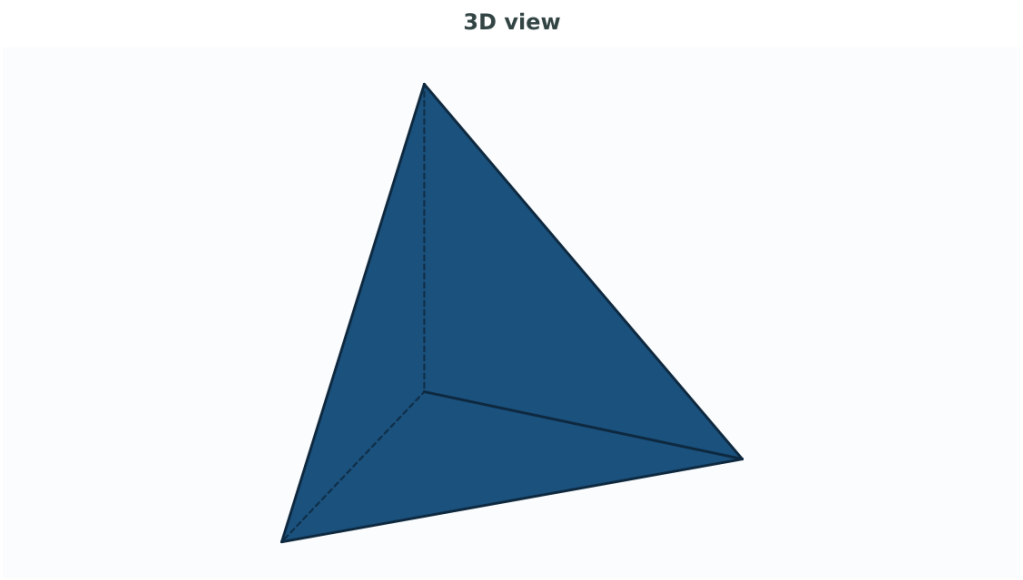
Code

```
SELECT ST_Dimension('TIN Z (((0 0 0,0 0 1,0 1 0,0 0 0)),((0 0 0,0 1 0,1 0 0,0 0 0)),((0 0 ↩
    0,1 0 0,0 0 1,0 0 0)),((1 0 0,0 1 0,0 0 1,1 0 0)))');
```

Output

```
2
```

Figure



Code

See Also

ST_NDims, CG_MakeSolid, CG_Volume

7.4.6 ST_Dump

ST_Dump — Returns a set of `geometry_dump` rows for the components of a geometry.

Synopsis

geometry_dump[] **ST_Dump**(geometry g1);

Description

A set-returning function (SRF) that extracts the components of a geometry. It returns a set of geometry_dump rows, each containing a geometry (*`geom`* field) and an array of integers (*`path`* field).

For an atomic geometry type (POINT,LINESTRING,POLYGON) a single record is returned with an empty *`path`* array and the input geometry as *`geom`*. For a collection or multi-geometry a record is returned for each of the collection components, and the *`path`* denotes the position of the component inside the collection.

ST_Dump is useful for expanding geometries. It is the inverse of a ST_Collect / GROUP BY, in that it creates new rows. For example it can be use to expand MULTIPOLYGONS into POLYGONS.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Availability: PostGIS 1.0.0RC1. Requires PostgreSQL 7.3 or higher.



Note
Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

 This function supports 3d and will not drop the z-index.

Examples

Dump the components of geometries stored in a table.

Code

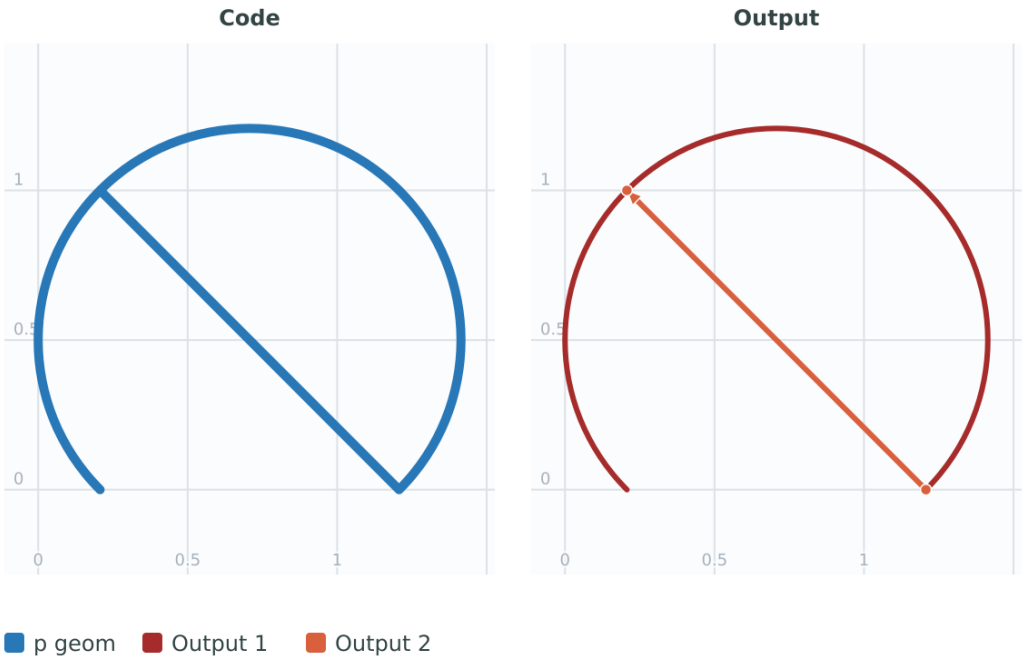
```
SELECT sometable.field1,
       (ST_Dump(sometable.geom)).geom AS geom
FROM sometable;
```

Break a CompoundCurve into its constituent LineStrings and CircularStrings.

Code

```
SELECT a.geom, ST_HasArc(a.geom)
  FROM ( SELECT (ST_Dump(p_geom)).geom AS geom
         FROM (SELECT 'COMPOUNDCURVE(CIRCULARSTRING(0 0,1 1,1 0),(1 0,0 1))'::geometry AS
    p_geom) AS b
        ) AS a;
```

Figure

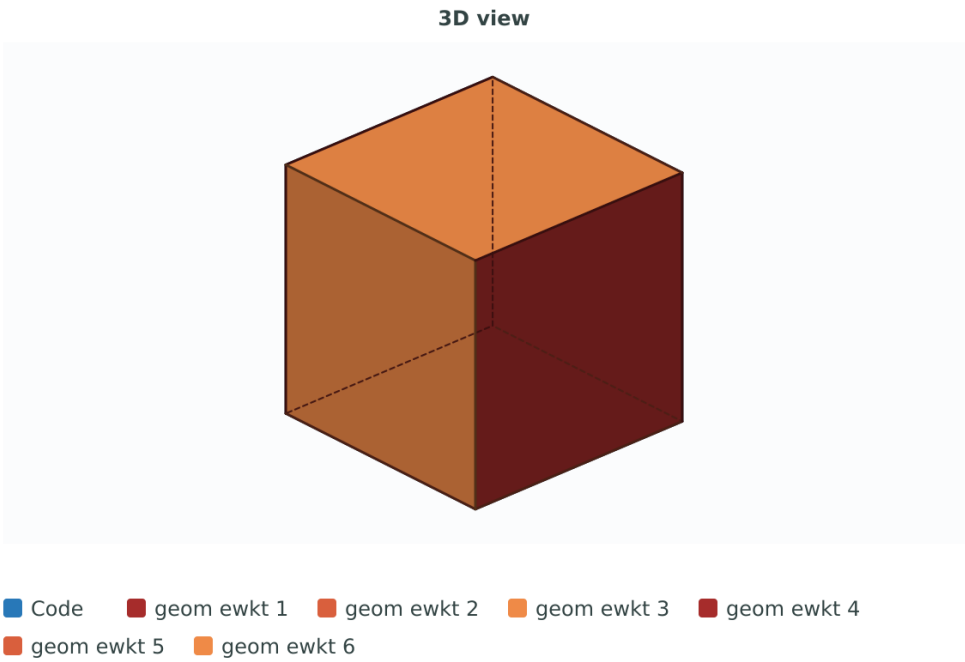


This example breaks a PolyhedralSurface into its faces.

Code

```
SELECT (a.p_geom).path[1] As path,(a.p_geom).geom As geom_ewkt
  FROM (SELECT ST_Dump('POLYHEDRALSURFACE(
((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),((1 1 0,1 1 1,1 0 1,1 0
    0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)' ) AS p_geom )  AS a;
```

Figure

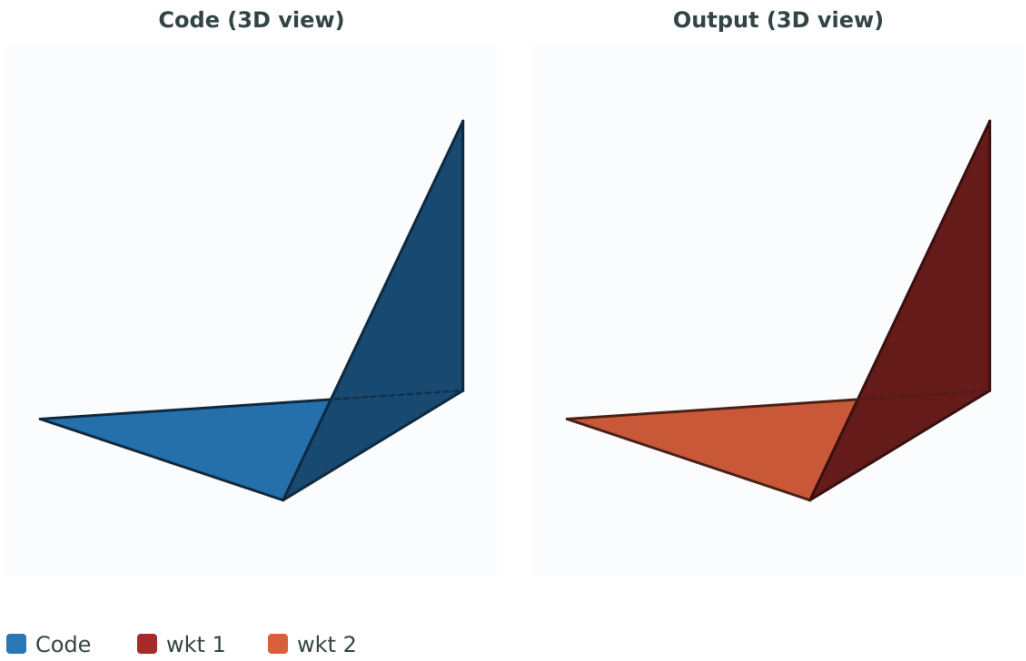


This example uses a TIN geometry.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_Dump('TIN (((
                 0 0 0,
                 0 0 1,
                 0 1 0,
                 0 0 0
             )),((
                 0 0 0,
                 0 1 0,
                 1 1 0,
                 0 0 0
             ))
             )' ) AS gdump
    ) AS g;
```

Figure



See Also

geometry_dump, Section 13.6, ST_Collect, ST_GeometryN

7.4.7 ST_DumpPoints

ST_DumpPoints — Returns a set of `geometry_dump` rows for the coordinates in a geometry.

Synopsis

geometry_dump[] **ST_DumpPoints**(geometry geom);

Description

A set-returning function (SRF) that extracts the coordinates (vertices) of a geometry. It returns a set of geometry_dump rows, each containing a geometry (*geom* field) and an array of integers (*path* field).

- the *geom* field `POINT`s represent the coordinates of the supplied geometry.
- the *path* field (an `integer[]`) is an index enumerating the coordinate positions in the elements of the supplied geometry. The indices are 1-based. For example, for a `LINESTRING` the paths are `{i}` where `i` is the `nth` coordinate in the `LINESTRING`. For a `POLYGON` the paths are `{i,j}` where `i` is the ring number (1 is outer; inner rings follow) and `j` is the coordinate position in the ring.

To obtain a single geometry containing the coordinates use ST_Points.

Enhanced: 2.1.0 Faster speed. Reimplemented as native-C.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Availability: 1.5.0

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

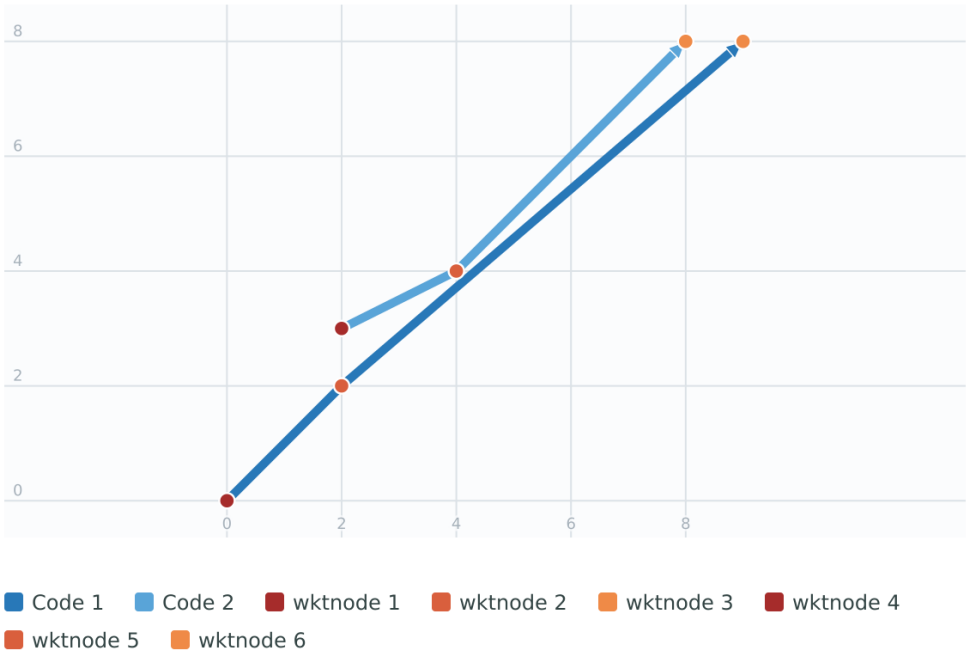
Examples

Explode a table of LineStrings into their vertices.

Code

```
SELECT edge_id, (dp).path[1] As index,(dp).geom As wktnode
FROM (SELECT 1 As edge_id,
    ST_DumpPoints(ST_GeomFromText('LINESTRING(1 2,3 4,10 10)')) AS dp
     UNION ALL
     SELECT 2 As edge_id,
    ST_DumpPoints(ST_GeomFromText('LINESTRING(3 5,5 6,9 10)')) AS dp
   ) As foo;
```

Figure



Dump every vertex from a mixed GeometryCollection.

Code

```
SELECT path,geom
FROM (
  SELECT (ST_DumpPoints(g.geom)).*
  FROM
    (SELECT
       'GEOMETRYCOLLECTION(
          POINT ( 0 1 ),
          LINESTRING ( 0 3,3 4 ),
```

```
          POLYGON (( 2 0,2 3,0 2,2 0 )),
          POLYGON (( 3 0,3 3,6 3,6 0,3 0 ),
                   ( 5 1,4 2,5 2,5 1 )),
          MULTIPOLYGON (
                  (( 0 5,0 8,4 8,4 5,0 5 ),
                   ( 1 6,3 6,2 7,1 6 )),
                  (( 5 4,5 8,6 7,5 4 ))
          )
        )'::geometry AS geom
    ) AS g
  ) j;
```

Output

```
   path    | st_astext
-----------+------------
 {1,1}     | POINT(0 1)
 {2,1}     | POINT(0 3)
 {2,2}     | POINT(3 4)
 {3,1,1}   | POINT(2 0)
 {3,1,2}   | POINT(2 3)
 {3,1,3}   | POINT(0 2)
 {3,1,4}   | POINT(2 0)
 {4,1,1}   | POINT(3 0)
 {4,1,2}   | POINT(3 3)
 {4,1,3}   | POINT(6 3)
 {4,1,4}   | POINT(6 0)
 {4,1,5}   | POINT(3 0)
 {4,2,1}   | POINT(5 1)
 {4,2,2}   | POINT(4 2)
 {4,2,3}   | POINT(5 2)
 {4,2,4}   | POINT(5 1)
 {5,1,1,1} | POINT(0 5)
 {5,1,1,2} | POINT(0 8)
 {5,1,1,3} | POINT(4 8)
 {5,1,1,4} | POINT(4 5)
 {5,1,1,5} | POINT(0 5)
 {5,1,2,1} | POINT(1 6)
 {5,1,2,2} | POINT(3 6)
 {5,1,2,3} | POINT(2 7)
 {5,1,2,4} | POINT(1 6)
 {5,2,1,1} | POINT(5 4)
 {5,2,1,2} | POINT(5 8)
 {5,2,1,3} | POINT(6 7)
 {5,2,1,4} | POINT(5 4)
(29 rows)
```

Collect the dumped vertices to visualize them over the source geometry.

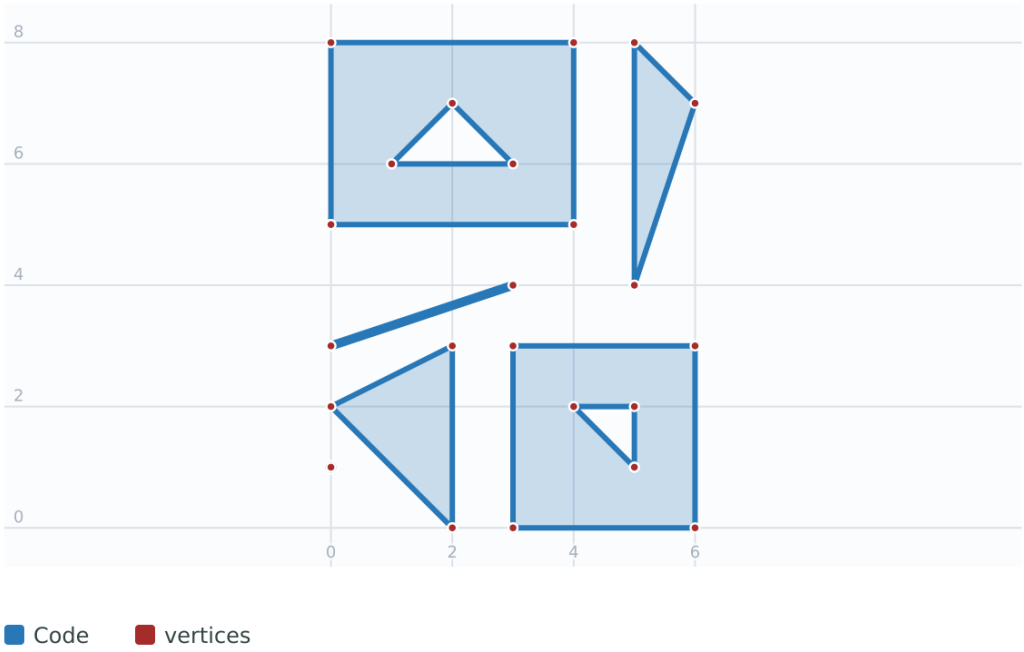
Code

```
WITH source(geom) AS (VALUES (
  'GEOMETRYCOLLECTION(
     POINT (0 1),
     LINESTRING (0 3,3 4),
     POLYGON ((2 0,2 3,0 2,2 0)),
     POLYGON ((3 0,3 3,6 3,6 0,3 0),(5 1,4 2,5 2,5 1)),
     MULTIPOLYGON (((0 5,0 8,4 8,4 5,0 5),(1 6,3 6,2 7,1 6)),
                   ((5 4,5 8,6 7,5 4))))'::geometry
))
SELECT ST_Collect((dp).geom ORDER BY (dp).path) AS vertices
FROM source
CROSS JOIN LATERAL ST_DumpPoints(source.geom) AS dp;
```

Output

```
MULTIPOINT((0 1),(0 3),(3 4),(2 0),(2 3),(0 2),(2 0),(3 0),(3 3),(6 3),(6 0),(3 0),(5 1),(4 2),(5 2),(5 1),(0 5),(0 8) ↩
    ,(4 8),(4 5),(0 5),(1 6),(3 6),(2 7),(1 6),(5 4),(5 8),(6 7),(5 4))
```

Figure



This example uses a cube represented as a PolyhedralSurface.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_DumpPoints('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )' ) AS gdump
    ) AS g;
```

Output

```
  path   |     wkt
---------+-------------
 {1,1,1} | POINT(0 0 0)
 {1,1,2} | POINT(0 0 1)
 {1,1,3} | POINT(0 1 1)
 {1,1,4} | POINT(0 1 0)
 {1,1,5} | POINT(0 0 0)
 {2,1,1} | POINT(0 0 0)
 {2,1,2} | POINT(0 1 0)
 {2,1,3} | POINT(1 1 0)
 {2,1,4} | POINT(1 0 0)
 {2,1,5} | POINT(0 0 0)
 {3,1,1} | POINT(0 0 0)
 {3,1,2} | POINT(1 0 0)
 {3,1,3} | POINT(1 0 1)
 {3,1,4} | POINT(0 0 1)
 {3,1,5} | POINT(0 0 0)
 {4,1,1} | POINT(1 1 0)
 {4,1,2} | POINT(1 1 1)
 {4,1,3} | POINT(1 0 1)
 {4,1,4} | POINT(1 0 0)
 {4,1,5} | POINT(1 1 0)
 {5,1,1} | POINT(0 1 0)
 {5,1,2} | POINT(0 1 1)
 {5,1,3} | POINT(1 1 1)
 {5,1,4} | POINT(1 1 0)
 {5,1,5} | POINT(0 1 0)
 {6,1,1} | POINT(0 0 1)
 {6,1,2} | POINT(1 0 1)
```

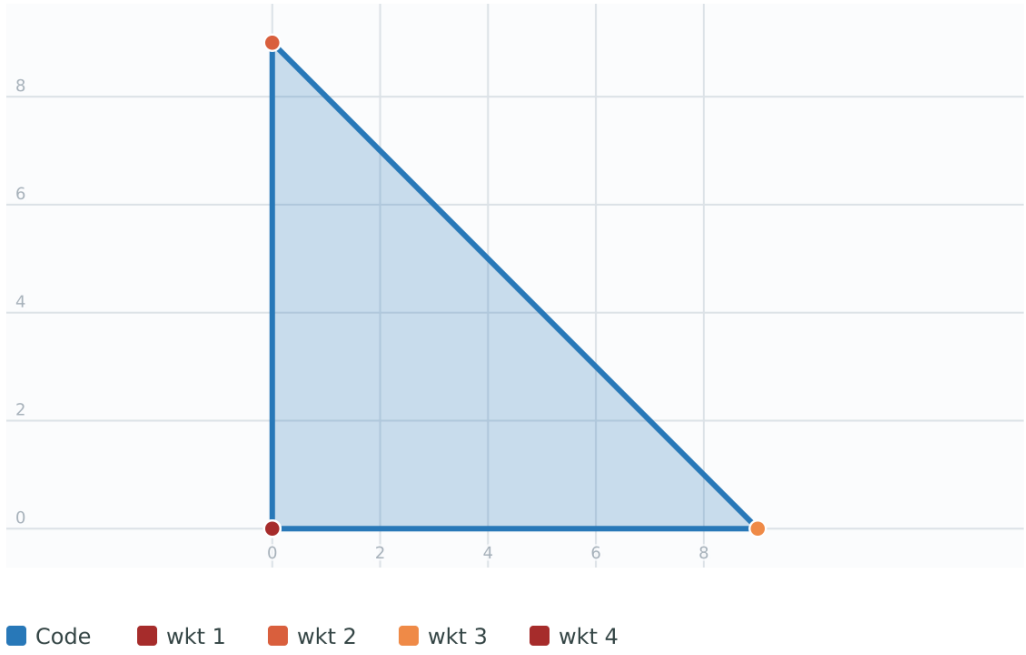
```
 {6,1,3} | POINT(1 1 1)
 {6,1,4} | POINT(0 1 1)
 {6,1,5} | POINT(0 0 1)
(30 rows)
```

This example uses a Triangle geometry.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_DumpPoints('TRIANGLE ((
                  0 0,
                  0 9,
                  9 0,
                  0 0
              ))' ) AS gdump
    ) AS g;
```

Figure



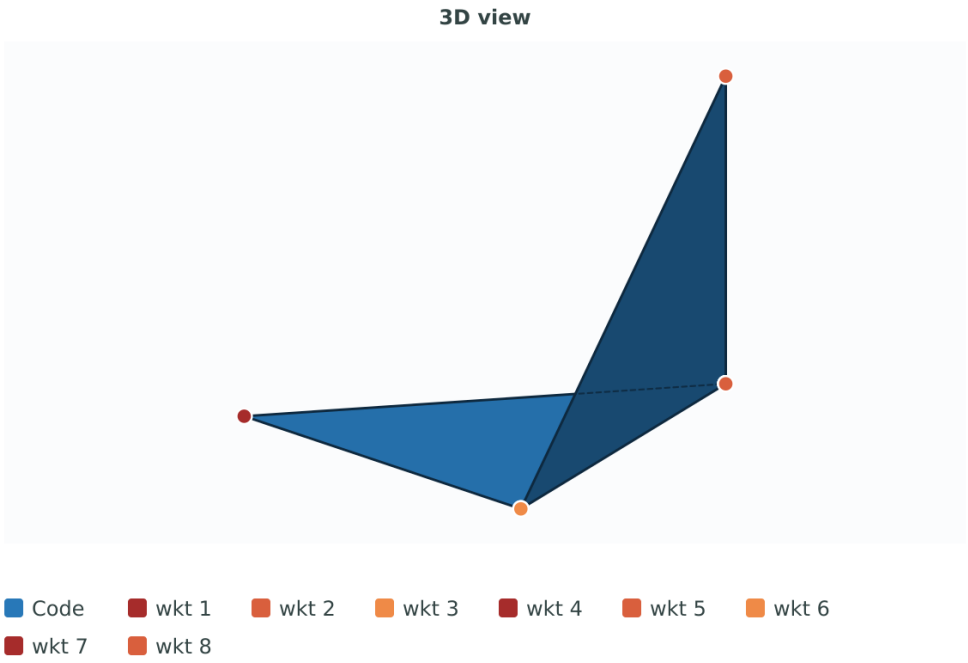
This example uses a TIN geometry.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_DumpPoints('TIN (((
                  0 0 0,
                  0 0 1,
                  0 1 0,
                  0 0 0
              )),((
                  0 0 0,
                  0 1 0,
                  1 1 0,
                  0 0 0
```

```
            ))
            )' ) AS gdump
    ) AS g;
```

Figure



See Also

geometry_dump, Section 13.6, ST_Dump, ST_DumpRings, ST_Points

7.4.8 ST_DumpSegments

ST_DumpSegments — Returns a set of `geometry_dump` rows for the segments in a geometry.

Synopsis

geometry_dump[] **ST_DumpSegments**(geometry geom);

Description

A set-returning function (SRF) that extracts the segments of a geometry. It returns a set of geometry_dump rows, each containing a geometry (*geom* field) and an array of integers (*path* field).

- the *geom* field `LINESTRING`s represent the linear segments of the supplied geometry, while the `CIRCULARSTRING`s represent the arc segments.
- the *path* field (an `integer[]`) is an index enumerating the segment start point positions in the elements of the supplied geometry. The indices are 1-based. For example, for a `LINESTRING` the paths are `{i}` where `i` is the `nth` segment start point in the `LINESTRING`. For a `POLYGON` the paths are `{i,j}` where `i` is the ring number (1 is outer; inner rings follow) and `j` is the segment start point position in the ring.

Availability: 3.2.0

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

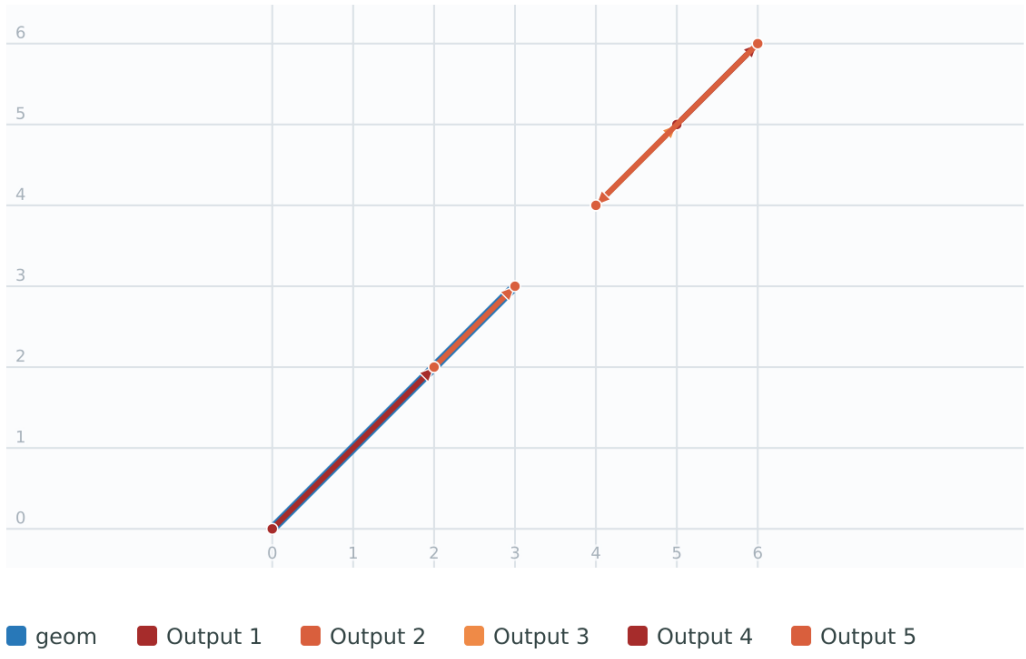
This function supports 3d and will not drop the z-index.

Standard Geometry Examples

Code

```
SELECT path,geom
FROM (
    SELECT (ST_DumpSegments(g.geom)).*
    FROM (SELECT 'GEOMETRYCOLLECTION(
    LINESTRING(1 1,3 3,4 4),
    POLYGON((5 5,6 6,7 7,5 5))
)'::geometry AS geom
        ) AS g
) j;
```

Figure



TIN and Triangle Examples

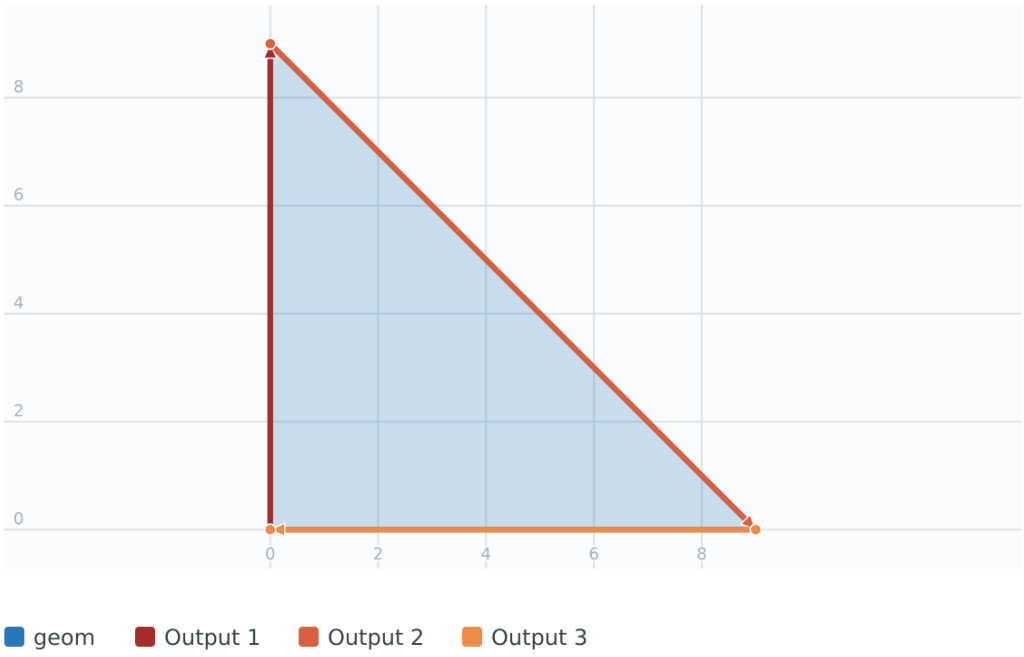
This example uses a Triangle geometry.

Code

```
SELECT path,geom
FROM (
    SELECT (ST_DumpSegments(g.geom)).*
    FROM (SELECT 'TRIANGLE((
        0 0,
        0 9,
        9 0,
```

```
        0 0
    ))'::geometry AS geom
        ) AS g
) j;
```

Figure

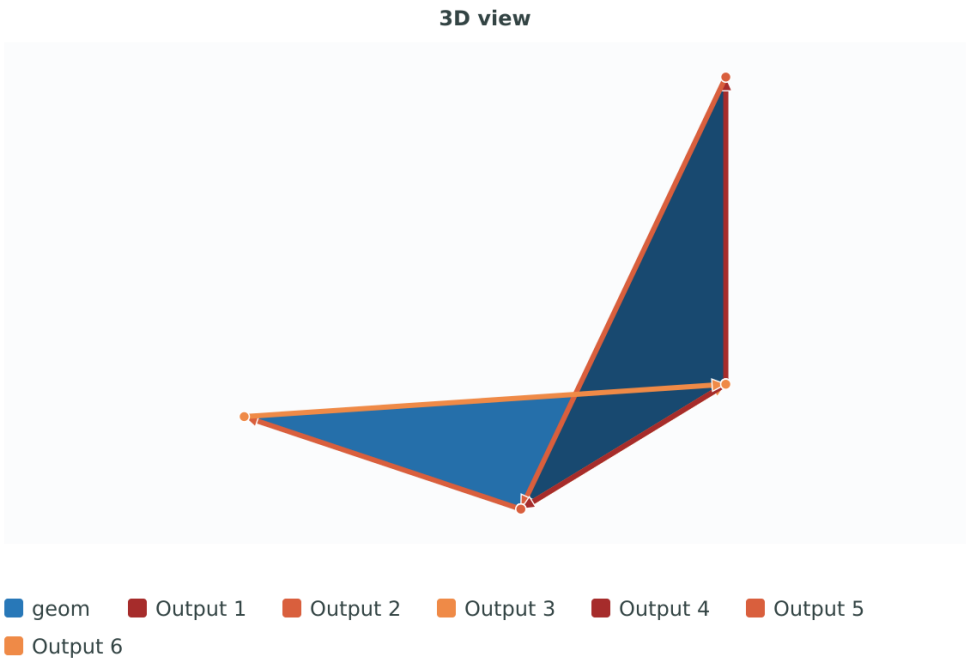


This example uses a TIN geometry.

Code

```
SELECT path,geom
FROM (
    SELECT (ST_DumpSegments(g.geom)).*
    FROM (SELECT 'TIN(((
        0 0 0,
        0 0 1,
        0 1 0,
        0 0 0
    )),((
        0 0 0,
        0 1 0,
        1 1 0,
        0 0 0
    ))
    )'::geometry AS geom
        ) AS g
) j;
```

Figure



See Also

geometry_dump, Section 13.6, ST_Dump, ST_DumpRings

7.4.9 ST_DumpRings

ST_DumpRings — Returns a set of `geometry_dump` rows for the exterior and interior rings of a Polygon.

Synopsis

geometry_dump[] **ST_DumpRings**(geometry a_polygon);

Description

A set-returning function (SRF) that extracts the rings of a polygon. It returns a set of geometry_dump rows, each containing a geometry (*`geom`* field) and an array of integers (*`path`* field).

The *`geom`* field contains each ring as a POLYGON. The *`path`* field is an integer array of length 1 containing the polygon ring index. The exterior ring (shell) has index 0. The interior rings (holes) have indices of 1 and higher.



Note
This only works for POLYGON geometries, not MULTIPOLYGONs. Use ST_Dump to extract polygon elements from polygonal geometries: `ST_DumpRings((ST_Dump(geom)).geom )`

Availability: PostGIS 1.1.3. Requires PostgreSQL 7.3 or higher.

 This function supports 3d and will not drop the z-index.

Examples

Extracting all rings as polygons.

Code

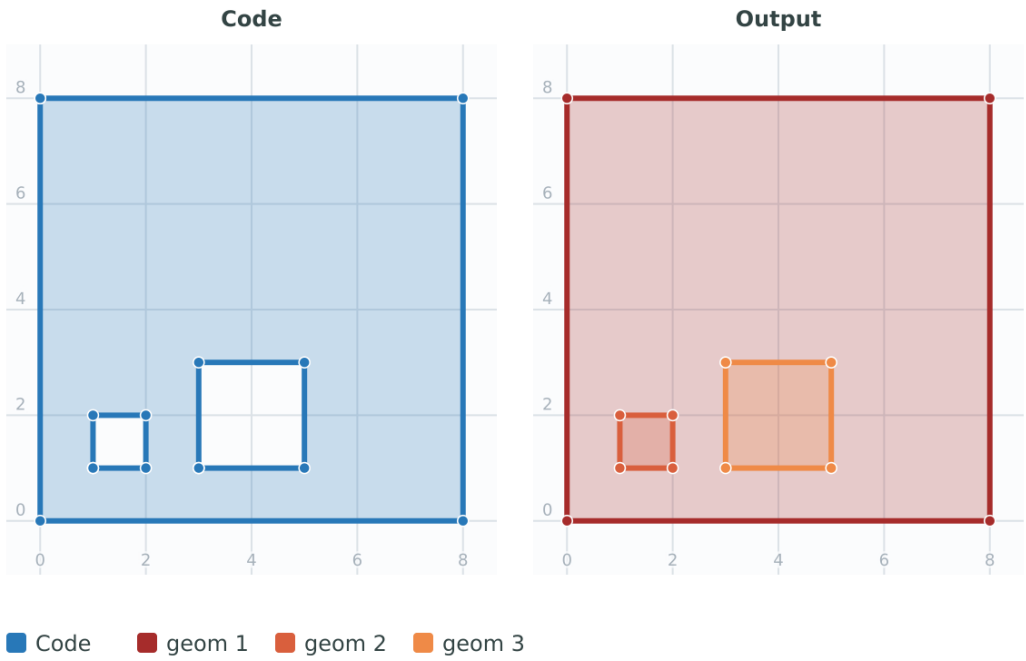
```
SELECT polyTable.field1, polyTable.field1,
      (ST_DumpRings(polyTable.geom)).geom As geom
FROM polyTable
```

Extracting shell and holes from a polygon.

Code

```
SELECT path,geom As geom
FROM ST_DumpRings('POLYGON ((1 9,9 9,9 1,1 1,1 9),(2 2,2 3,3 3,3 2,2 2),(4 2,4 4,6 4,6 2,4 ↩
    2))');
```

Figure



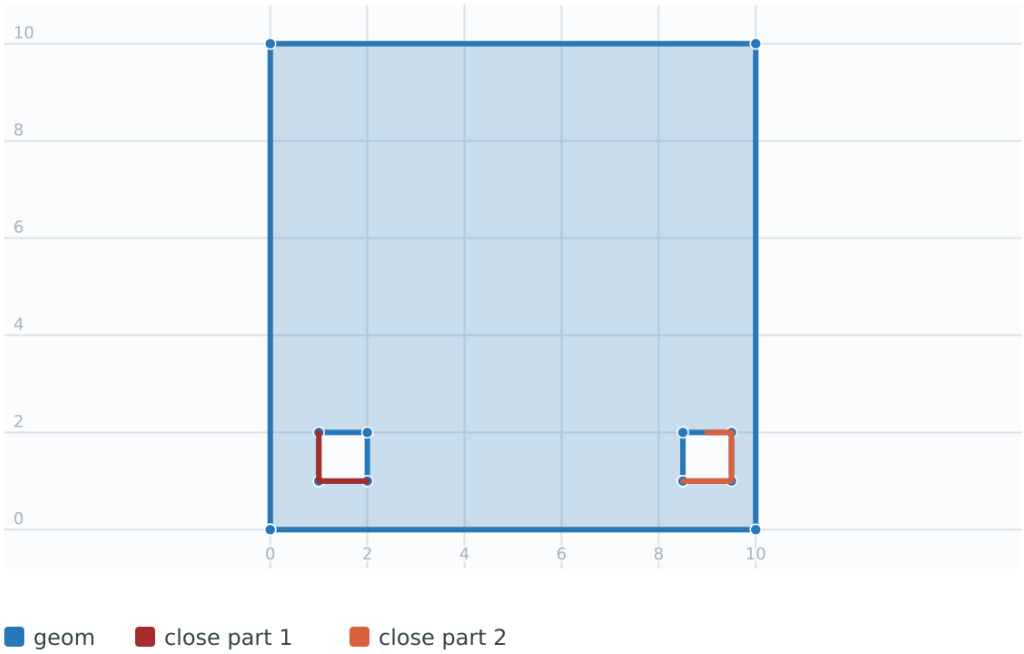
Finding interior rings which lie close to the exterior ring.

Code

```
WITH poly AS (
  SELECT 'POLYGON ((
      0 0, 10 0, 10 10, 0 10, 0 0
    ), (
      1 1, 1 2, 2 2, 2 1, 1 1
    ), (
      8.5 1, 8.5 2, 9.5 2, 9.5 1, 8.5 1
    ))'::geometry AS geom
),
rings AS (
  SELECT d.path[1] AS ring_no,
         ST_Boundary(d.geom) AS ring_geom
  FROM poly
  CROSS JOIN LATERAL ST_DumpRings(poly.geom) AS d
```

```
),
shell AS (
  SELECT ring_geom
  FROM rings
  WHERE ring_no = 0
),
holes AS (
  SELECT ring_no, ring_geom
  FROM rings
  WHERE ring_no > 0
)
SELECT h.ring_no,
       ST_Normalize(ST_Multi(ST_LineMerge(
         ST_Intersection(ST_Buffer(s.ring_geom, 1.0), h.ring_geom)
       ))) AS close_part
FROM shell AS s
JOIN holes AS h
  ON ST_DWithin(s.ring_geom, h.ring_geom, 1.0);
```

Figure



See Also

geometry_dump, Section 13.6, ST_Boundary, ST_Buffer, ST_Dump, ST_DWithin, ST_ExteriorRing, ST_InteriorRingN, ST_Intersecti

7.4.10 ST_EndPoint

ST_EndPoint — Returns the last point of a LineString, CircularLineString, or NURBSCurve.

Synopsis

geometry **ST_EndPoint**(geometry g);

Description

Returns the last point of a LINESTRING, CIRCULARLINESTRING, or NURBSCURVE geometry as a POINT. Returns NULL if the input is not a LINESTRING, CIRCULARLINESTRING, or NURBSCURVE.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.4

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Changed: 2.0.0 no longer works with single geometry MultiLineStrings. In older versions of PostGIS a single-line MultiLineString would work with this function and return the end point. In 2.0.0 it returns NULL like any other MultiLineString. The old behavior was an undocumented feature, but people who assumed they had their data stored as LINESTRING may experience these returning NULL in 2.0.0.

Examples

End point of a LineString

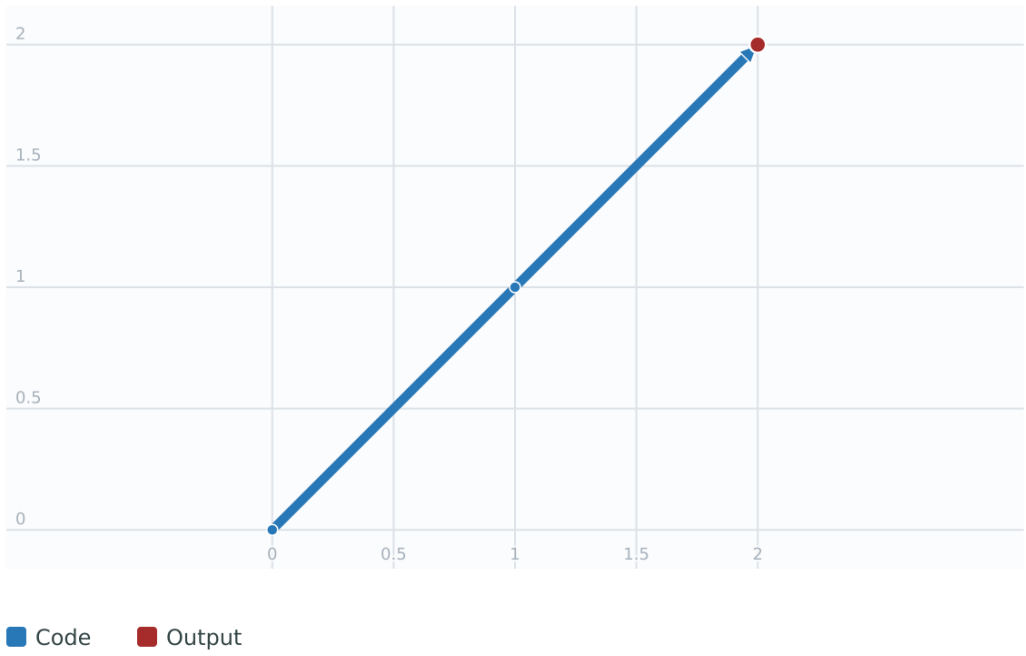
Code

```
SELECT ST_EndPoint('LINESTRING(1 1,2 2,3 3)'::geometry);
```

Output

```
POINT(3 3)
```

Figure



End point of a non-LineString is NULL

Code

```
SELECT ST_EndPoint('POINT(1 1)'::geometry) IS NULL AS is_null;
```

Output

```
t
```

End point of a 3D LineString

3D endpoint.

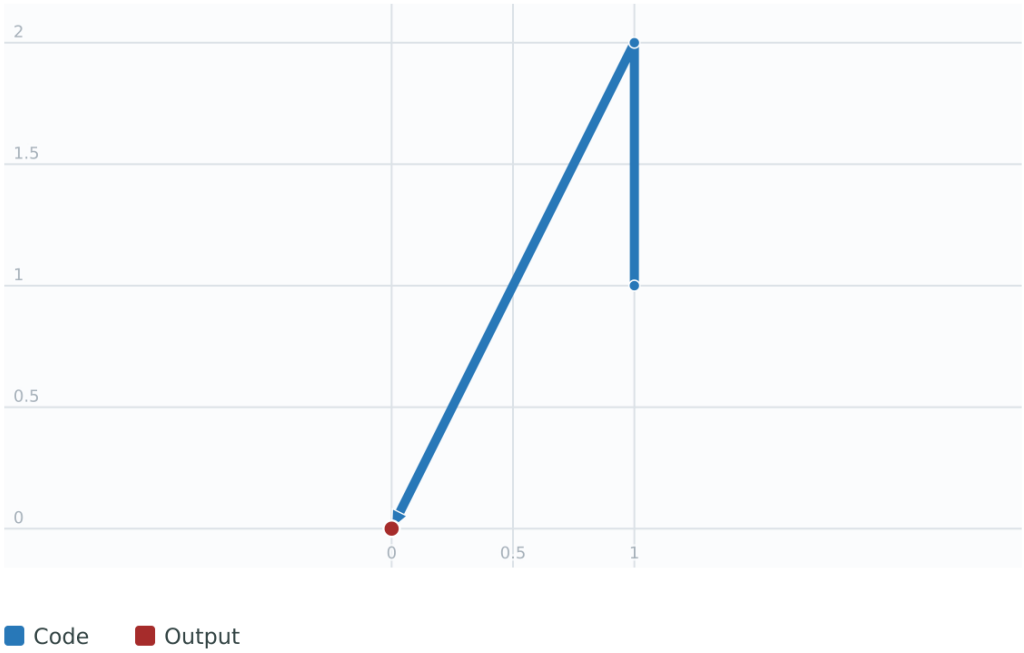
Code

```
SELECT ST_EndPoint('LINESTRING(1 1 2,1 2 3,0 0 5)');
```

Output

```
POINT(0 0 5)
```

Figure



End point of a CircularString

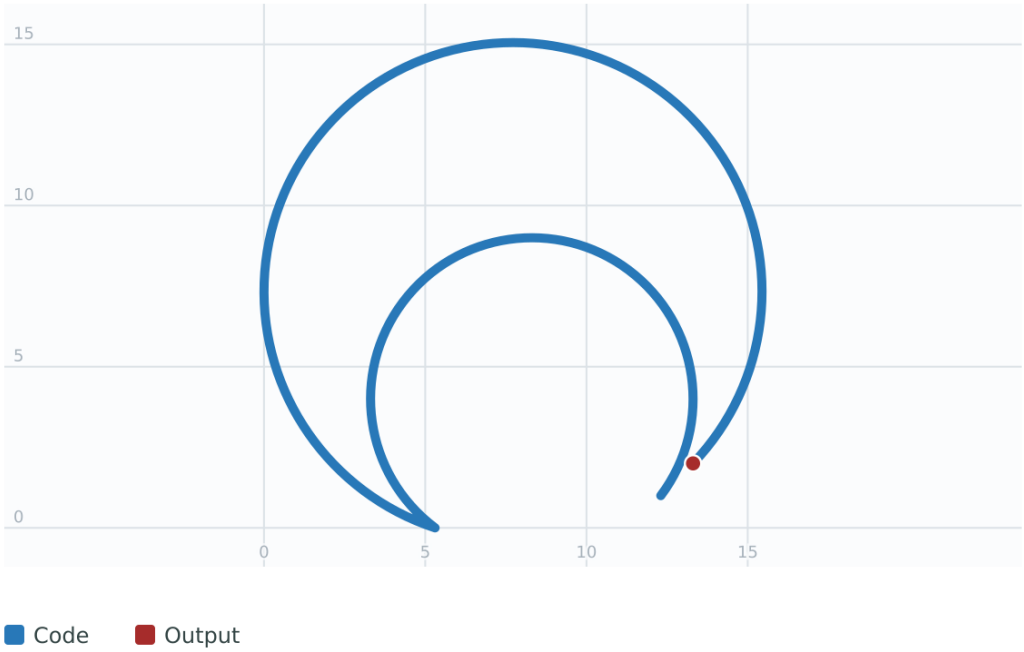
Code

```
SELECT ST_EndPoint('CIRCULARSTRING(5 2,-3 2,-2 1,-4 2,6 3)'::geometry);
```

Output

```
POINT(6 3)
```

Figure



End point of a NURBSCurve

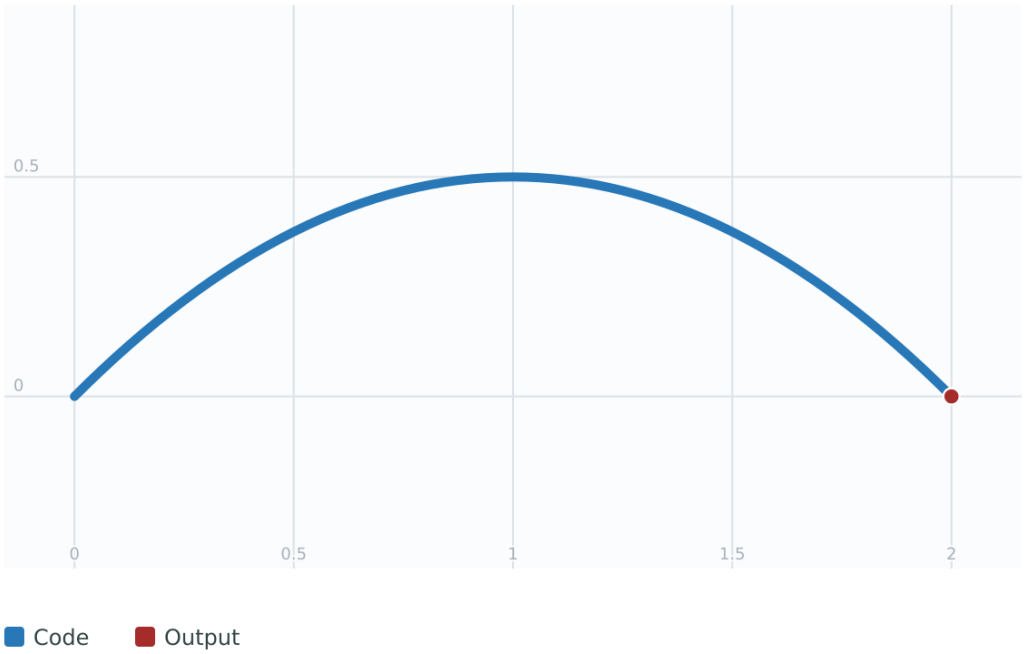
Code

```
SELECT ST_EndPoint('NURBSCURVE(2, (0 0, 1 1, 2 0))'::geometry);
```

Output

```
POINT(2 0)
```

Figure



See Also

ST_PointN, ST_StartPoint

7.4.11 ST_Envelope

ST_Envelope — Returns a geometry representing the bounding box of a geometry.

Synopsis

geometry **ST_Envelope**(geometry g1);

Description

Returns the double-precision (float8) minimum bounding box for the supplied geometry, as a geometry. The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MINX, MAXY), (MAXX, MAXY), (MAXX, MINY), (MINX, MINY)). (PostGIS will add a ZMIN/ZMAX coordinate as well).

Degenerate cases (vertical lines, points) will return a geometry of lower dimension than POLYGON, ie. POINT or LINESTRING.

Availability: 1.5.0 behavior changed to output double precision instead of float4

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.19

Examples

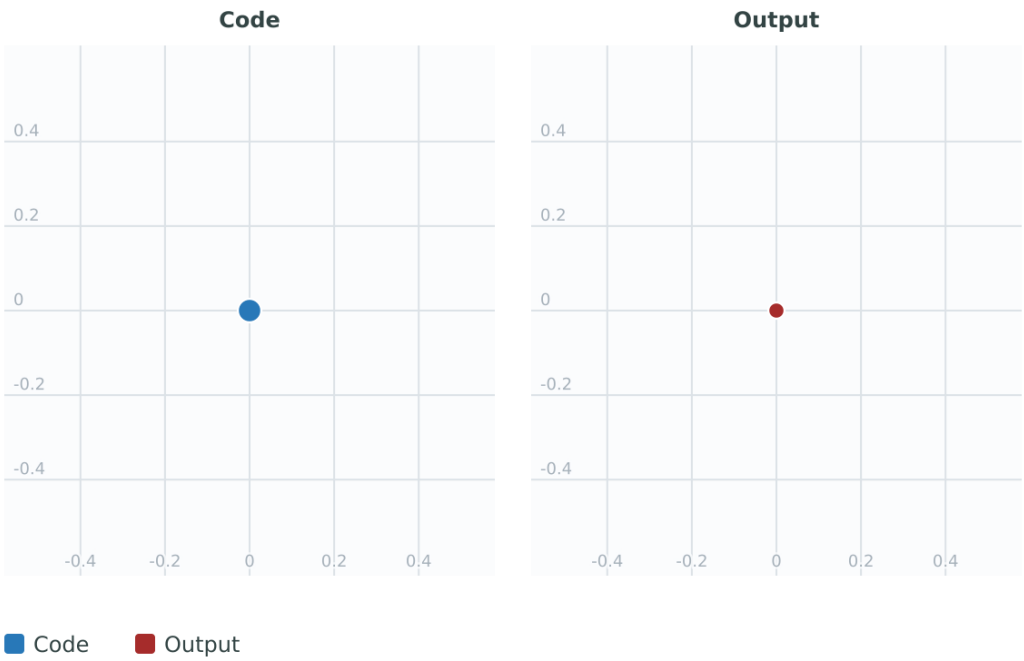
Code

```
SELECT ST_Envelope('POINT(1 3)'::geometry);
```

Output

```
POINT(1 3)
```

Figure



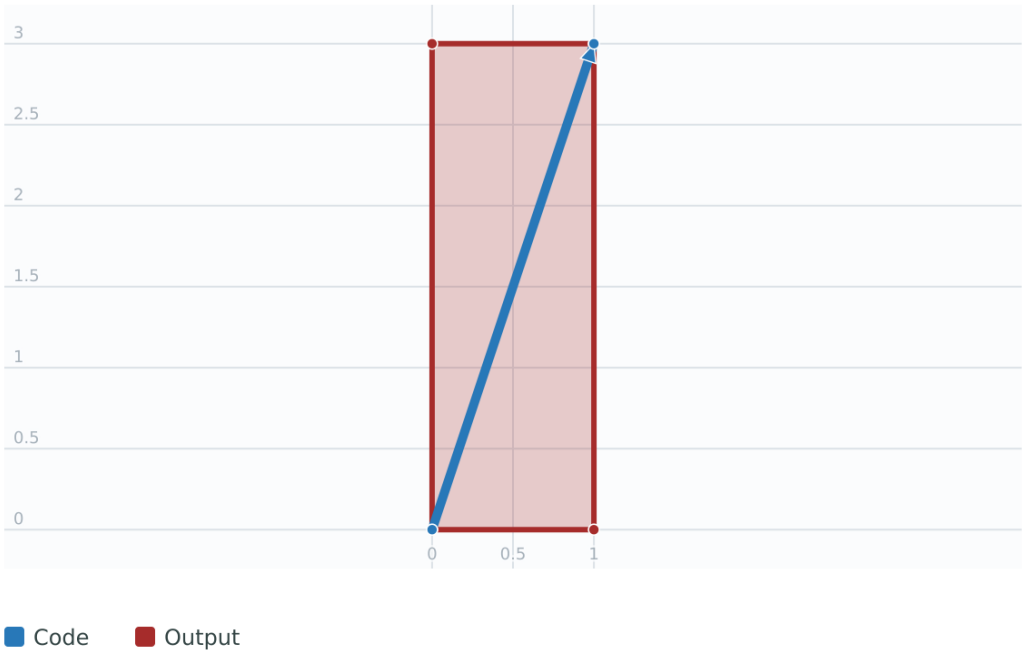
Code

```
SELECT ST_Envelope('LINESTRING(0 0,1 3)'::geometry);
```

Output

```
POLYGON((0 0,0 3,1 3,1 0,0 0))
```

Figure



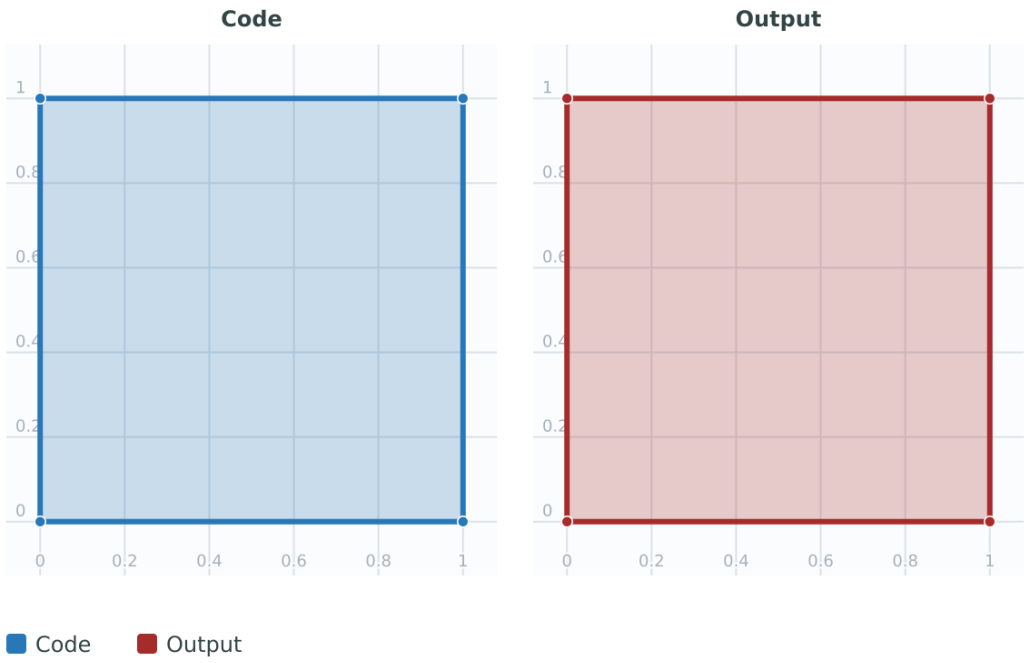
Code

```
SELECT ST_Envelope('POLYGON((0 0,0 1,1.0000001 1,1.0000001 0,0 0))'::geometry);
```

Output

```
POLYGON((0 0,0 1,1.0000001 1,1.0000001 0,0 0))
```

Figure



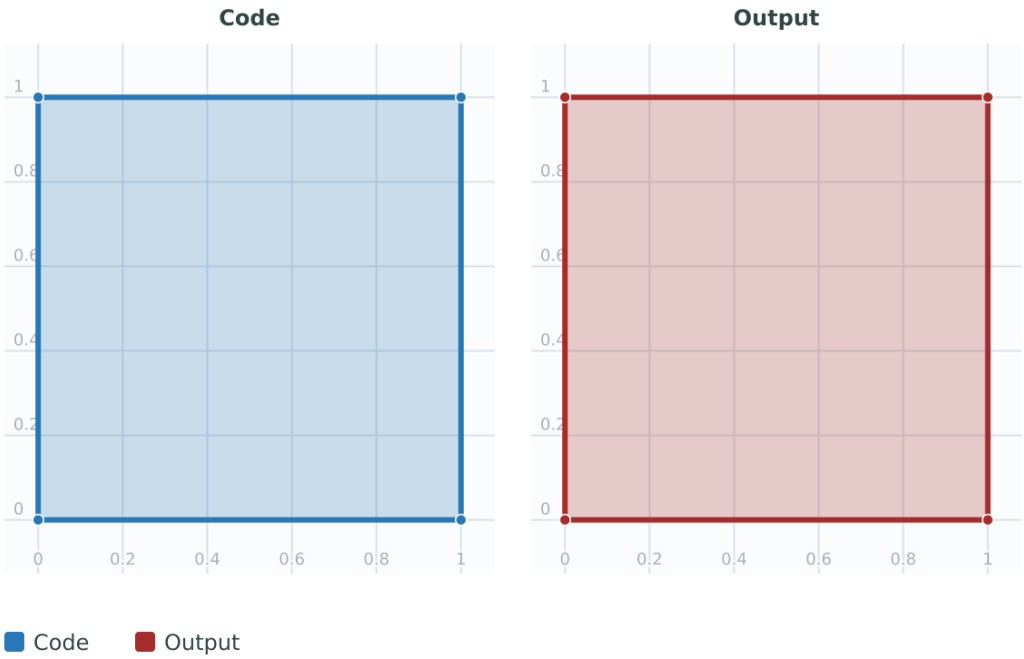
Code

```
SELECT ST_Envelope('POLYGON((0 0,0 1,1.0000000001 1,1.0000000001 0,0 0))'::geometry);
```

Output

```
POLYGON((0 0,0 1,1.0000000001 1,1.0000000001 0,0 0))
```

Figure



Code

```
SELECT Box3D(geom), Box2D(geom), ST_Envelope(geom) As envelopewkt
    FROM (SELECT 'POLYGON((0 0,0 1000012333334.34545678,1.0000001 1,1.0000001 0,0 0))':: ↩
    geometry As geom) As foo;
```

Envelope of a point and LineString.

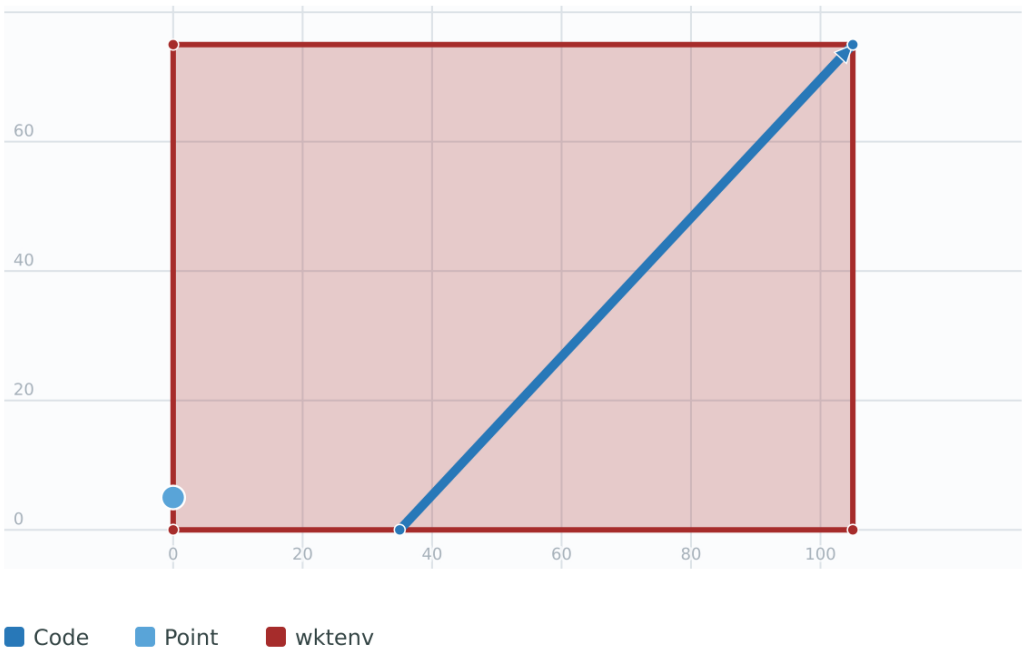
Code

```
SELECT ST_Envelope(ST_Collect(
            'LINESTRING(55 75,125 150)',
            ST_Point(20, 80)
            )) As wktenv;
```

Output

```
POLYGON((20 75,20 150,125 150,125 75,20 75))
```

Figure



See Also

Box2D, Box3D, ST_OrientedEnvelope

7.4.12 ST_ExteriorRing

ST_ExteriorRing — Returns a LineString representing the exterior ring of a Polygon.

Synopsis

geometry **ST_ExteriorRing**(geometry a_polygon);

Description

Returns a LINESTRING representing the exterior ring (shell) of a POLYGON. Returns NULL if the geometry is not a polygon.



Note
This function does not support MULTIPOLYGONs. For MULTIPOLYGONs use in conjunction with ST_GeometryN or ST_Dump

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. 2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 8.2.3, 8.3.3

This function supports 3d and will not drop the z-index.

Examples

Extract the exterior ring from a Polygon with a hole.

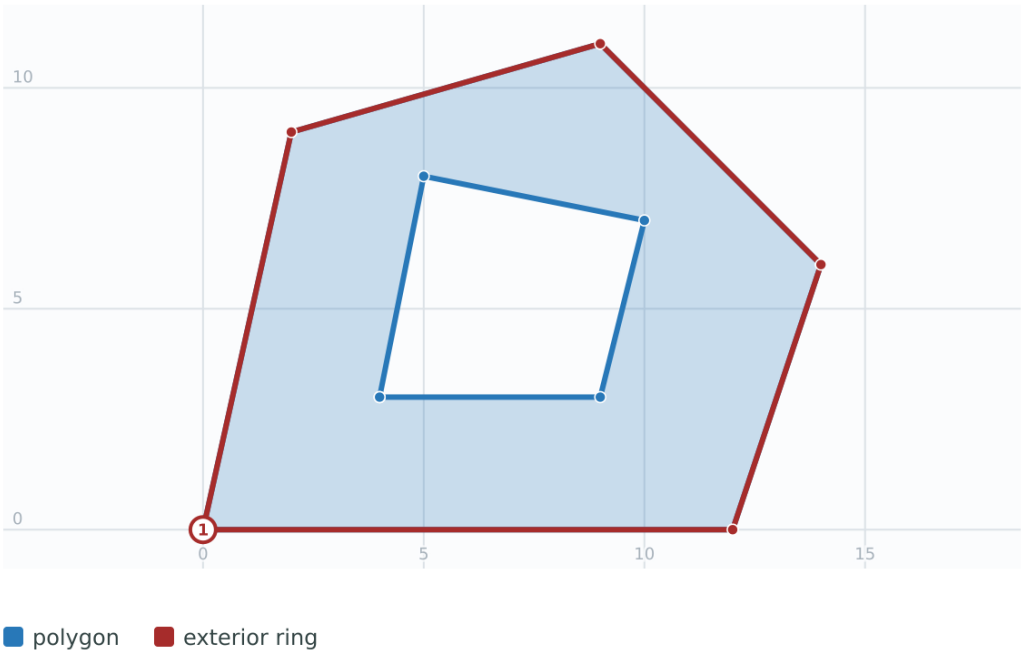
Code

```
WITH input(geom) AS (VALUES (
  'POLYGON((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3))'::geometry
))
SELECT geom AS input_polygon,
       ST_ExteriorRing(geom) AS exterior_ring
FROM input;
```

Output

```
POLYGON((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3)) | LINESTRING(0 0,2 9,9 11,14 6,12 0,0 0)
```

Figure



For a MultiPolygon, dump the component Polygons before extracting their exterior rings. Both component Polygons in this example have holes.

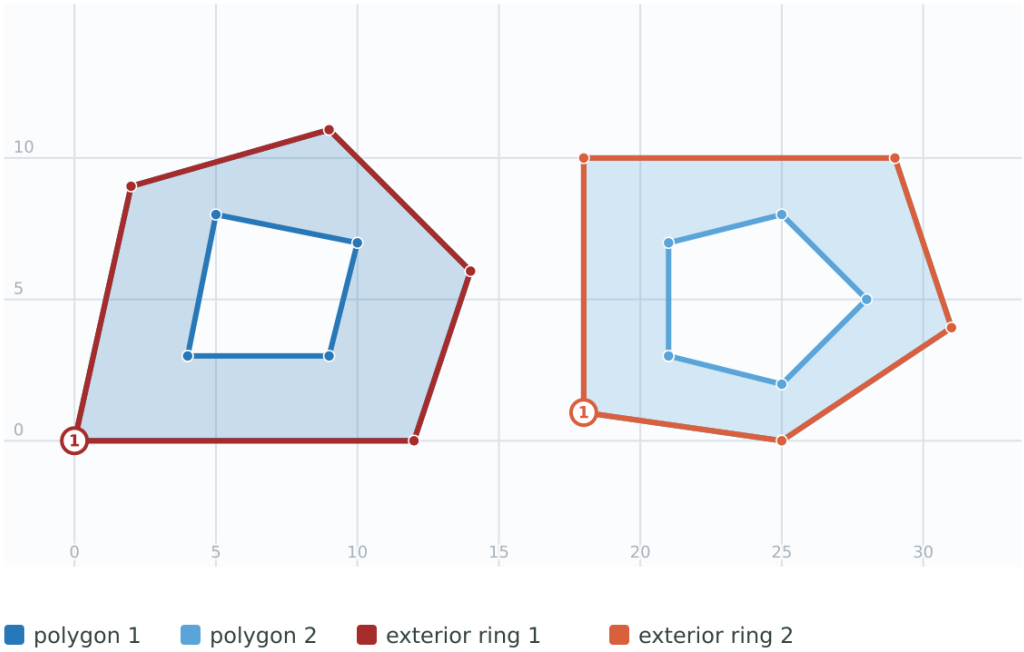
Code

```
WITH input(geom) AS (VALUES (
  'MULTIPOLYGON(
    ((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3)),
    ((18 1,18 10,29 10,31 4,25 0,18 1),(21 3,25 2,28 5,25 8,21 7,21 3))
  )'::geometry
))
SELECT part.geom AS input_polygon,
       ST_ExteriorRing(part.geom) AS exterior_ring
FROM input
CROSS JOIN LATERAL ST_Dump(geom) AS part
ORDER BY part.path;
```

Output

```
POLYGON((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3)) | LINESTRING(0 0,2 9,9 11,14 6,12 0,0 0)
POLYGON((18 1,18 10,29 10,31 4,25 0,18 1),(21 3,25 2,28 5,25 8,21 7,21 3)) | LINESTRING(18 1,18 10,29 10,31 4,25 0,18 ↩
    1)
```

Figure



Extract exterior rings from a table of Polygons.

Code

```
SELECT gid, ST_ExteriorRing(geom) AS ering
FROM sometable;
```

For a table of MultiPolygons, dump the component Polygons and collect their exterior rings into a MultiLineString.

Code

```
SELECT gid, ST_Collect(ST_ExteriorRing(geom)) AS erings
FROM (
  SELECT gid, (ST_Dump(geom)).geom AS geom
  FROM sometable
) AS foo
GROUP BY gid;
```

This example returns the 3D exterior ring of a Polygon.

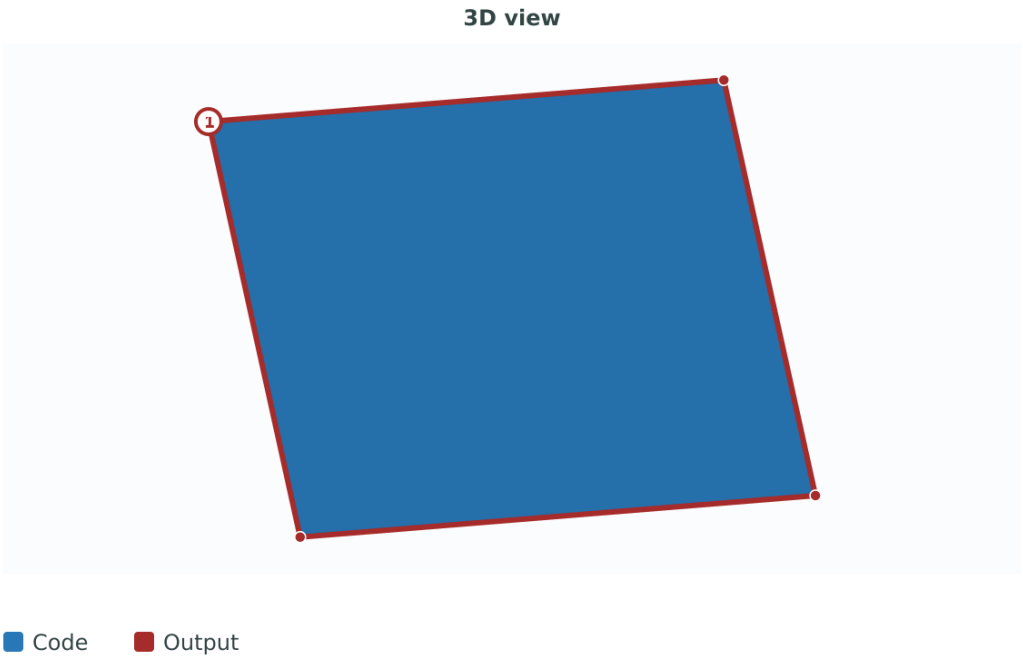
Code

```
SELECT ST_ExteriorRing(
    'POLYGON Z ((0 0 5,4 0 5,4 3 5,0 3 5,0 0 5))'
    );
```

Output

```
LINESTRING(0 0 5,4 0 5,4 3 5,0 3 5,0 0 5)
```

Figure



See Also

ST_InteriorRingN, ST_Boundary, ST_NumInteriorRings

7.4.13 ST_GeometryN

ST_GeometryN — Return an element of a geometry collection.

Synopsis

geometry **ST_GeometryN**(geometry geomA, integer n);

Description

Returns the 1-based Nth element of a GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING, MULTICURVE, or MULTIPOLYGON. For a unitary geometry, including a TIN or PolyhedralSurface, an index of 1 returns the input geometry and any other index returns NULL. Use ST_PatchN and ST_NumPatches to access and count the faces of a TIN or PolyhedralSurface.



Note
Index is 1-based as for OGC specs since version 0.8.0. Previous versions implemented this as 0-based instead.



Note
To extract all elements of a geometry, ST_Dump is more efficient and works for atomic geometries.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Changed: 2.0.0 Prior versions would return NULL for singular geometries. This was changed to return the geometry for ST_GeometryN(..,1) case.

Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and are not decomposed into patches; use ST_PatchN to access faces.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 9.1.5

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

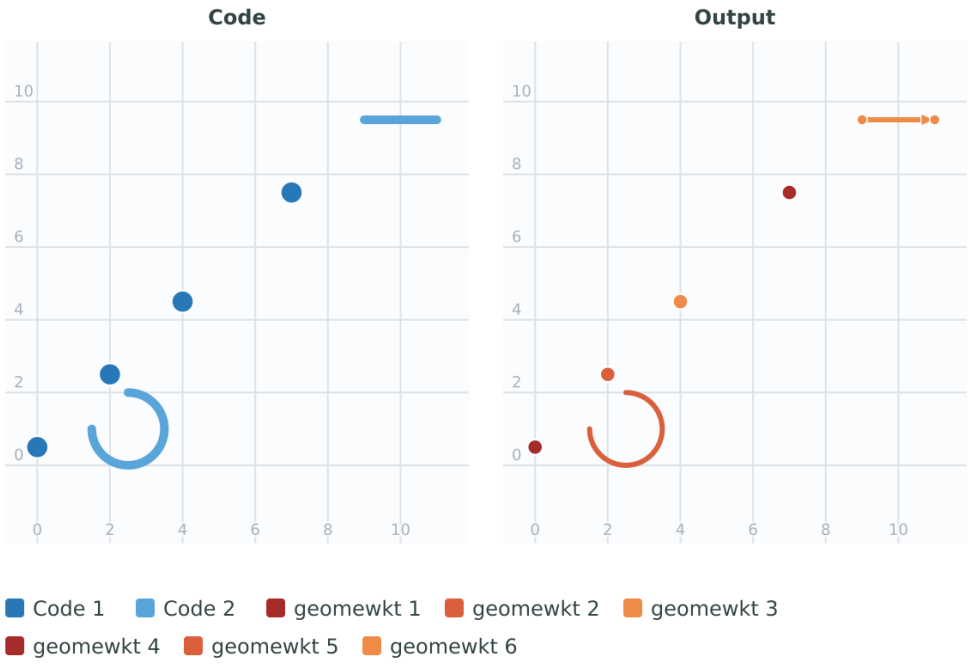
Standard Examples

Extracting a subset of points from a 3D multipoint.

Code

```
SELECT n, ST_GeometryN(geom, n) As geomewkt
FROM (
VALUES
    (1, 'MULTIPOINT((1 2 7), (3 4 7), (5 6 7), (8 9 10))'::geometry ),
    (2, 'MULTICURVE(CIRCULARSTRING(2.5 2.5,4.5 2.5, 3.5 3.5), (10 11, 12 11))'::geometry )
    ) As foo(id, geom)
    CROSS JOIN generate_series(1,100) n
WHERE n <= ST_NumGeometries(geom)
ORDER BY id, n;
```

Figure



Extracting all geometries (useful when you want to assign an id).

Code

```
SELECT gid, n, ST_GeometryN(geom, n)
FROM sometable CROSS JOIN generate_series(1,100) n
WHERE n <= ST_NumGeometries(geom);
```

Polyhedral Surfaces, TIN and Triangle Examples

A PolyhedralSurface or TIN is treated as a single geometry, not as a collection of faces. `ST_GeometryN` does not decompose the surface into patches, so an index greater than 1 returns `NULL`. Use ST_PatchN and ST_NumPatches to access individual faces.

This example requests the third element of a PolyhedralSurface.

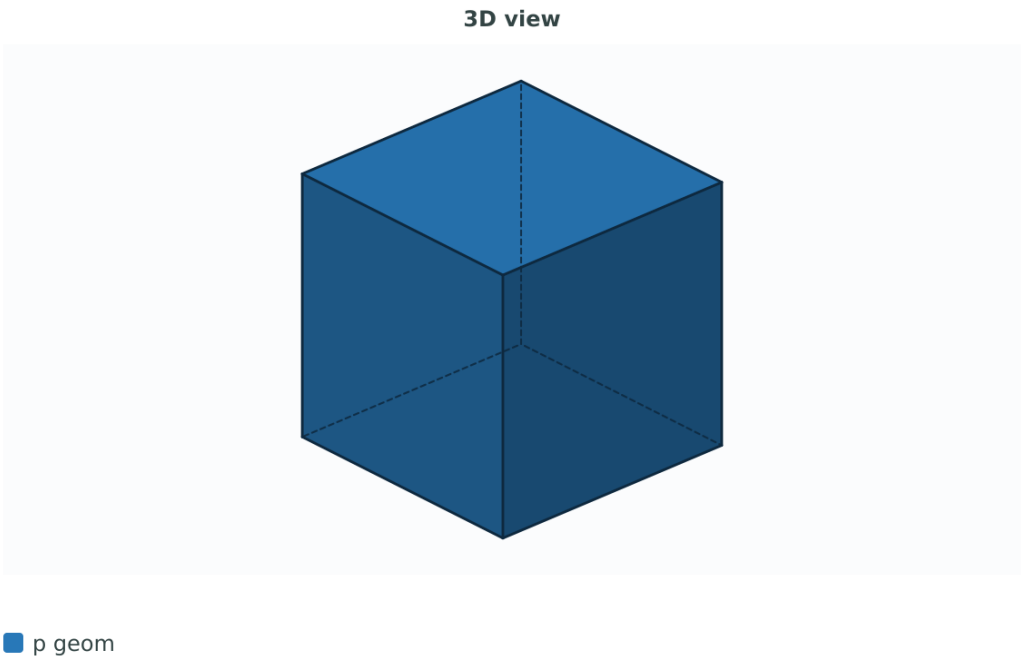
Code

```
SELECT ST_GeometryN(p_geom, 3) As geom_ewkt
  FROM (SELECT 'POLYHEDRALSURFACE(
((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)'::geometry  AS p_geom )  AS a;
```

Output

```
NULL
```

Figure



This example requests the second element of a TIN.

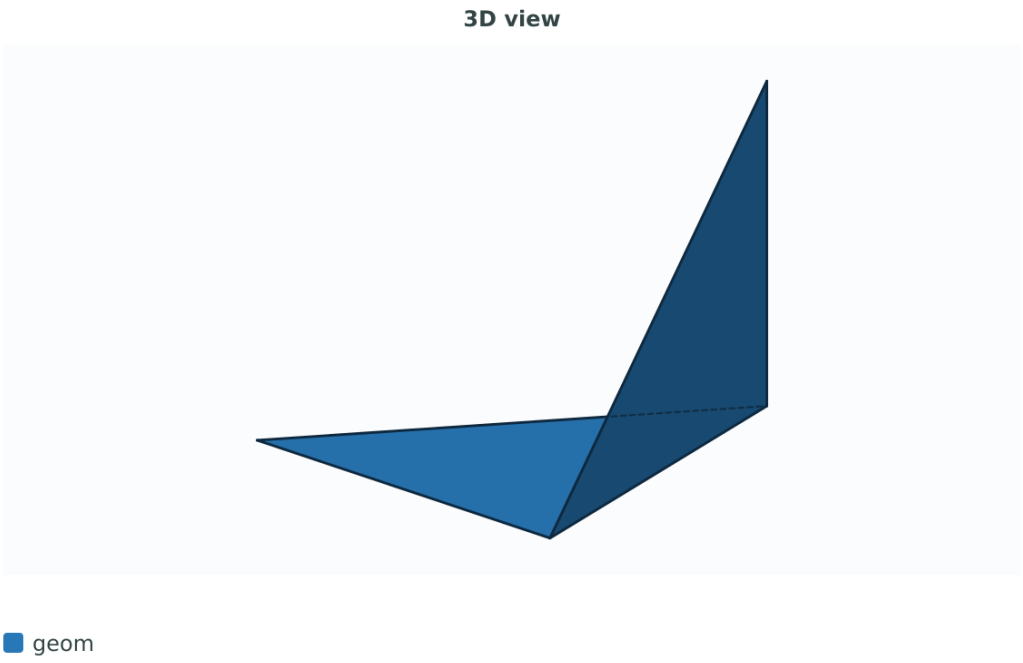
Code

```
SELECT ST_GeometryN(geom, 2) as wkt
  FROM
    (SELECT
       'TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
            )),((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
            ))
            )'::geometry  AS geom
    ) AS g;
```

Output

```
NULL
```

Figure



See Also

ST_Dump, ST_NumGeometries, ST_PatchN, ST_NumPatches

7.4.14 ST_GeometryType

ST_GeometryType — Returns the SQL-MM type of a geometry as text.

Synopsis

text **ST_GeometryType**(geometry g1);

Description

Returns the type of the geometry as a string. EG: 'ST_LineString', 'ST_Polygon','ST_MultiPolygon' etc. This function differs from GeometryType(geometry) in the case of the string and ST in front that is returned, as well as the fact that it will not indicate whether the geometry is measured.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.4

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_GeometryType('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)':: ↩
    geometry);
```

Output

```
ST_LineString
```

Code

```
SELECT ST_GeometryType('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry);
```

Output

```
ST_PolyhedralSurface
```

Code

```
SELECT ST_GeometryType(geom) AS result
FROM
  (SELECT
     'TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
          )),((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
          ))
          )'::geometry AS geom
  ) AS g;
```

Output

```
ST_Tin
```

See Also

GeometryType

7.4.15 ST_HasArc

ST_HasArc — Tests if a geometry contains a circular arc

Synopsis

boolean **ST_HasArc**(geometry geomA);

Description

Returns true if a geometry or geometry collection contains a circular string

Availability: 1.2.3?

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

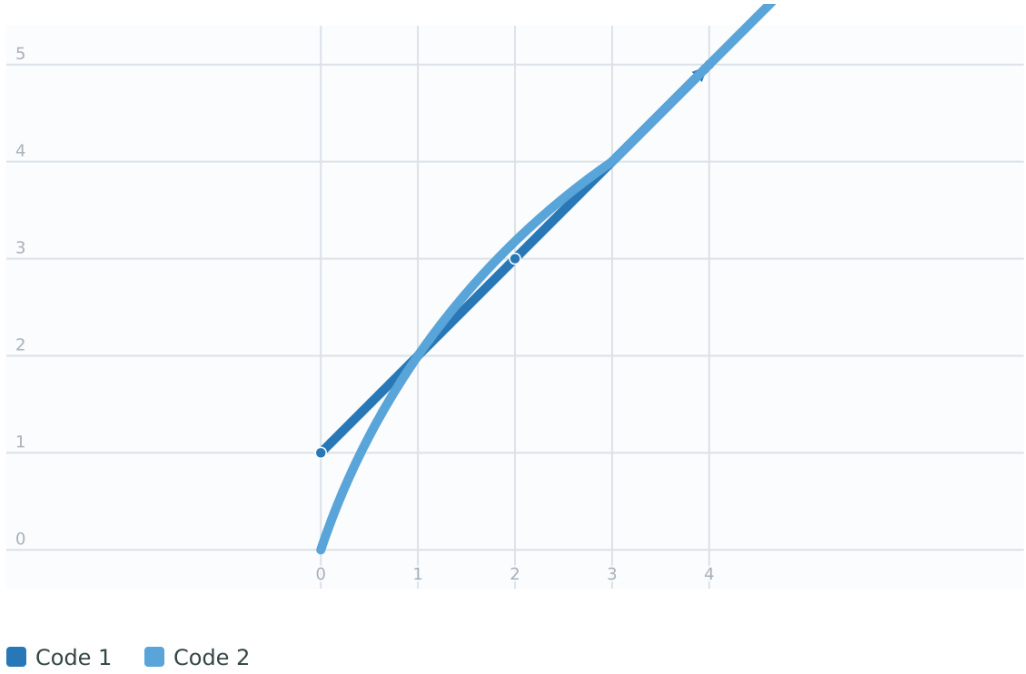
Code

```
SELECT ST_HasArc(ST_Collect('LINESTRING(1 2,3 4,5 6)', 'CIRCULARSTRING(1 1,2 3,4 5,6 7,5 6) ↩
    '));
```

Output

```
t
```

Figure



See Also

ST_CurveToLine, ST_LineToCurve

7.4.16 ST_InteriorRingN

ST_InteriorRingN — Returns the Nth interior ring (hole) of a Polygon.

Synopsis

geometry **ST_InteriorRingN**(geometry a_polygon, integer n);

Description

Returns the Nth interior ring (hole) of a POLYGON geometry as a LINESTRING. The index starts at 1. Returns NULL if the geometry is not a polygon or the index is out of range.



Note
This function does not support MULTIPOLYGONs. For MULTIPOLYGONs use in conjunction with ST_GeometryN or ST_Dump

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

 This method implements the SQL/MM specification. SQL-MM 3: 8.2.6, 8.3.5

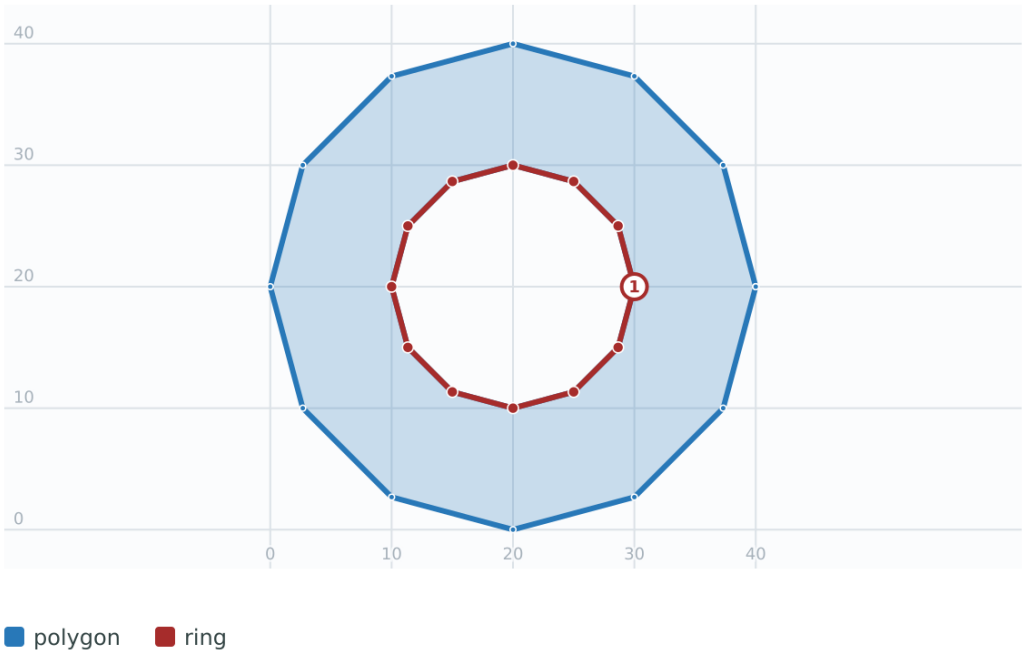
 This function supports 3d and will not drop the z-index.

Examples

Code

```
WITH data AS (
  SELECT ST_BuildArea(ST_Collect(
    ST_Buffer(ST_Point(1, 2), 20, 3),
    ST_Buffer(ST_Point(1, 2), 10, 3)
  )) AS geom
)
SELECT geom AS input_polygon,
       ST_InteriorRingN(geom, 1) AS ring
FROM data;
```

Figure



See Also

ST_ExteriorRing, ST_BuildArea, ST_Collect, ST_Dump, ST_NumInteriorRing, ST_NumInteriorRings

7.4.17 ST_NumCurves

ST_NumCurves — Return the number of component curves in a CompoundCurve.

Synopsis

integer **ST_NumCurves**(geometry a_compoundcurve);

Description

Return the number of component curves in a CompoundCurve, zero for an empty CompoundCurve, or NULL for a non-CompoundCurve input.

This method implements the SQL/MM specification. SQL-MM 3: 8.2.6, 8.3.5

This function supports 3d and will not drop the z-index.

Examples

This CompoundCurve contains three component curves.

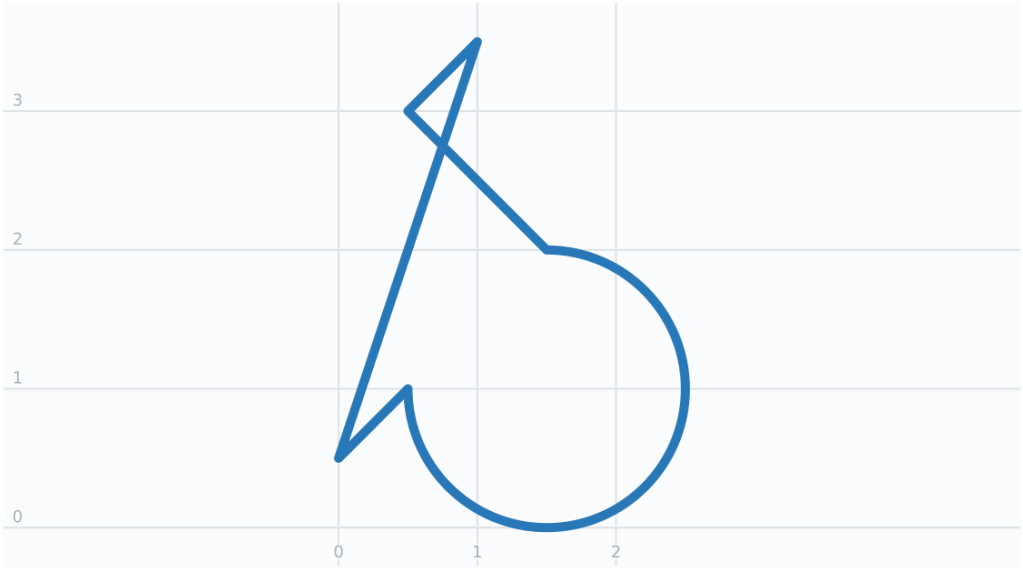
Code

```
SELECT ST_NumCurves('COMPOUNDCURVE(
    (2 2,2.5 2.5),
    CIRCULARSTRING(2.5 2.5,4.5 2.5,3.5 3.5),
    (3.5 3.5,2.5 4.5,3 5,2 2)
  )');
```

Output

```
3
```

Figure



An empty CompoundCurve has no component curves.

Code

```
SELECT ST_NumCurves('COMPOUNDCURVE EMPTY');
```

Output

```
0
```

See Also

ST_CurveN, ST_Dump, ST_ExteriorRing, ST_NumInteriorRings, ST_NumGeometries

7.4.18 ST_CurveN

ST_CurveN — Returns the Nth component curve geometry of a CompoundCurve.

Synopsis

geometry **ST_CurveN**(geometry a_compoundcurve, integer index);

Description

Returns the Nth component curve geometry of a CompoundCurve. The index starts at 1. Returns NULL if the geometry is not a CompoundCurve or the index is out of range.

This method implements the SQL/MM specification. SQL-MM 3: 8.2.6, 8.3.5

This function supports 3d and will not drop the z-index.

Examples

This example extracts the second component curve, a CircularString, from a CompoundCurve.

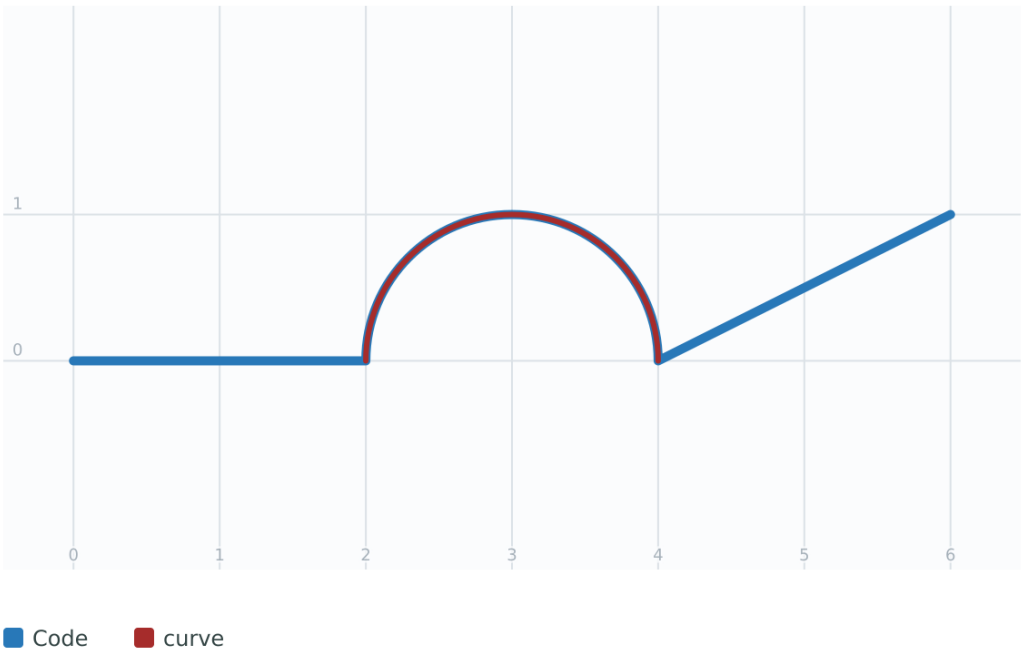
Code

```
SELECT ST_CurveN(
  'COMPOUNDCURVE(
    (0 0,2 0),
    CIRCULARSTRING(2 0,3 1,4 0),
    (4 0,6 1)
  )',
  2) AS curve;
```

Output

```
CIRCULARSTRING(2 0,3 1,4 0)
```

Figure



See Also

ST_NumCurves, ST_Dump, ST_ExteriorRing, ST_NumInteriorRings, ST_NumGeometries

7.4.19 ST_IsClosed

ST_IsClosed — Tests if a LineStrings's start and end points are coincident. For a PolyhedralSurface tests if it is closed (volumetric).

Synopsis

boolean **ST_IsClosed**(geometry g);

Description

Returns `TRUE` if the `LINESTRING`'s start and end points are coincident. For Polyhedral Surfaces, reports if the surface is areal (open) or volumetric (closed).

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.5, 9.3.3



Note
SQL-MM defines the result of `ST_IsClosed(NULL)` to be 0, while PostGIS returns `NULL`.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

This function supports Polyhedral surfaces.

Line String and Point Examples

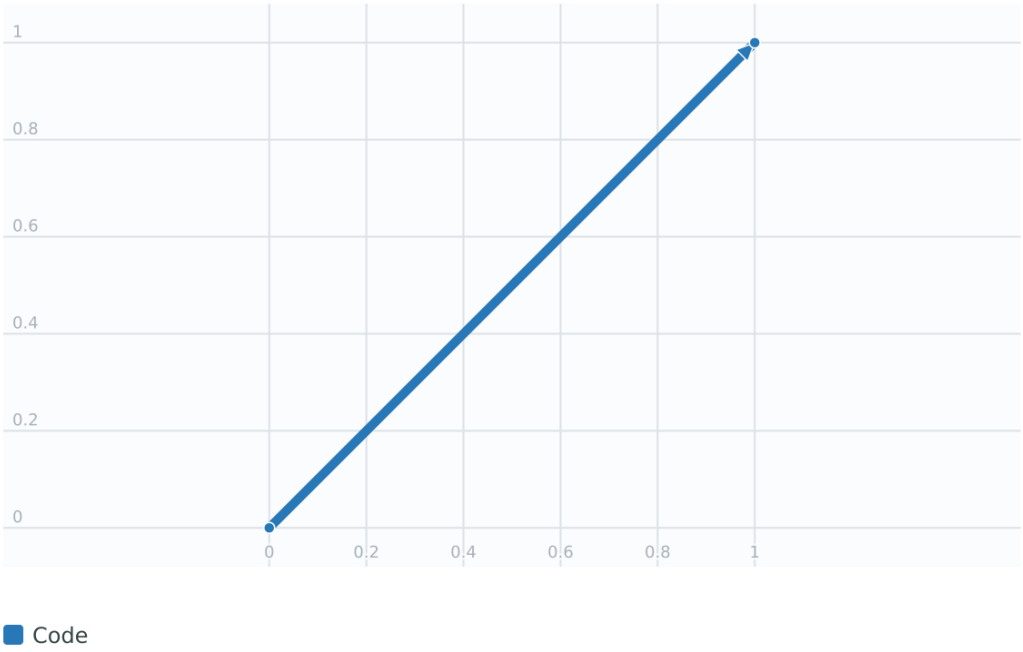
Code

```
SELECT ST_IsClosed('LINESTRING(0 0,1 1)'::geometry);
```

Output

```
f
```

Figure



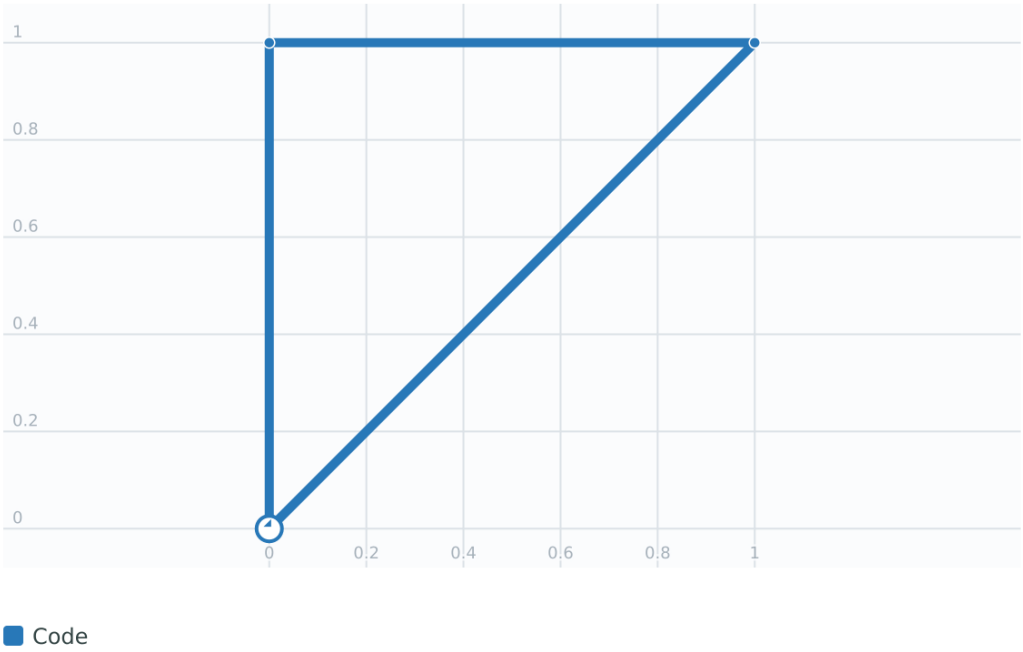
Code

```
SELECT ST_IsClosed('LINESTRING(0 0,0 1,1 1,0 0)'::geometry);
```

Output

```
t
```

Figure



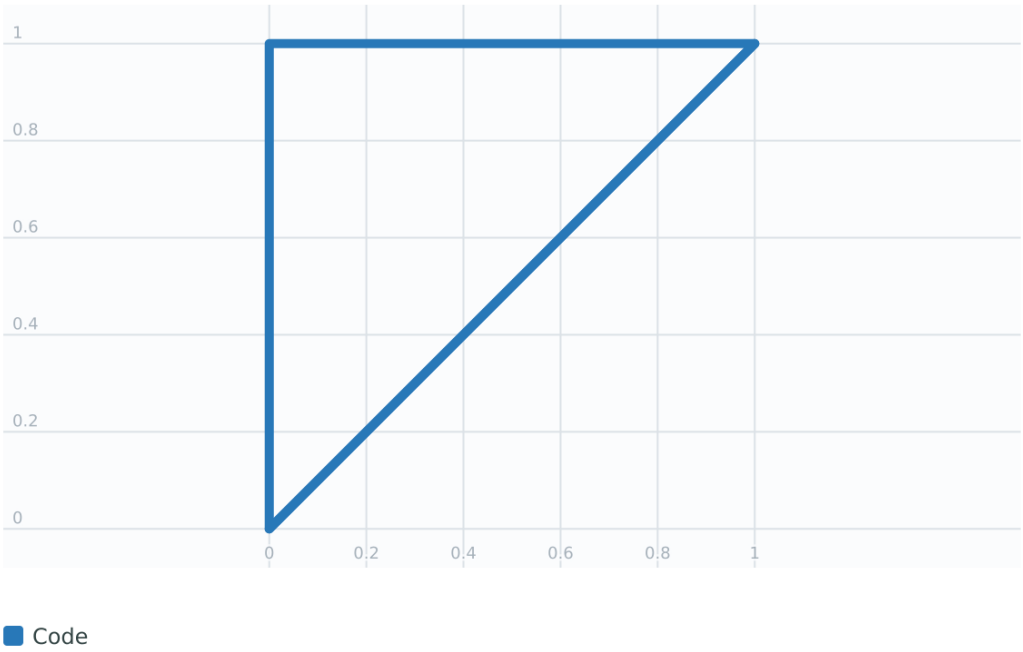
Code

```
SELECT ST_IsClosed('MULTILINESTRING((0 0,0 1,1 1,0 0),(0 0,1 1))'::geometry);
```

Output

```
f
```

Figure



Code

```
SELECT ST_IsClosed('POINT(0 0)'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsClosed('MULTIPOINT((0 0),(1 1))'::geometry);
```

Output

```
t
```

Polyhedral Surface Examples

This example uses a cube.

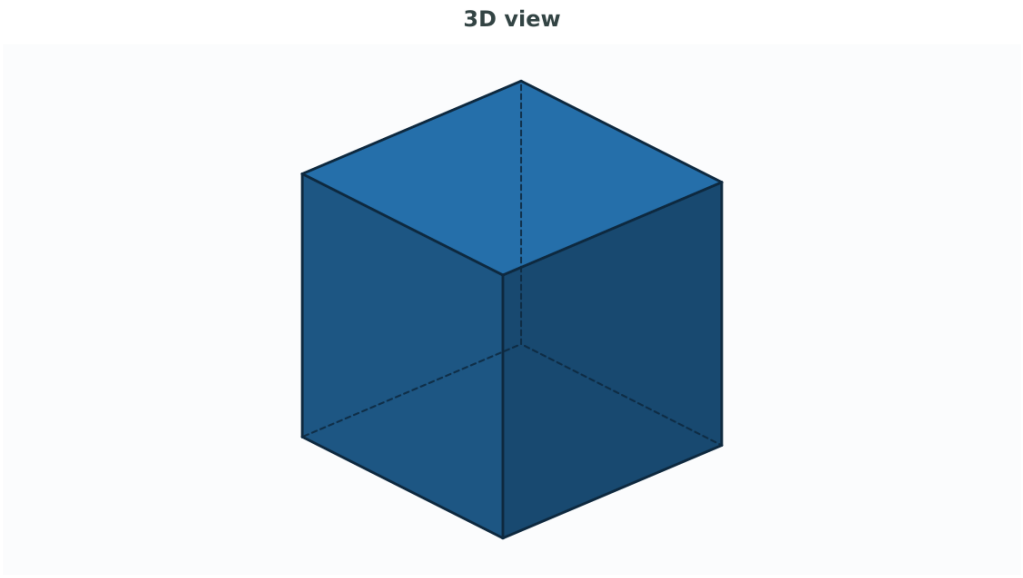
Code

```
SELECT ST_IsClosed('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )');
```

Output

```
t
```

Figure



This example uses the same cube with one side missing.

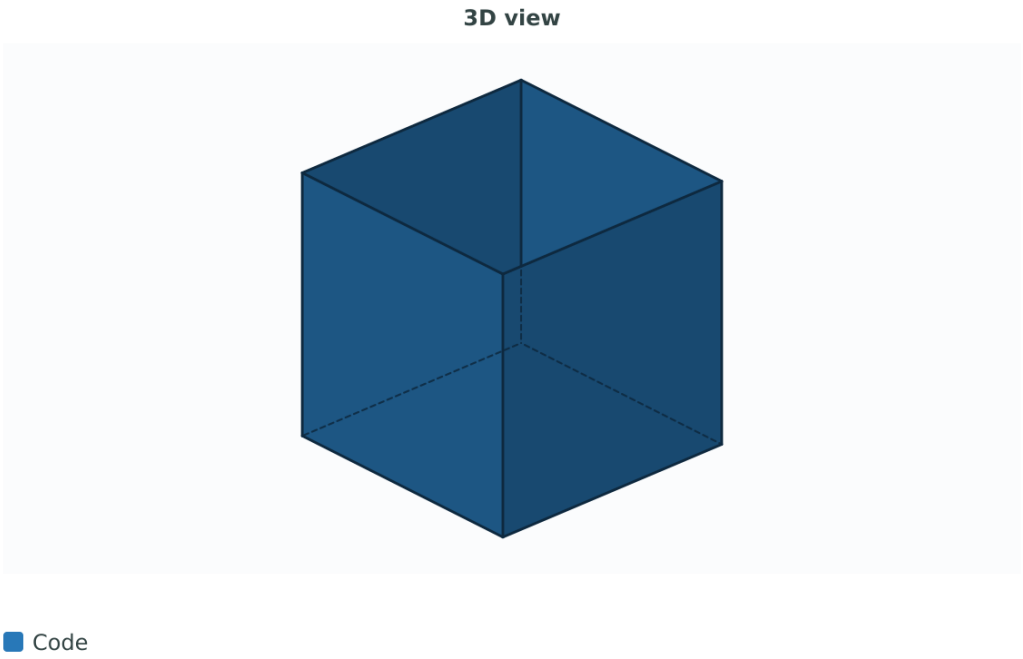
Code

```
SELECT ST_IsClosed('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
       ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
       ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
       ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)) )');
```

Output

```
f
```

Figure



See Also

ST_IsRing

7.4.20 ST_IsCollection

ST_IsCollection — Tests if a geometry is a geometry collection type.

Synopsis

boolean **ST_IsCollection**(geometry g);

Description

Returns `TRUE` if the geometry type of the argument a geometry collection type. Collection types are the following:

- GEOMETRYCOLLECTION
- MULTI{POINT,POLYGON,LINESTRING,CURVE,SURFACE}
- COMPOUNDCURVE



Note
This function analyzes the type of the geometry. This means that it will return `TRUE` on collections that are empty or that contain a single element.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.

Examples

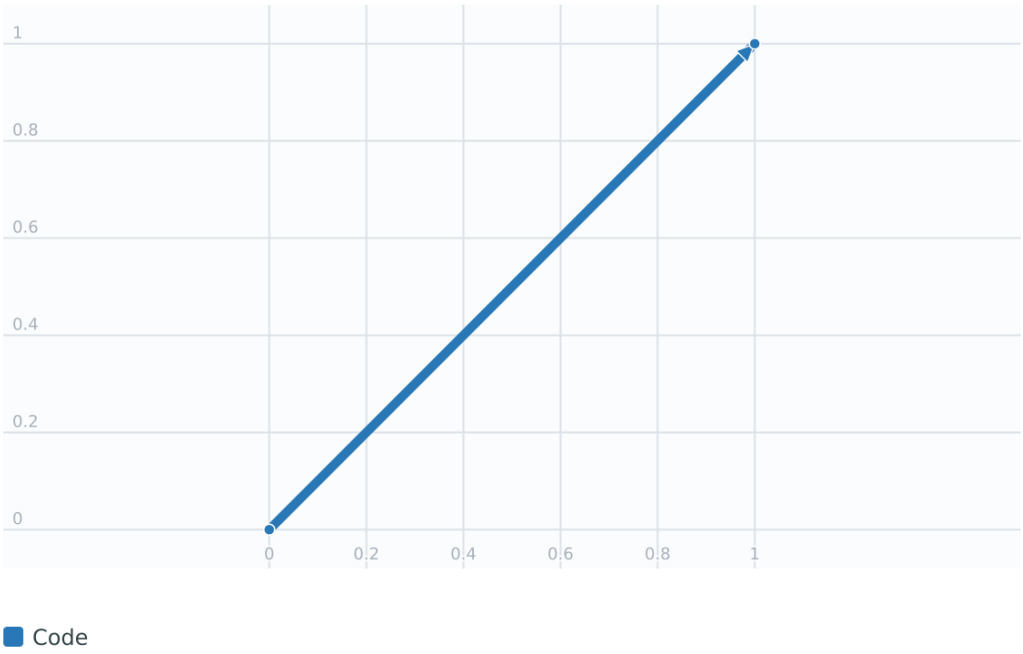
Code

```
SELECT ST_IsCollection('LINESTRING(0 0,1 1)'::geometry);
```

Output

```
f
```

Figure



Code

```
SELECT ST_IsCollection('MULTIPOINT EMPTY'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsCollection('MULTIPOINT((0 0))'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsCollection('MULTIPOINT((0 0),(42 42))'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsCollection('GEOMETRYCOLLECTION(POINT(0 0))'::geometry);
```

Output

```
t
```

See Also

ST_NumGeometries

7.4.21 ST_IsEmpty

ST_IsEmpty — Tests if a geometry is empty.

Synopsis

boolean **ST_IsEmpty**(geometry geomA);

Description

Returns true if this Geometry is an empty geometry. If true, then this Geometry represents an empty geometry collection, polygon, point etc.



Note
SQL-MM defines the result of ST_IsEmpty(NULL) to be 0, while PostGIS returns NULL.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1


This method implements the SQL/MM specification. SQL-MM 3: 5.1.7


This method supports Circular Strings and Curves.

Changed: 2.0.0 In prior versions of PostGIS ST_GeomFromText('GEOMETRYCOLLECTION(EMPTY)') was allowed. This is now illegal in PostGIS 2.0.0 to better conform with SQL/MM standards

Examples

Code

```
SELECT ST_IsEmpty('GEOMETRYCOLLECTION EMPTY'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsEmpty('POLYGON EMPTY'::geometry);
```

Output

```
t
```

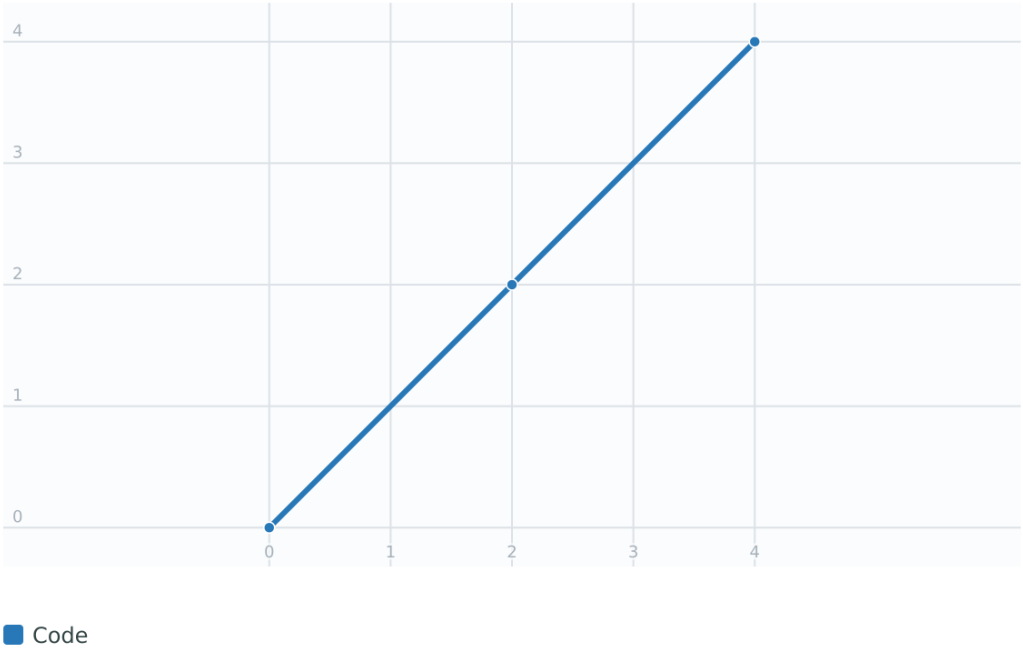
Code

```
SELECT ST_IsEmpty('POLYGON((1 2,3 4,5 6,1 2))'::geometry);
```

Output

```
f
```

Figure



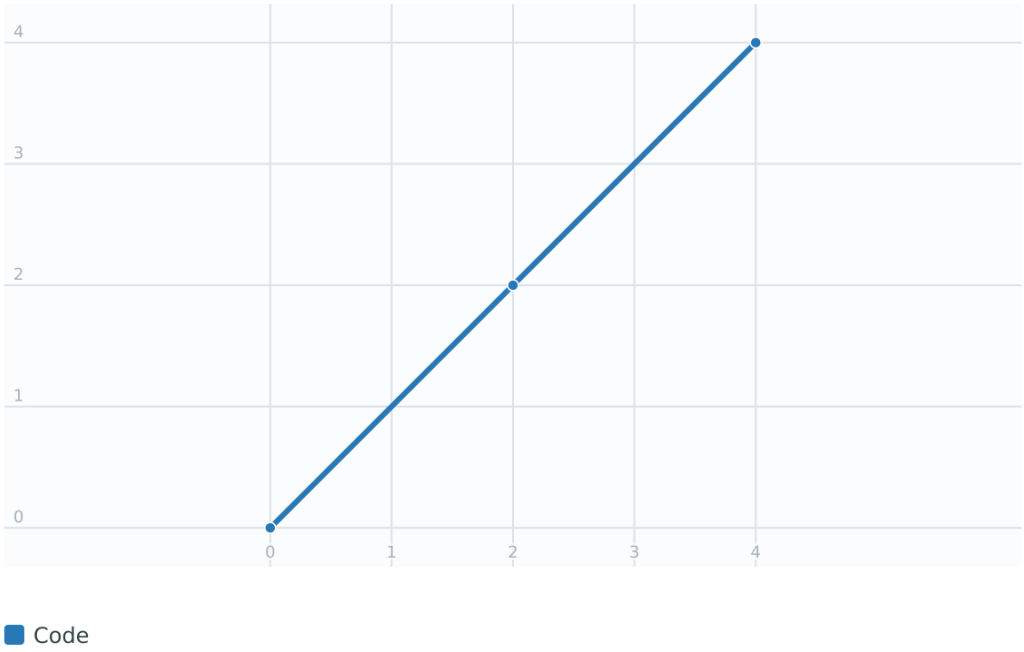
Code

```
SELECT ST_IsEmpty('POLYGON((1 2,3 4,5 6,1 2))'::geometry) = false;
```

Output

```
 t
(1 row)
```

Figure



Code

```
SELECT ST_IsEmpty('CIRCULARSTRING EMPTY'::geometry);
```

Output

```
t
```

7.4.22 ST_IsPolygonCCW

ST_IsPolygonCCW — Tests if Polygons have exterior rings oriented counter-clockwise and interior rings oriented clockwise.

Synopsis

boolean **ST_IsPolygonCCW**(geometry geom);

Description

Returns true if all polygonal components of the input geometry use a counter-clockwise orientation for their exterior ring, and a clockwise direction for all interior rings.

Returns true if the geometry has no polygonal components.



Note
Closed linestrings are not considered polygonal components, so you would still get a true return by passing a single closed linestring no matter its orientation.



Note
If a polygonal geometry does not use reversed orientation for interior rings (i.e., if one or more interior rings are oriented in the same direction as an exterior ring) then both ST_IsPolygonCW and ST_IsPolygonCCW will return false.

Availability: 2.4.0


This function supports 3d and will not drop the z-index.


This function supports M coordinates.

See Also

ST_ForcePolygonCW, ST_ForcePolygonCCW, ST_IsPolygonCW

7.4.23 ST_IsPolygonCW

ST_IsPolygonCW — Tests if Polygons have exterior rings oriented clockwise and interior rings oriented counter-clockwise.

Synopsis

boolean **ST_IsPolygonCW**(geometry geom);

Description

Returns true if all polygonal components of the input geometry use a clockwise orientation for their exterior ring, and a counter-clockwise direction for all interior rings.

Returns true if the geometry has no polygonal components.



Note
Closed linestrings are not considered polygonal components, so you would still get a true return by passing a single closed linestring no matter its orientation.



Note
If a polygonal geometry does not use reversed orientation for interior rings (i.e., if one or more interior rings are oriented in the same direction as an exterior ring) then both ST_IsPolygonCW and ST_IsPolygonCCW will return false.

Availability: 2.4.0


This function supports 3d and will not drop the z-index.


This function supports M coordinates.

See Also

ST_ForcePolygonCW, ST_ForcePolygonCCW, ST_IsPolygonCW

7.4.24 ST_IsRing

ST_IsRing — Tests if a LineString is closed and simple.

Synopsis

boolean **ST_IsRing**(geometry g);

Description

Returns `TRUE` if this `LINESTRING` is both ST_IsClosed (`ST_StartPoint(g) ~= ST_EndPoint(g)`) and ST_IsSimple (does not self intersect).

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. 2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 7.1.6



Note
SQL-MM defines the result of `ST_IsRing(NULL)` to be 0, while PostGIS returns `NULL`.

Examples

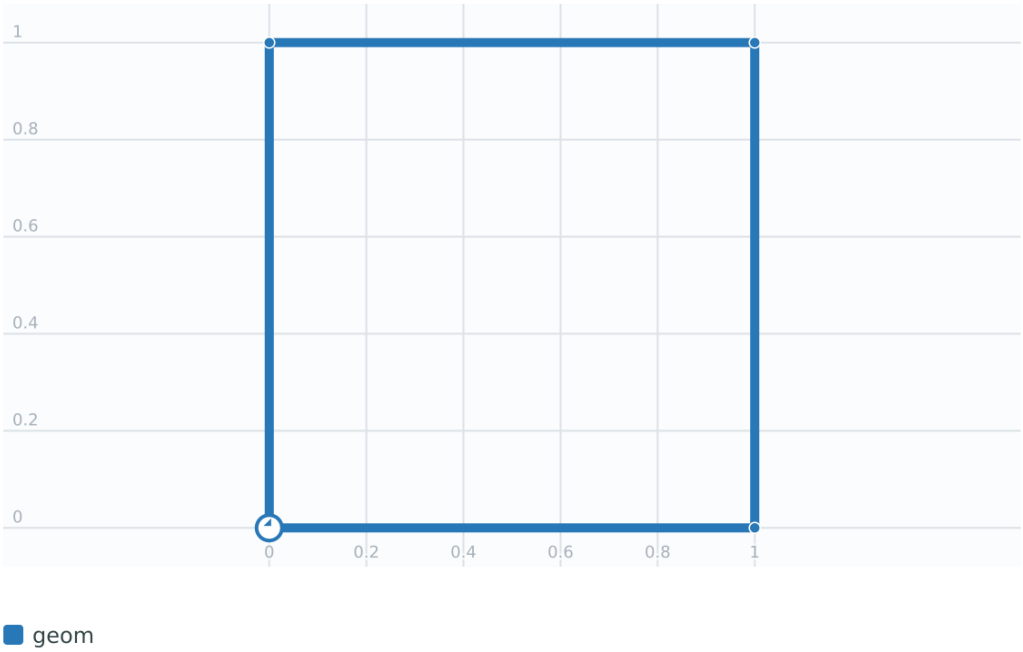
Code

```
SELECT ST_IsRing(geom), ST_IsClosed(geom), ST_IsSimple(geom)
FROM (SELECT 'LINESTRING(0 0,0 1,1 1,1 0,0 0)'::geometry AS geom) AS foo;
```

Output

```
 st_isring | st_isclosed | st_issimple
-----------+-------------+-------------
 t         | t           | t
(1 row)
```

Figure



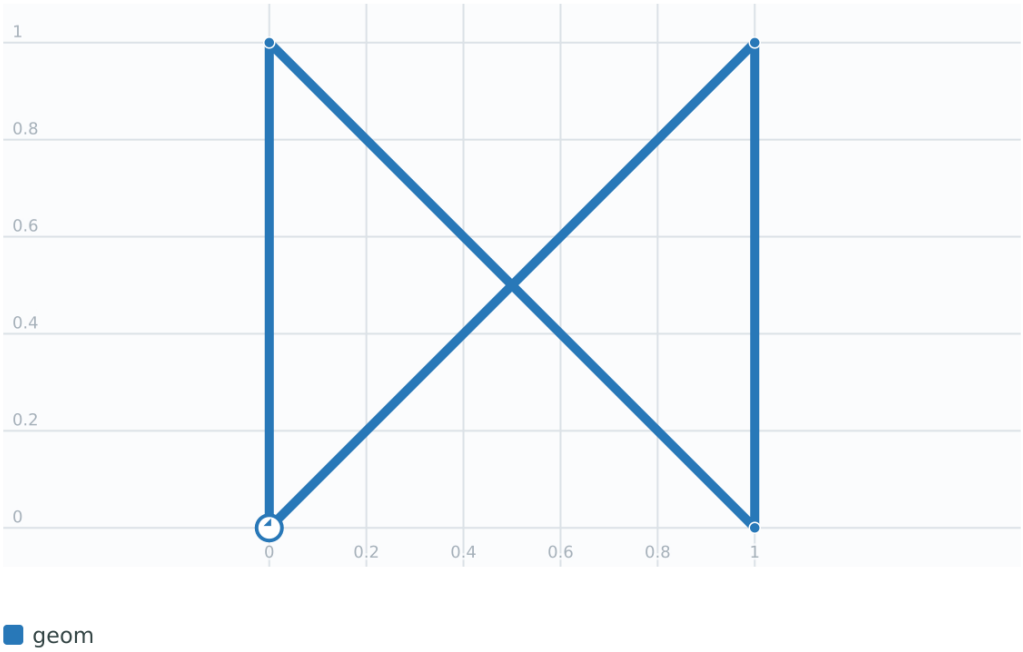
Code

```
SELECT ST_IsRing(geom), ST_IsClosed(geom), ST_IsSimple(geom)
FROM (SELECT 'LINESTRING(0 0,0 1,1 0,1 1,0 0)'::geometry AS geom) AS foo;
```

Output

```
 st_isring | st_isclosed | st_issimple
-----------+-------------+-------------
 f         | t           | f
(1 row)
```

Figure



See Also

ST_IsClosed, ST_IsSimple, ST_StartPoint, ST_EndPoint

7.4.25 ST_IsSimple

ST_IsSimple — Tests if a geometry has no points of self-intersection or self-tangency.

Synopsis

boolean **ST_IsSimple**(geometry geomA);

Description

Returns true if this Geometry has no anomalous geometric points, such as self-intersection or self-tangency. For more information on the OGC's definition of geometry simplicity and validity, refer to "Ensuring OpenGIS compliance of geometries"



Note
SQL-MM defines the result of ST_IsSimple(NULL) to be 0, while PostGIS returns NULL.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1


This method implements the SQL/MM specification. SQL-MM 3: 5.1.8


This function supports 3d and will not drop the z-index.

Examples

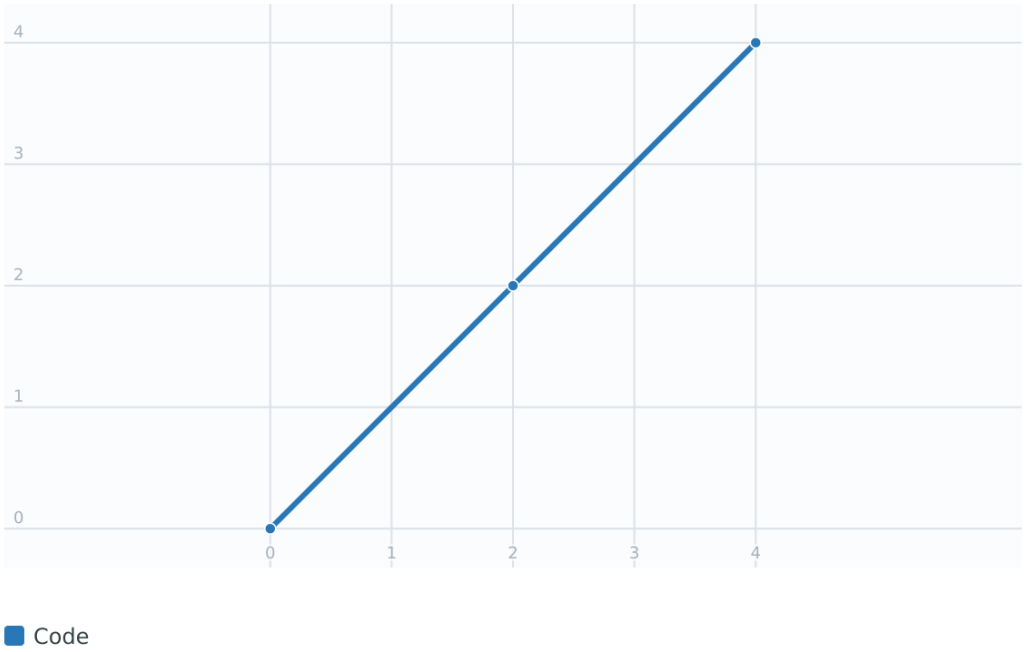
Code

```
SELECT ST_IsSimple('POLYGON((1 2,3 4,5 6,1 2))');
```

Output

```
f
```

Figure



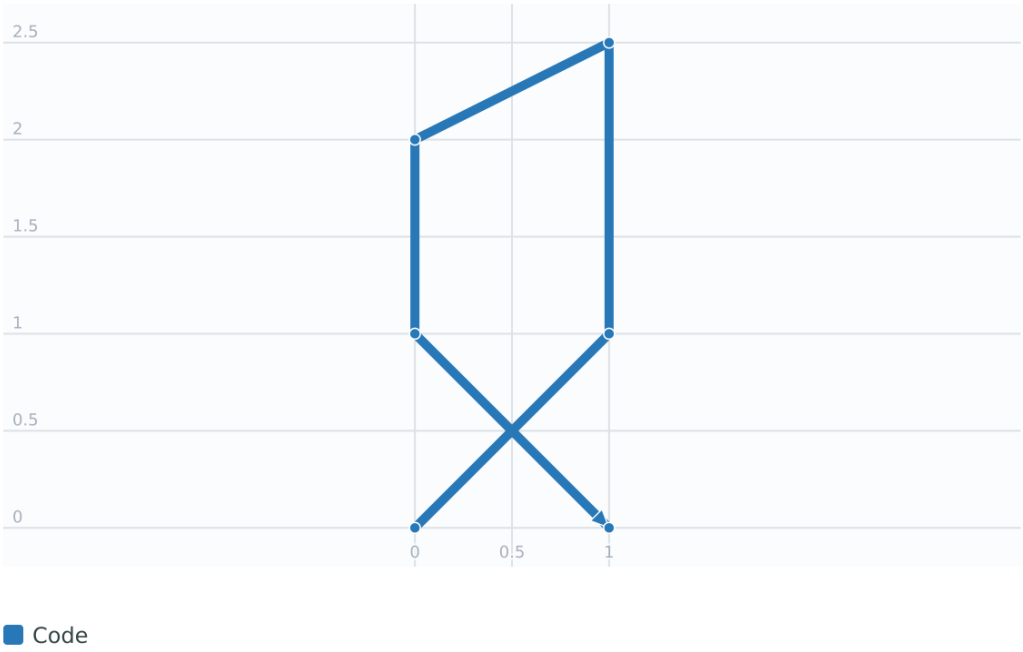
Code

```
SELECT ST_IsSimple('LINESTRING(1 1,2 2,2 3.5,1 3,1 2,2 1)');
```

Output

```
f
```

Figure



See Also

ST_IsValid

7.4.26 ST_M

ST_M — Returns the M coordinate of a Point.

Synopsis

float **ST_M**(geometry a_point);

Description

Return the M coordinate of a Point, or NULL if not available. Input must be a Point.



Note
This is not (yet) part of the OGC spec, but is listed here to complete the point coordinate extractor function list.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

 This method implements the SQL/MM specification.

 This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_M('POINT(1 2 3 4)');
```

Output

```
4
```

See Also

ST_GeomFromEWKT, ST_X, ST_Y, ST_Z

7.4.27 ST_MemSize

ST_MemSize — Returns the amount of memory space a geometry takes.

Synopsis

integer **ST_MemSize**(geometry geomA);

Description

Returns the amount of memory space (in bytes) the geometry takes.

This complements the PostgreSQL built-in database object functions pg_column_size, pg_size_pretty, pg_relation_size, pg_total_relatio



Note

pg_relation_size which gives the byte size of a table may return byte size lower than ST_MemSize. This is because pg_relation_size does not add toasted table contribution and large geometries are stored in TOAST tables.
pg_total_relation_size - includes, the table, the toasted tables, and the indexes.
pg_column_size returns how much space a geometry would take in a column considering compression, so may be lower than ST_MemSize


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Changed: 2.2.0 name changed to ST_MemSize to follow naming convention.

Examples

Compare the memory used by all geometries in a Massachusetts town dataset with the subset for Boston.

Code

```
SELECT pg_size_pretty(sum(ST_MemSize(geom))) AS totgeomsum,
       pg_size_pretty(sum(CASE WHEN town = 'BOSTON' THEN ST_MemSize(geom) ELSE 0 END)) AS ↩
    bossum,
       cast(
           sum(CASE WHEN town = 'BOSTON' THEN ST_MemSize(geom) ELSE 0 END) * 100.0
           / sum(ST_MemSize(geom)) AS numeric(10, 2)
       ) AS perbos
FROM towns;
```

Output

```
 totgeomsum | bossum | perbos
------------+--------+--------
 1522 kB    | 30 kB  |   1.99
(1 row)
```

Return the memory size of a CircularString in bytes.

Code

```
SELECT ST_MemSize('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)'::geometry);
```

Output

```
 st_memsize
------------
         80
(1 row)
```

Calculate the percentage of a table's total size used by geometry data.

Code

```
SELECT pg_total_relation_size('public.neighborhoods') AS fulltable_size,
       sum(ST_MemSize(geom)) AS geomsize,
       sum(ST_MemSize(geom)) * 100.0
           / pg_total_relation_size('public.neighborhoods') AS pergeom
FROM neighborhoods;
```

Output

```
 fulltable_size | geomsize |         pergeom
----------------+----------+-------------------------
         262144 |    96238 | 36.711883544921875000000
(1 row)
```

7.4.28 ST_NDims

ST_NDims — Returns the coordinate dimension of a geometry.

Synopsis

integer **ST_NDims**(geometry g1);

Description

Returns the coordinate dimension of the geometry. PostGIS supports 2 - (x,y) , 3 - (x,y,z) or 2D with measure - x,y,m, and 4 - 3D with measure space x,y,z,m

 This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_NDims('POINT(1 1)') As d2point,
ST_NDims('POINT(1 1 2)') As d3point,
ST_NDims('POINTM(1 1 0.5)') As d2pointm;
```

Output

```
 d2point | d3point | d2pointm
---------+---------+----------
       2 |       3 |        3
```

See Also

ST_CoordDim, ST_Dimension, ST_GeomFromEWKT

7.4.29 ST_NPoints

ST_NPoints — Returns the number of points (vertices) in a geometry.

Synopsis

integer **ST_NPoints**(geometry g1);

Description

Return the number of points in a geometry. Works for all geometries.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.



Note
Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

Examples

Count the points in a 2D LineString.

Code

```
SELECT ST_NPoints('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)');
```

Output

```
 st_npoints
------------
          4
(1 row)
```

Count the points in a 3D LineString.

Code

```
SELECT ST_NPoints('LINESTRING(77.29 29.07 1,77.42 29.26 0,77.27 29.31 -1,77.29 29.07 3)');
```

Output

```
 st_npoints
------------
          4
(1 row)
```

See Also

ST_NumPoints

7.4.30 ST_NRings

ST_NRings — Returns the number of rings in a polygonal geometry.

Synopsis

integer **ST_NRings**(geometry geomA);

Description

If the geometry is a polygon or multi-polygon returns the number of rings. Unlike NumInteriorRings, it counts the outer rings as well.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

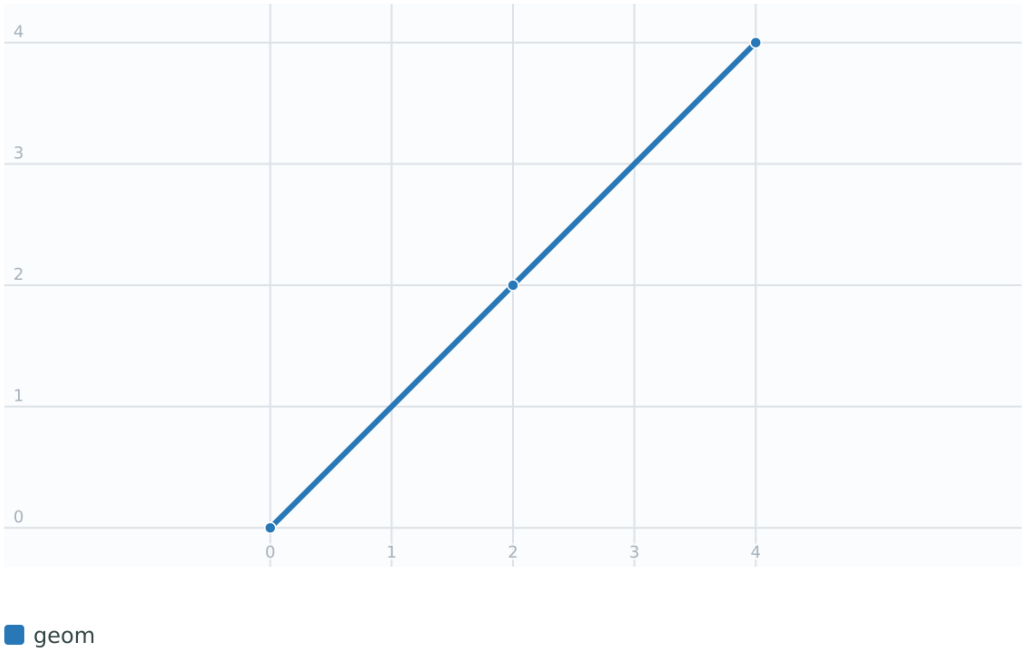
Code

```
SELECT ST_NRings(geom) As Nrings, ST_NumInteriorRings(geom) As ninterrings
FROM (SELECT 'POLYGON((1 2,3 4,5 6,1 2))'::geometry As geom) As foo;
```

Output

```
 nrings | ninterrings
--------+-------------
      1 |           0
(1 row)
```

Figure



See Also

ST_NumInteriorRings

7.4.31 ST_NumGeometries

ST_NumGeometries — Returns the number of elements in a geometry collection.

Synopsis

integer **ST_NumGeometries**(geometry geom);

Description

Returns the number of elements in a geometry collection (GEOMETRYCOLLECTION or MULTI*). For non-empty atomic geometries returns 1. For empty geometries returns 0.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Changed: 2.0.0 In prior versions this would return NULL if the geometry was not a collection/MULTI type. 2.0.0+ now returns 1 for single geometries e.g POLYGON, LINESTRING, POINT.

Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and return 1; use ST_NumPatches to count faces.

This method implements the SQL/MM specification. SQL-MM 3: 9.1.4

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

This example returns 1; prior versions would have returned NULL.

Code

```
SELECT ST_NumGeometries('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)');
```

Output

```
 st_numgeometries
------------------
                1
(1 row)
```

A MultiPoint counts as one geometry within a GeometryCollection.

Code

```
SELECT ST_NumGeometries('GEOMETRYCOLLECTION(MULTIPOINT((-2 3),(-2 2)),
LINESTRING(5 5,10 10),
POLYGON((-7 4.2,-7.1 5,-7.1 4.3,-7 4.2)))');
```

Output

```
 st_numgeometries
------------------
                3
(1 row)
```

See Also

ST_GeometryN, ST_Multi

7.4.32 ST_NumInteriorRings

ST_NumInteriorRings — Returns the number of interior rings (holes) of a Polygon.

Synopsis

integer **ST_NumInteriorRings**(geometry a_polygon);

Description

Return the number of interior rings of a polygon geometry. Return NULL if the geometry is not a polygon.

 This method implements the SQL/MM specification. SQL-MM 3: 8.2.5

Changed: 2.0.0 - in prior versions it would allow passing a MULTIPOLYGON, returning the number of interior rings of first POLYGON.

Examples

Count the interior rings of a Polygon.

Code

```
SELECT gid, field1, field2, ST_NumInteriorRings(geom) AS numholes
FROM sometable;
```

For a MultiPolygon, dump its component Polygons and sum their interior ring counts.

Code

```
SELECT gid, field1, field2, SUM(ST_NumInteriorRings(geom)) AS numholes
FROM (
    SELECT gid, field1, field2, (ST_Dump(geom)).geom AS geom
    FROM sometable
) AS foo
GROUP BY gid, field1, field2;
```

See Also

ST_NumInteriorRing, ST_InteriorRingN

7.4.33 ST_NumInteriorRing

ST_NumInteriorRing — Returns the number of interior rings (holes) of a Polygon. Alias for ST_NumInteriorRings.

Synopsis

integer **ST_NumInteriorRing**(geometry a_polygon);

Description

This function is an alias for ST_NumInteriorRings.

Availability: 1.2.2

See Also

ST_NumInteriorRings, ST_InteriorRingN

7.4.34 ST_NumPatches

ST_NumPatches — Returns the number of faces in a PolyhedralSurface or TIN.

Synopsis

integer **ST_NumPatches**(geometry g1);

Description

Returns the number of faces in a PolyhedralSurface or TIN. Returns NULL for other geometries. Unlike ST_NumGeometries, this function counts individual faces.

Availability: 2.0.0

This function supports 3d and will not drop the z-index.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3: 8.5

This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_NumPatches('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )');
```

Output

```
 st_numpatches
---------------
             6
(1 row)
```

See Also

ST_GeomFromEWKT, ST_NumGeometries, ST_PatchN

7.4.35 ST_NumPoints

ST_NumPoints — Returns the number of points in a LineString or CircularString.

Synopsis

integer **ST_NumPoints**(geometry g1);

Description

Return the number of points in an ST_LineString or ST_CircularString value. Prior to 1.4 only works with linestrings as the specs state. From 1.4 forward this is an alias for ST_NPoints which returns number of vertices for not just linestrings. Consider using ST_NPoints instead which is multi-purpose and works with many geometry types.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 7.2.4

Examples

Code

```
SELECT ST_NumPoints('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)');
```

Output

```
 st_numpoints
--------------
            4
(1 row)
```

See Also

ST_NPoints

7.4.36 ST_PatchN

ST_PatchN — Returns the Nth face of a PolyhedralSurface or TIN.

Synopsis

geometry **ST_PatchN**(geometry geomA, integer n);

Description

Returns the 1-based Nth face of a PolyhedralSurface or TIN. Returns NULL for other geometries or an out-of-range index. Use this function to access faces, because ST_GeometryN treats PolyhedralSurface and TIN as unitary geometries.



Note
Index is 1-based.



Note
If you want to extract all elements of a geometry ST_Dump is more efficient.

Availability: 2.0.0


This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3: 8.5


This function supports 3d and will not drop the z-index.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Extract the 2nd face of the polyhedral surface.

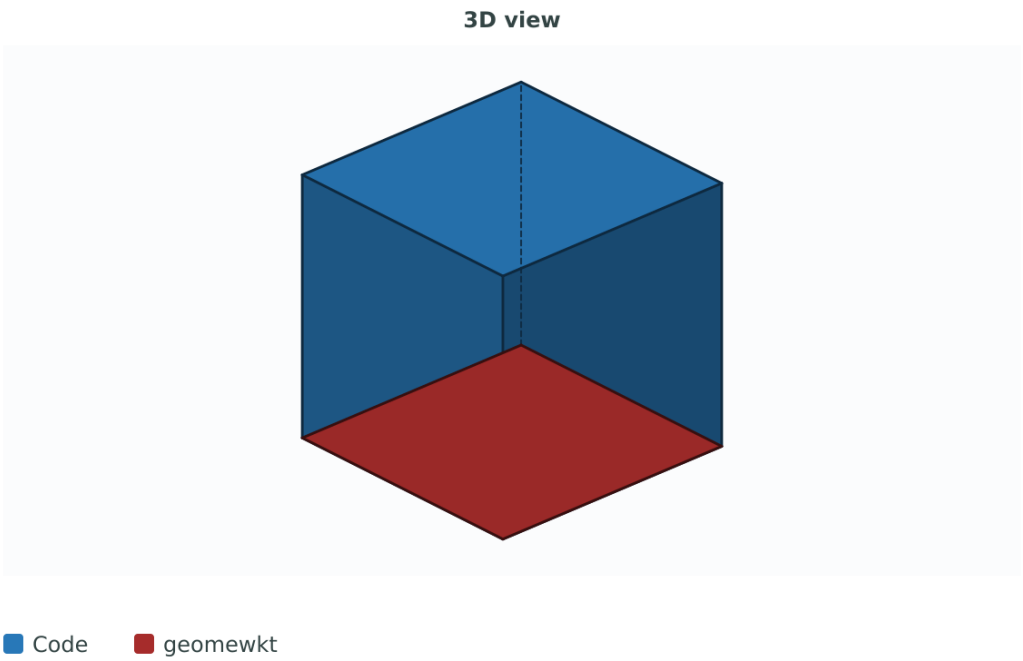
Code

```
SELECT ST_PatchN(geom, 2) As geomewkt
FROM (
VALUES ('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry) ) As ↩
    foo(geom);
```

Output

```
POLYGON((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0))
```

Figure



See Also

ST_AsEWKT, ST_GeomFromEWKT, ST_Dump, ST_GeometryN, ST_NumGeometries, ST_NumPatches

7.4.37 ST_PointN

ST_PointN — Returns the Nth point in the first LineString or circular LineString in a geometry.

Synopsis

geometry **ST_PointN**(geometry a_linestring, integer n);

Description

Return the Nth point in a single linestring or circular linestring in the geometry. Negative values are counted backwards from the end of the LineString, so that -1 is the last point. Returns NULL if there is no linestring in the geometry.



Note
Index is 1-based as for OGC specs since version 0.8.0. Backward indexing (negative index) is not in OGC. Previous versions implemented this as 0-based instead.



Note
If you want to get the Nth point of each LineString in a MultiLineString, use in conjunction with ST_Dump


This method implements the OGC Simple Features Implementation Specification for SQL 1.1.


This method implements the SQL/MM specification. SQL-MM 3: 7.2.5, 7.3.5


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.

Changed: 2.0.0 no longer works with single geometry multilinestrings. In older versions of PostGIS -- a single line multilinestring would work happily with this function and return the start point. In 2.0.0 it just returns NULL like any other multilinestring.

Changed: 2.3.0 : negative indexing available (-1 is last point)

Examples

This example extracts all POINT geometries from a LineString.

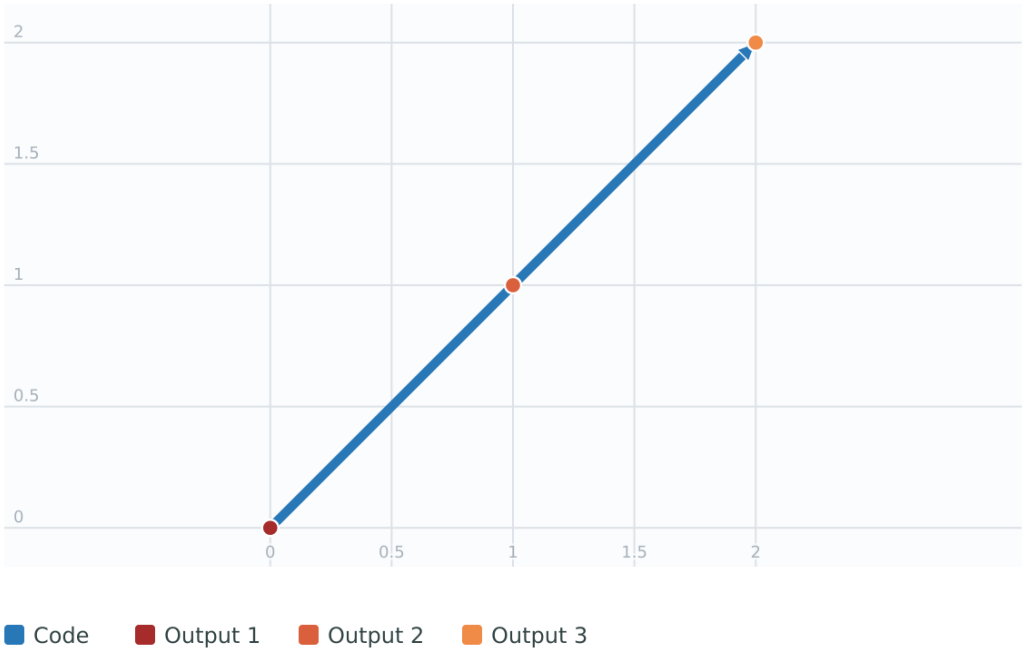
Code

```
SELECT ST_PointN(
      column1,
      generate_series(1, ST_NPoints(column1))
   )
FROM ( VALUES ('LINESTRING(0 0,1 1,2 2)'::geometry) ) AS foo;
```

Output

```
 POINT(0 0)
 POINT(1 1)
 POINT(2 2)
(3 rows)
```

Figure



This example uses a CircularString.

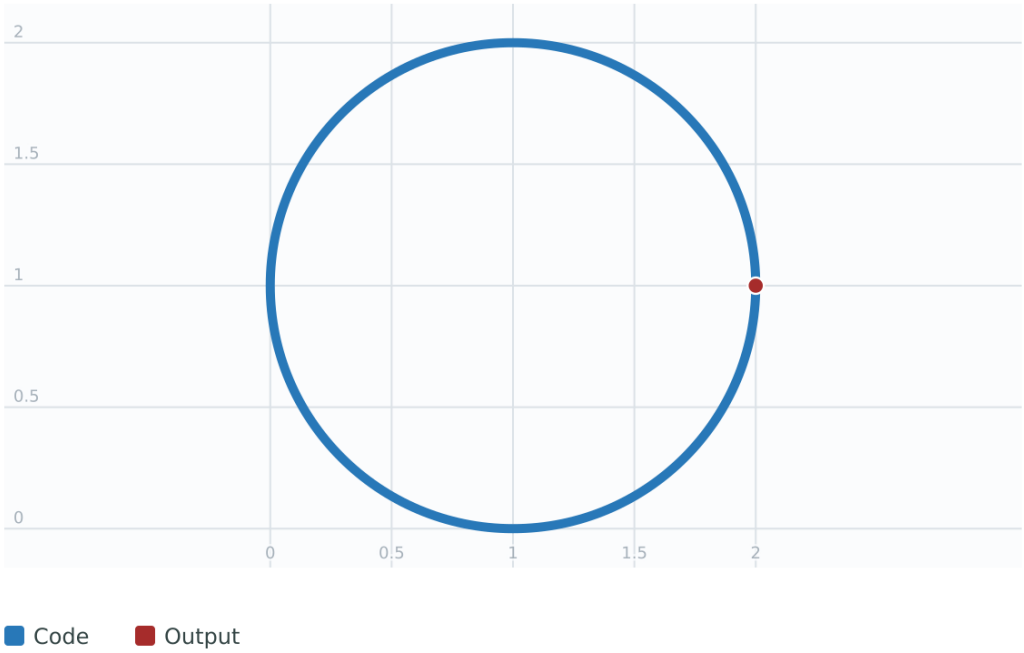
Code

```
SELECT ST_PointN('CIRCULARSTRING(1 2,3 2,1 2)', 2);
```

Output

```
POINT(3 2)
```

Figure



The index is 1-based. A negative index counts backward from the end of the LineString.

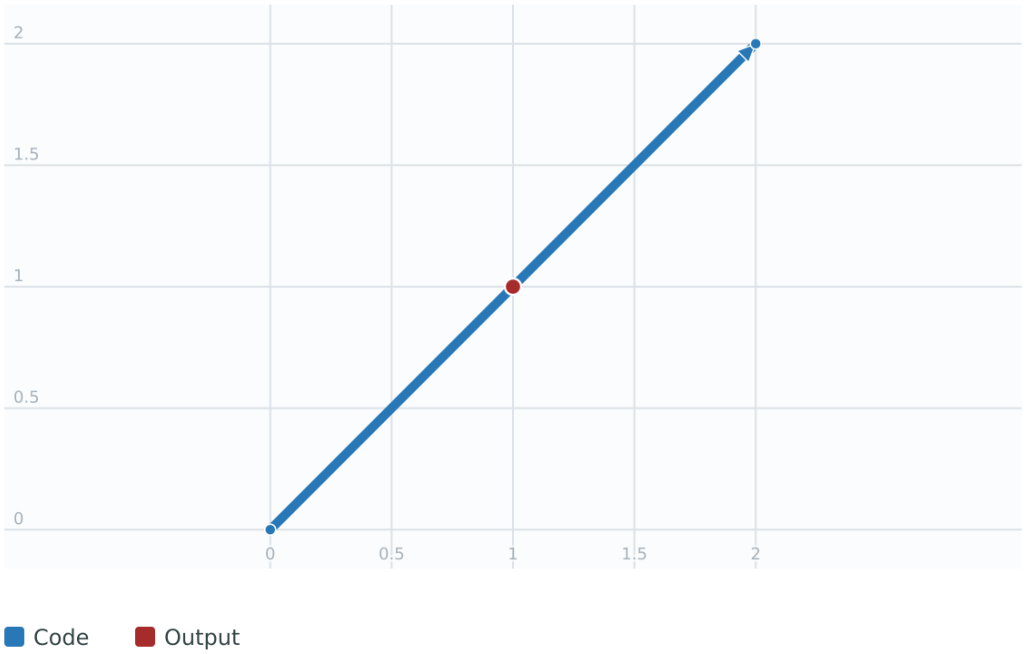
Code

```
SELECT f
FROM ST_GeomFromText('LINESTRING(0 0 0,1 1 1,2 2 2)') AS g
  CROSS JOIN ST_PointN(g, -2) AS f;
```

Output

```
POINT Z (1 1 1)
```

Figure



See Also

ST_NPoints

7.4.38 ST_Points

ST_Points — Returns a MultiPoint containing the coordinates of a geometry.

Synopsis

geometry **ST_Points**(geometry geom);

Description

Returns a MultiPoint containing all the coordinates of a geometry. Duplicate points are preserved, including the start and end points of ring geometries. (If desired, duplicate points can be removed by calling ST_RemoveRepeatedPoints on the result).

To obtain information about the position of each coordinate in the parent geometry use ST_DumpPoints.

M and Z coordinates are preserved if present.

 This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

Availability: 2.3.0

Examples

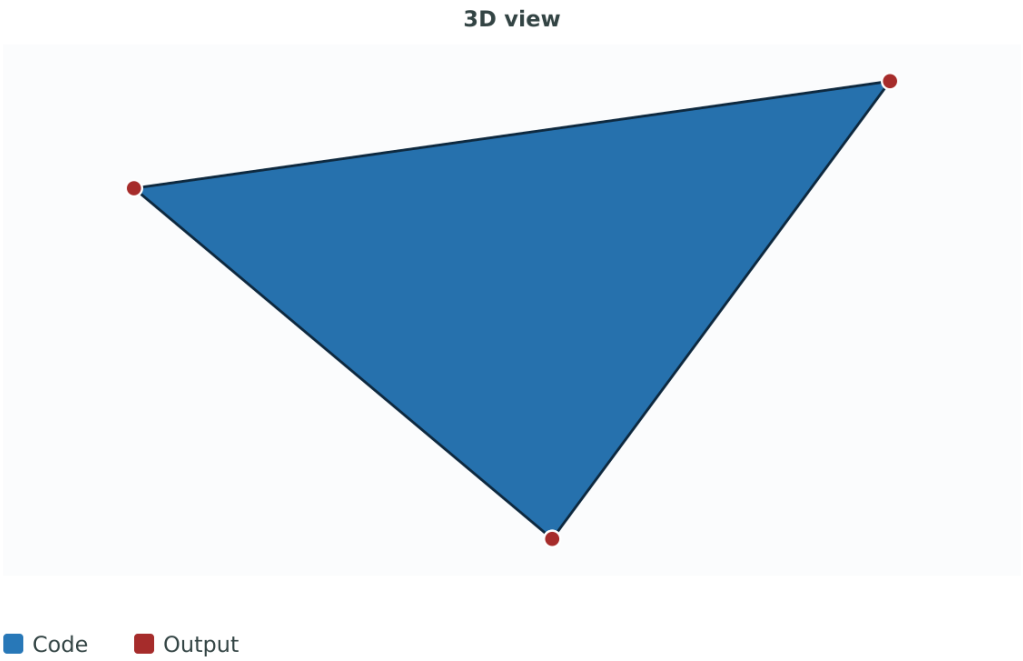
Code

```
SELECT ST_Points('POLYGON Z ((30 10 4,10 30 5,40 40 6,30 10 4))');
```

Output

```
MULTIPOINT Z ((30 10 4),(10 30 5),(40 40 6),(30 10 4))
```

Figure



See Also

ST_RemoveRepeatedPoints, ST_DumpPoints

7.4.39 ST_StartPoint

ST_StartPoint — Returns the first point of a LineString, CircularLineString, or NURBSCurve.

Synopsis

geometry **ST_StartPoint**(geometry geomA);

Description

Returns the first point of a LINESTRING, CIRCULARLINESTRING, or NURBSCURVE geometry as a POINT. For other geometries, returns the first point in coordinate order.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.3

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Enhanced: 3.2.0 returns a point for all geometries. Prior behavior returns NULLs if input was not a LineString.

Changed: 2.0.0 no longer works with single geometry MultiLineStrings. In older versions of PostGIS a single-line MultiLineString would work happily with this function and return the start point. In 2.0.0 it just returns NULL like any other MultiLineString. The old behavior was an undocumented feature, but people who assumed they had their data stored as LINESTRING may experience these returning NULL in 2.0.0.

Examples

Start point of a LineString

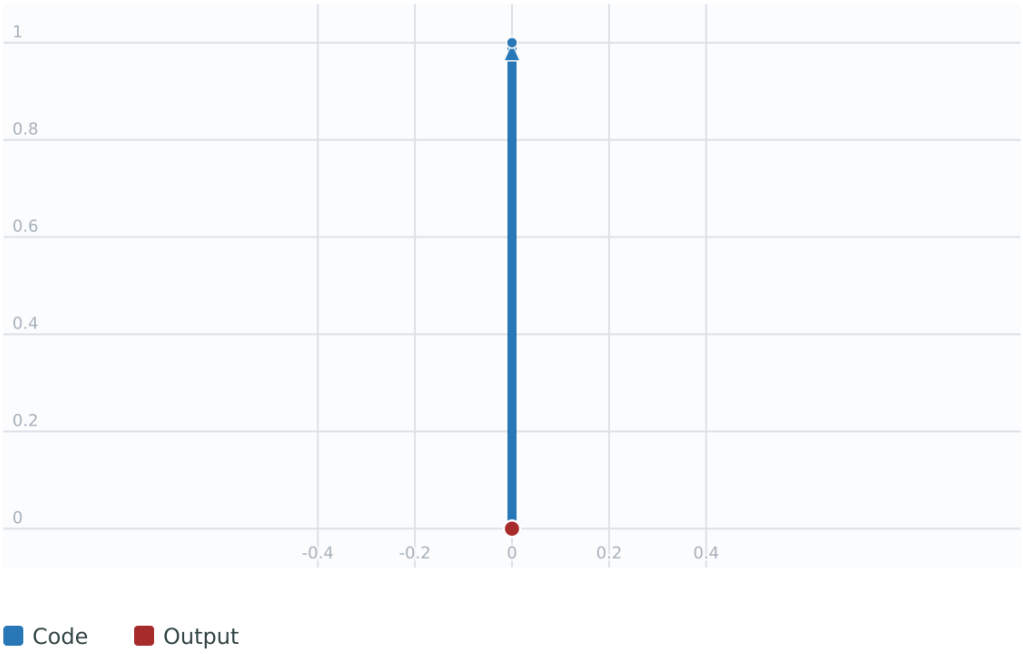
Code

```
SELECT ST_StartPoint('LINESTRING(0 1,0 2)'::geometry);
```

Output

```
POINT(0 1)
```

Figure



Start point of a Point is the Point itself

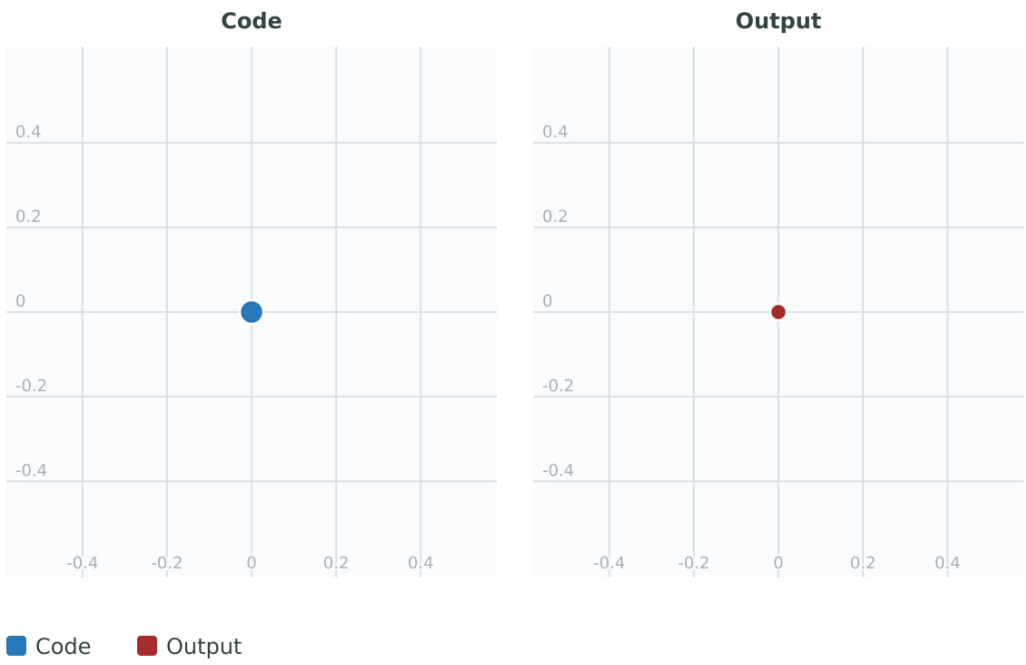
Code

```
SELECT ST_StartPoint('POINT(0 1)'::geometry);
```

Output

```
POINT(0 1)
```

Figure



Start point of a 3D LineString

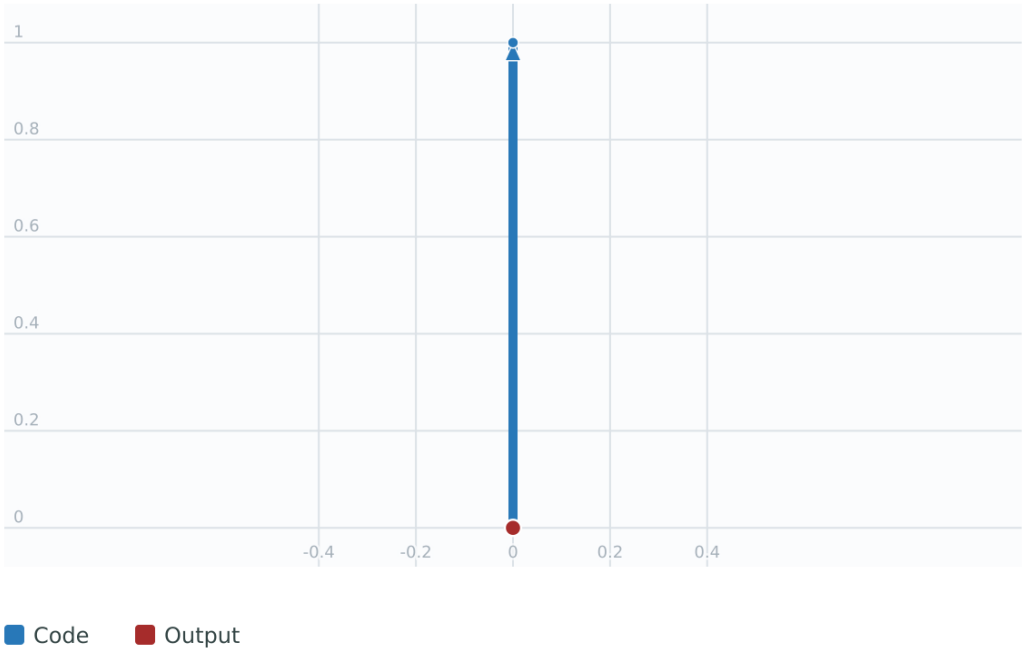
Code

```
SELECT ST_StartPoint('LINESTRING(0 1 1,0 2 2)'::geometry);
```

Output

```
POINT(0 1 1)
```

Figure



Start point of a CircularString

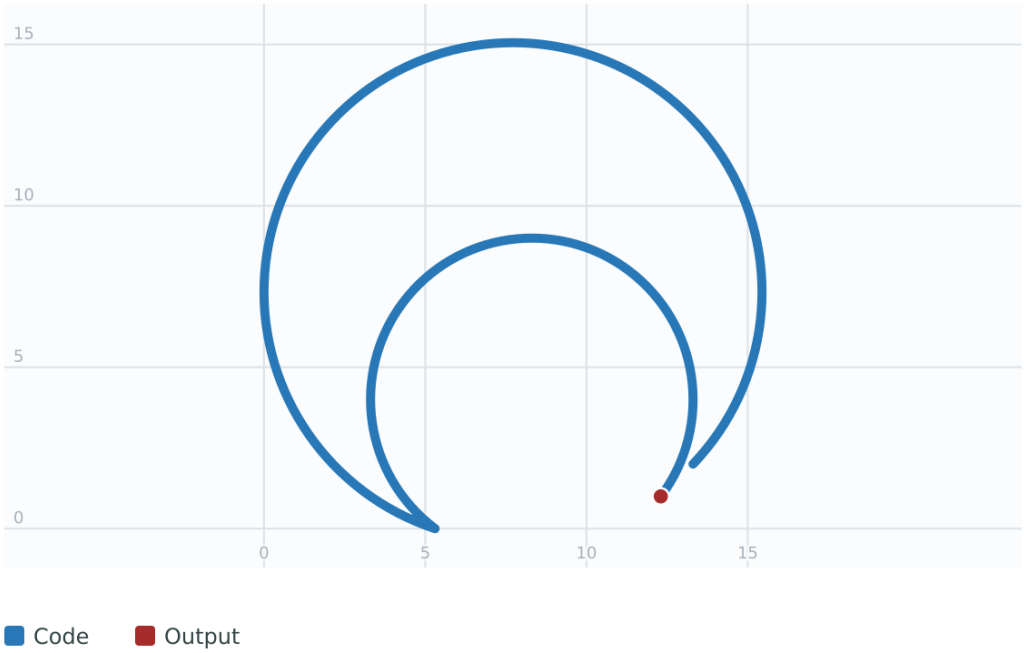
Code

```
SELECT ST_StartPoint('CIRCULARSTRING(5 2,-3 2,-2 1,-4 2,6 3)'::geometry);
```

Output

```
POINT(5 2)
```

Figure



Start point of a NURBSCurve

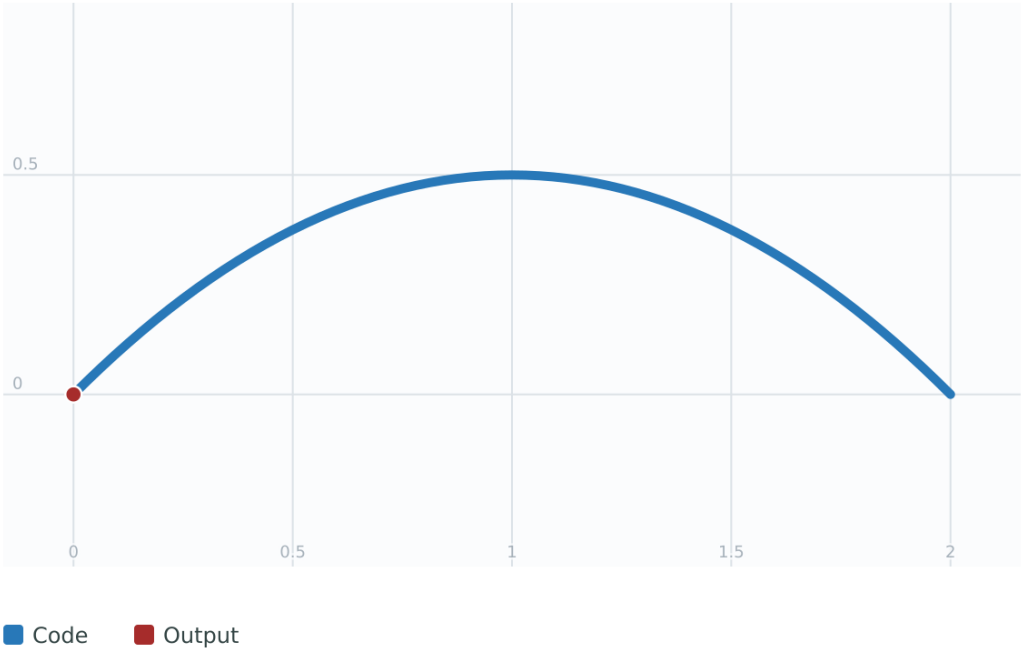
Code

```
SELECT ST_StartPoint('NURBSCURVE(2, (0 0, 1 1, 2 0))'::geometry);
```

Output

```
POINT(0 0)
```

Figure



Build edge node references from the start and end points of a LineString table. The example uses exact endpoint equality, so linework with nearly coincident endpoints should be snapped or otherwise cleaned first.

Code

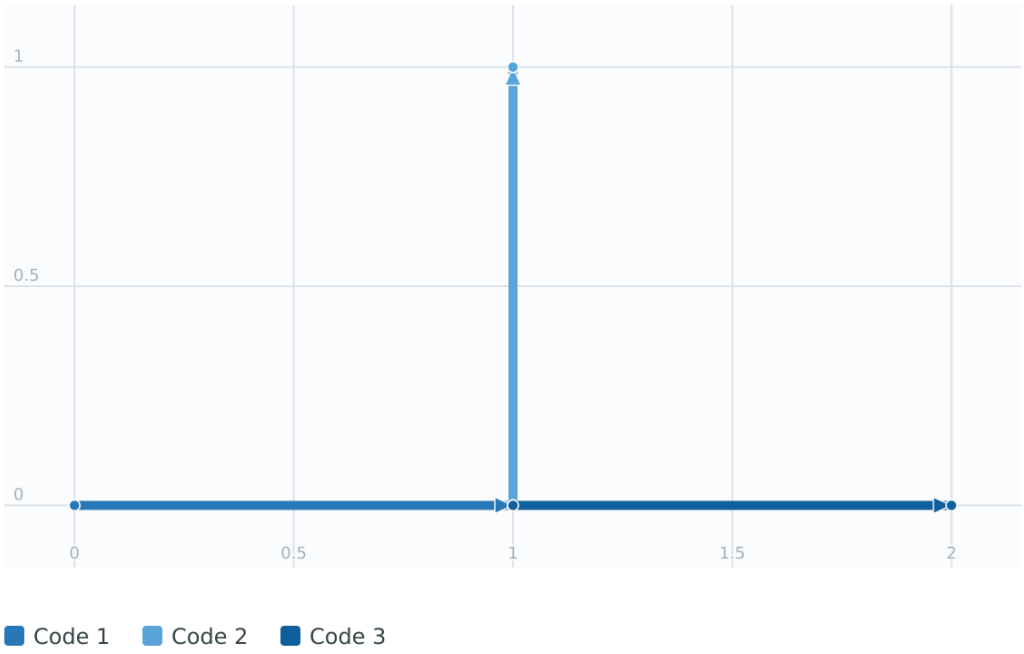
```
WITH roads(road_id, geom) AS (
  VALUES
    (1, 'LINESTRING(0 0, 1 0)'::geometry),
    (2, 'LINESTRING(1 0, 1 1)'::geometry),
    (3, 'LINESTRING(1 0, 2 0)'::geometry)
),
endpoints AS (
  SELECT road_id, 'from' AS end_name, ST_StartPoint(geom) AS geom,
         ST_AsEWKB(ST_StartPoint(geom)) AS endpoint_key
  FROM roads
  UNION ALL
  SELECT road_id, 'to' AS end_name, ST_EndPoint(geom) AS geom,
         ST_AsEWKB(ST_EndPoint(geom)) AS endpoint_key
  FROM roads
),
nodes AS (
  SELECT dense_rank() OVER (ORDER BY ST_X(geom), ST_Y(geom), endpoint_key) AS node_id,
         endpoint_key
  FROM (
    SELECT DISTINCT ON (endpoint_key) endpoint_key, geom
    FROM endpoints
    ORDER BY endpoint_key
  ) AS distinct_endpoints
```

```
)
SELECT e.road_id,
       max(n.node_id) FILTER (WHERE e.end_name = 'from') AS from_node,
       max(n.node_id) FILTER (WHERE e.end_name = 'to') AS to_node
FROM endpoints AS e
JOIN nodes AS n USING (endpoint_key)
GROUP BY e.road_id
ORDER BY e.road_id;
```

Output

```
 road_id | from_node | to_node
---------+-----------+---------
       1 |         1 |       2
       2 |         2 |       3
       3 |         2 |       4
```

Figure



See Also

ST_EndPoint, ST_PointN

7.4.40 ST_StartM

ST_StartM — Returns the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.

Synopsis

float8 **ST_StartM**(geometry geom);

Description

Returns the M coordinate of the first point of a LINESTRING, CIRCULARLINESTRING, or NURBSCURVE geometry. Returns NULL if the input geometry does not have an M dimension or is not a LINESTRING, CIRCULARLINESTRING, or NURBSCURVE.

This method supports Circular Strings and Curves.

Examples

Start M value of a LineString

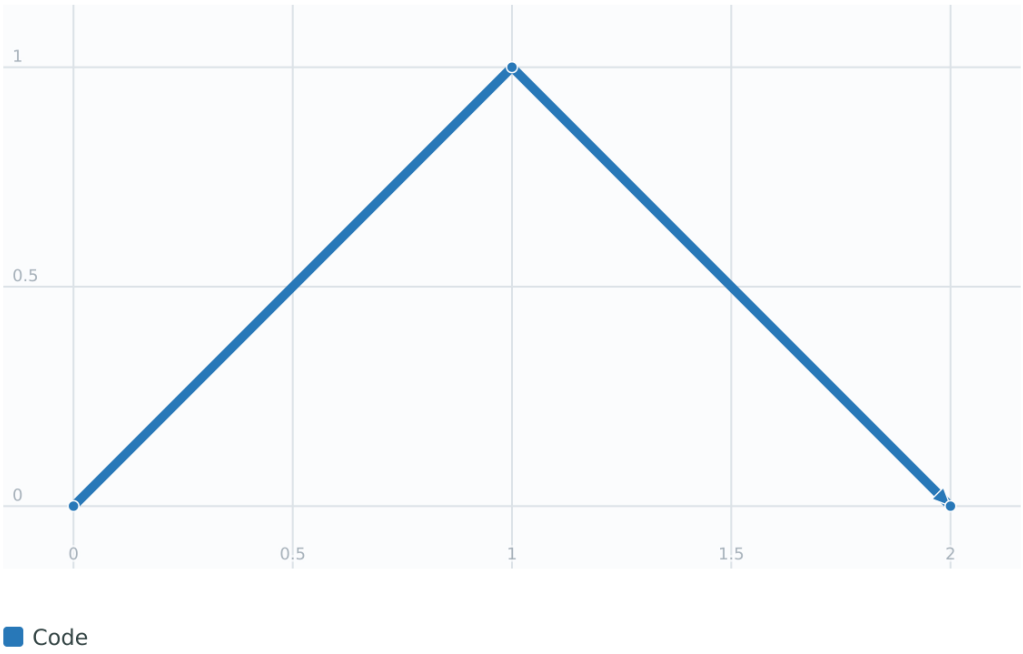
Code

```
SELECT ST_StartM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry);
```

Output

```
10
```

Figure



Start M value of a CircularString

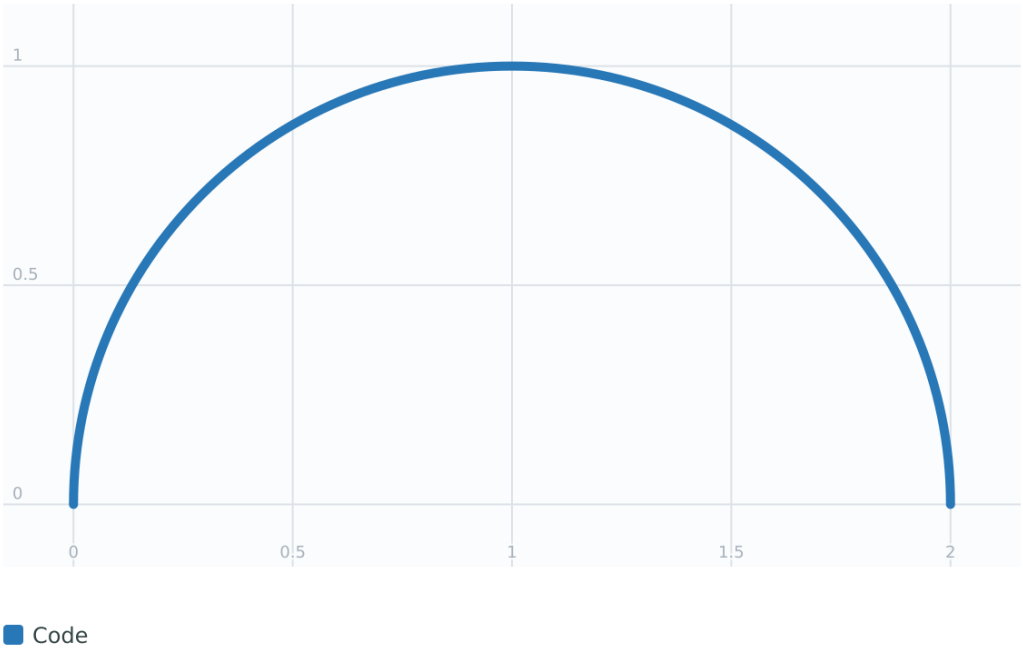
Code

```
SELECT ST_StartM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry);
```

Output

```
5
```

Figure



Start M value of a NURBSCurve

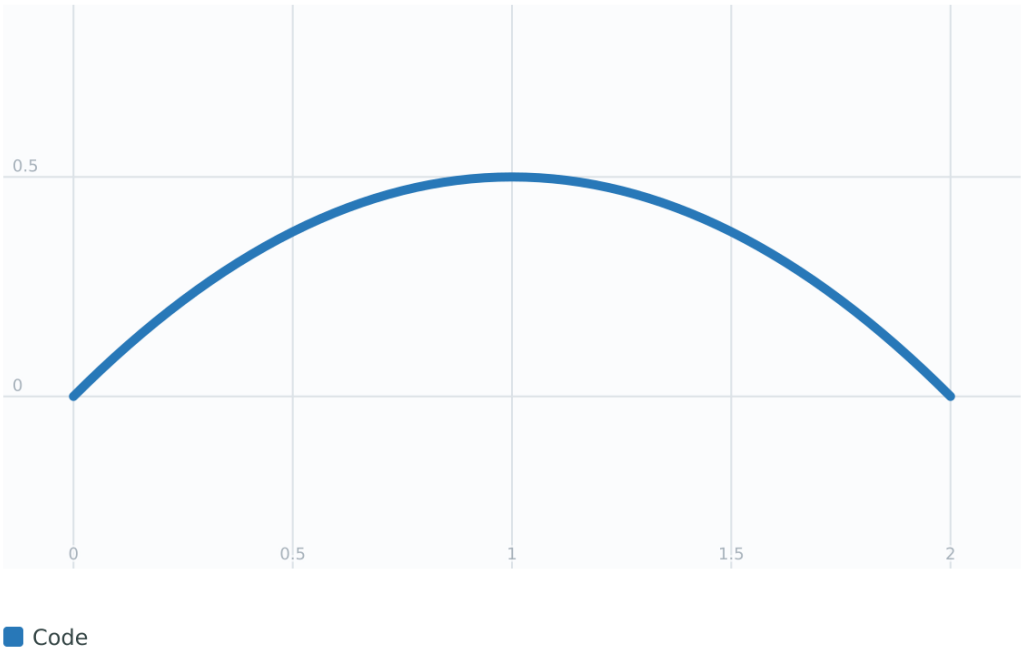
Code

```
SELECT ST_StartM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry);
```

Output

```
100
```

Figure



Geometry without M dimension returns NULL

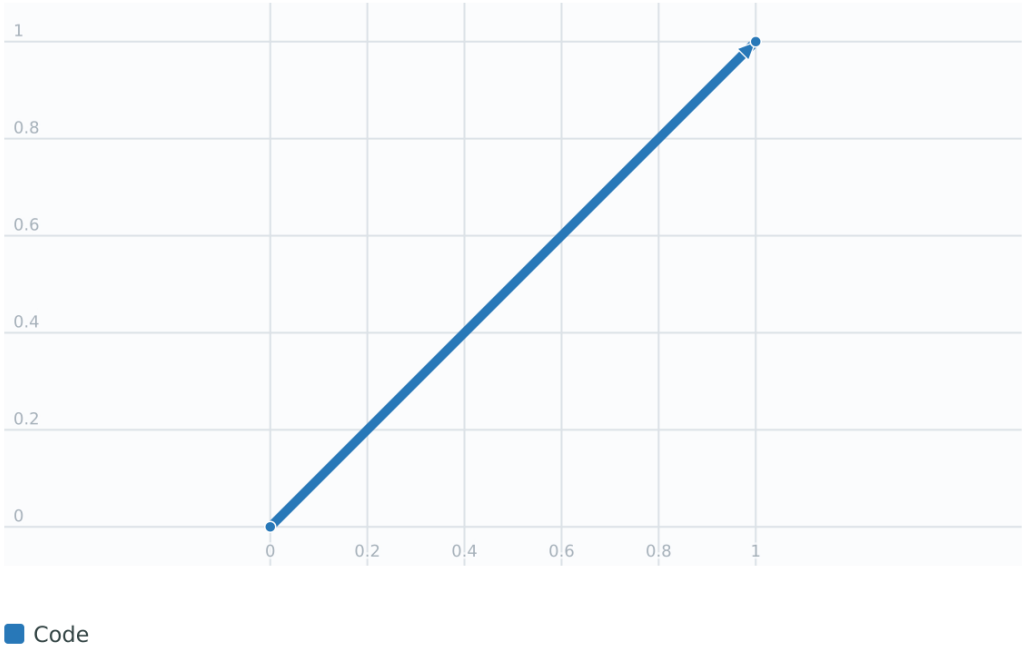
Code

```
SELECT ST_StartM('LINESTRING(0 0,1 1)'::geometry) IS NULL AS is_null;
```

Output

```
t
```

Figure



See Also

ST_EndM, ST_SetStartM, ST_StartPoint

7.4.41 ST_EndM

ST_EndM — Returns the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.

Synopsis

float8 **ST_EndM**(geometry geom);

Description

Returns the M coordinate of the last point of a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry. Returns `NULL` if the input geometry does not have an M dimension or is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

 This method supports Circular Strings and Curves.

Examples

End M value of a LineString

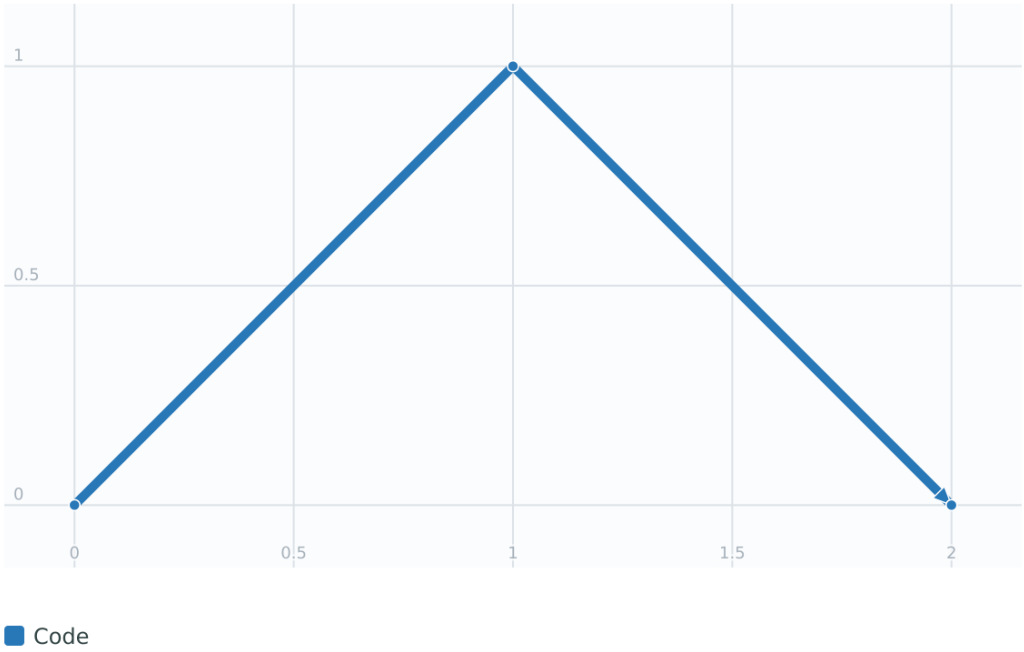
Code

```
SELECT ST_EndM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry);
```

Output

```
30
```

Figure



End M value of a CircularString

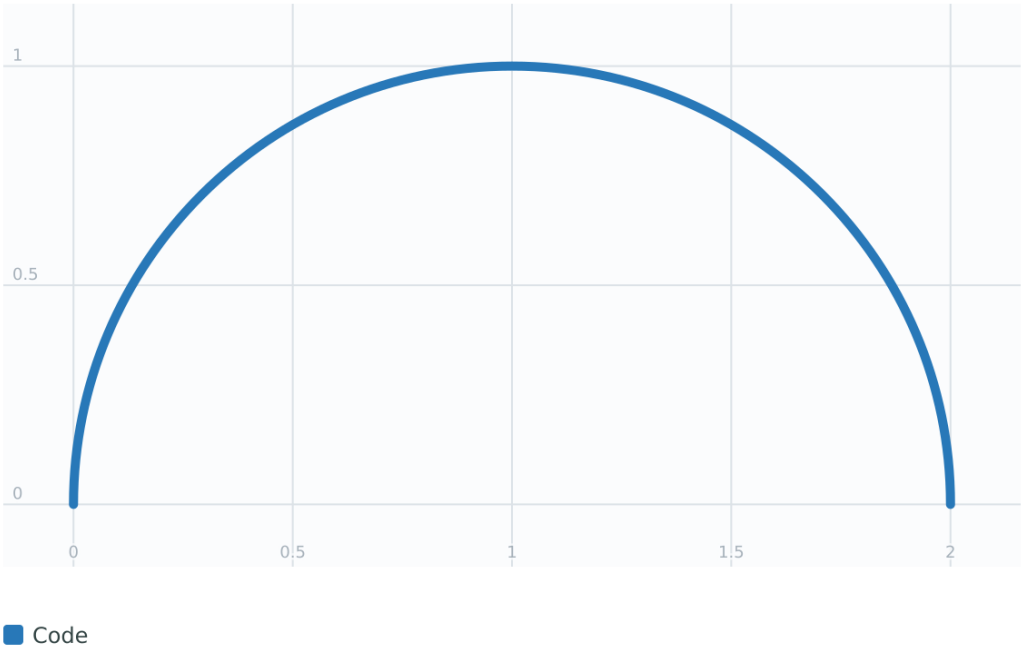
Code

```
SELECT ST_EndM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry);
```

Output

```
15
```

Figure



End M value of a NURBSCurve

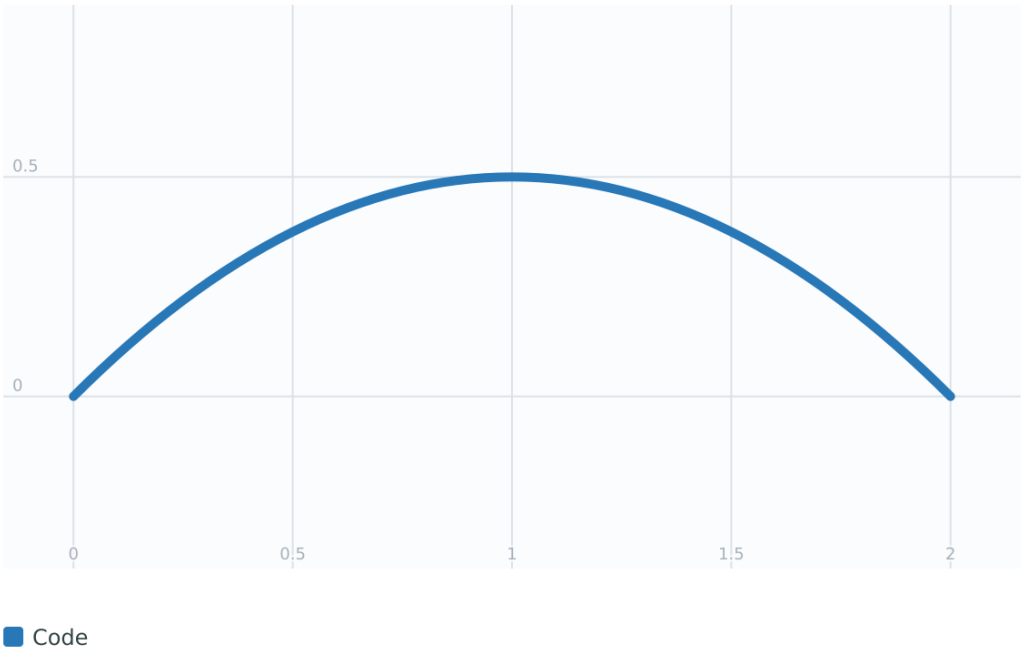
Code

```
SELECT ST_EndM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry);
```

Output

```
300
```

Figure



Geometry without M dimension returns NULL

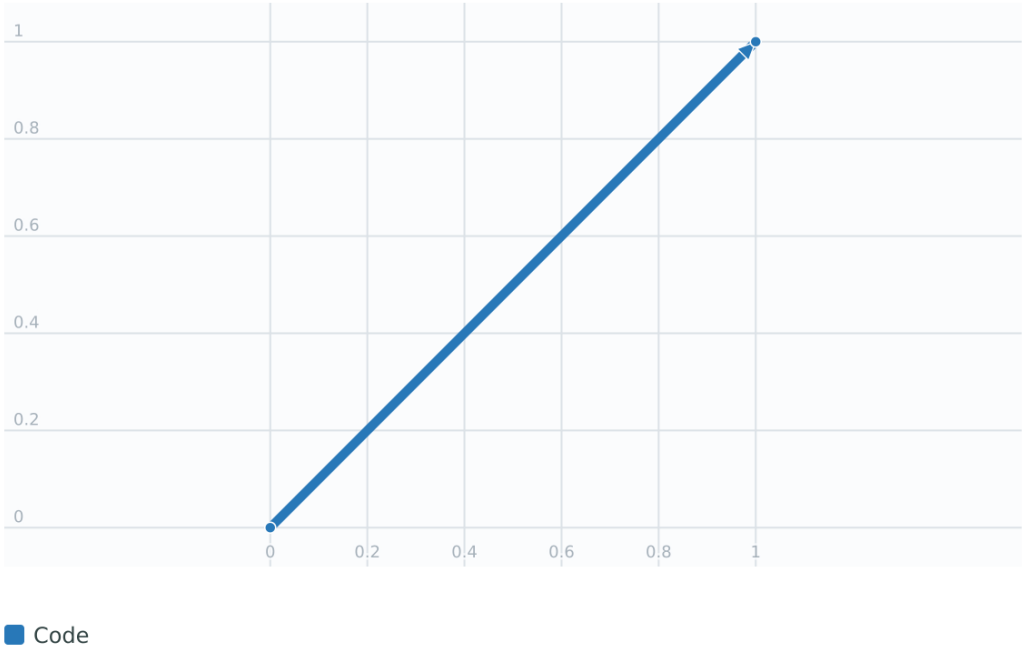
Code

```
SELECT ST_EndM('LINESTRING(0 0,1 1)'::geometry) IS NULL AS is_null;
```

Output

```
t
```

Figure



See Also

ST_StartM, ST_SetEndM, ST_EndPoint

7.4.42 ST_SetStartM

ST_SetStartM — Sets the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.

Synopsis

geometry **ST_SetStartM**(geometry geom, float8 m);

Description

Returns a copy of the input `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry with the M coordinate of the first point set to the specified value. If the input geometry does not have an M dimension, it will be added. Returns `NULL` if the input is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

 This method supports Circular Strings and Curves.

Examples

Set start M value of a LineString

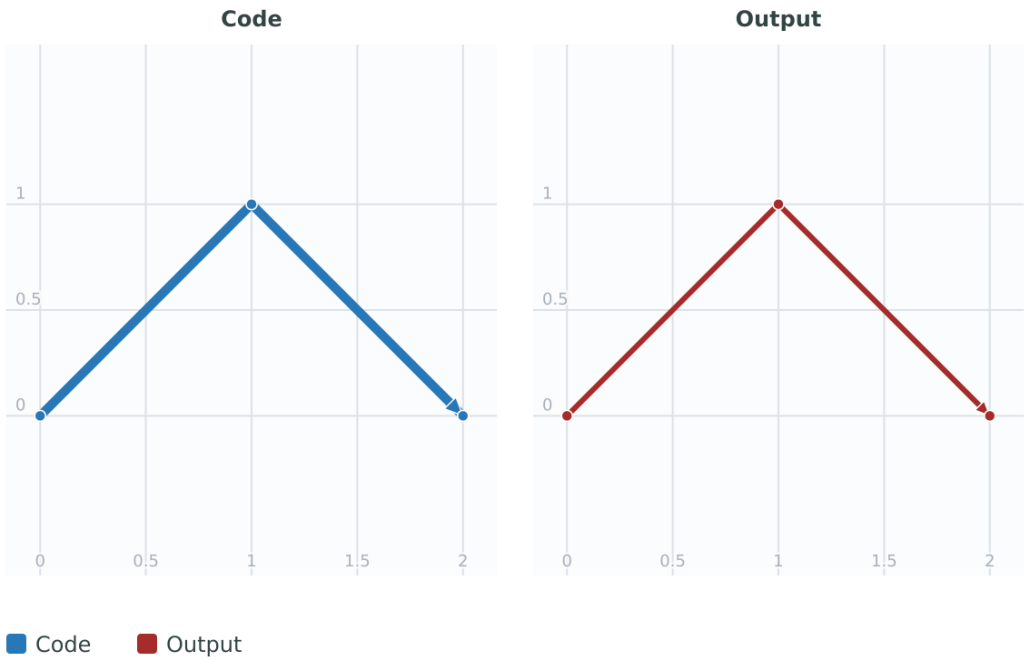
Code

```
SELECT ST_SetStartM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry, 100);
```

Output

```
LINESTRING M (0 0 100,1 1 20,2 0 30)
```

Figure



Add M dimension and set start M value

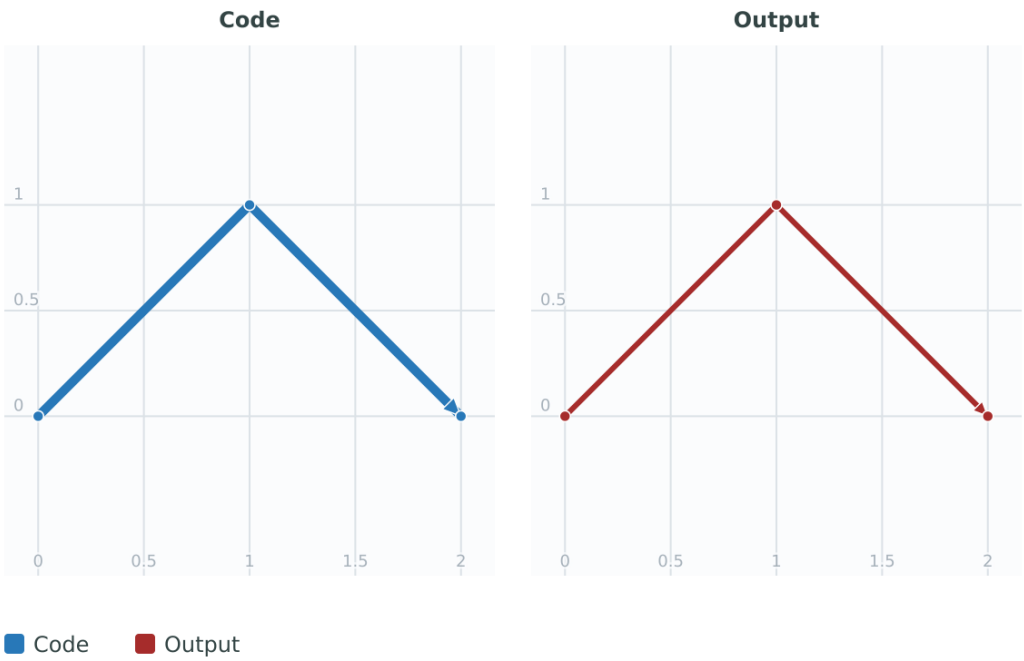
Code

```
SELECT ST_SetStartM('LINESTRING(0 0,1 1,2 0)'::geometry, 50);
```

Output

```
LINESTRING M (0 0 50,1 1 0,2 0 0)
```

Figure



Set start M value of a CircularString

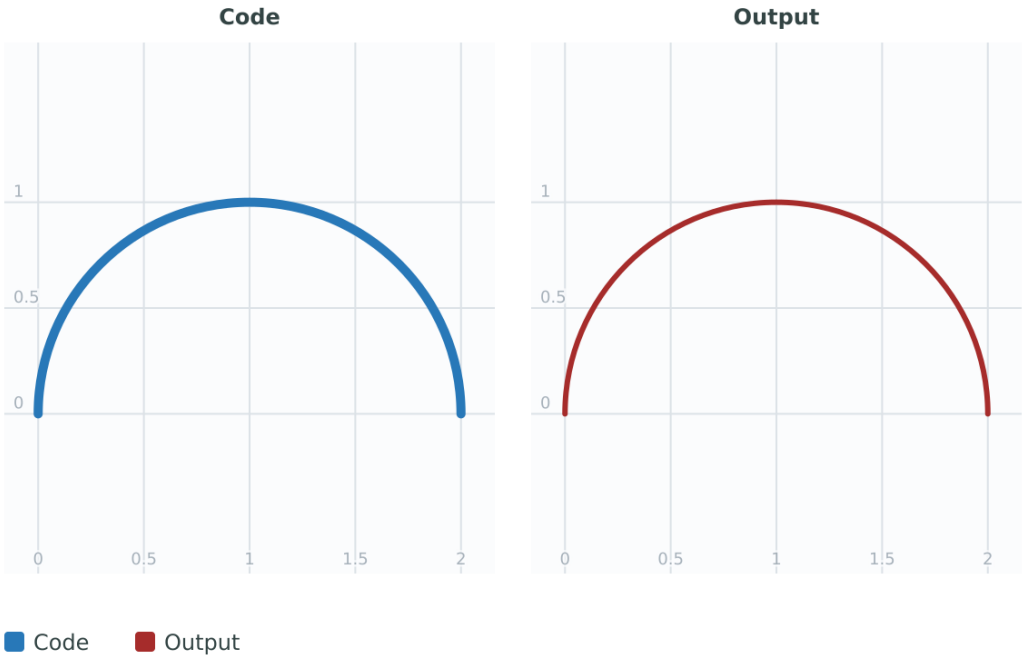
Code

```
SELECT ST_SetStartM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry, 99);
```

Output

```
CIRCULARSTRING M (0 0 99,1 1 10,2 0 15)
```

Figure



Set start M value of a NURBSCurve

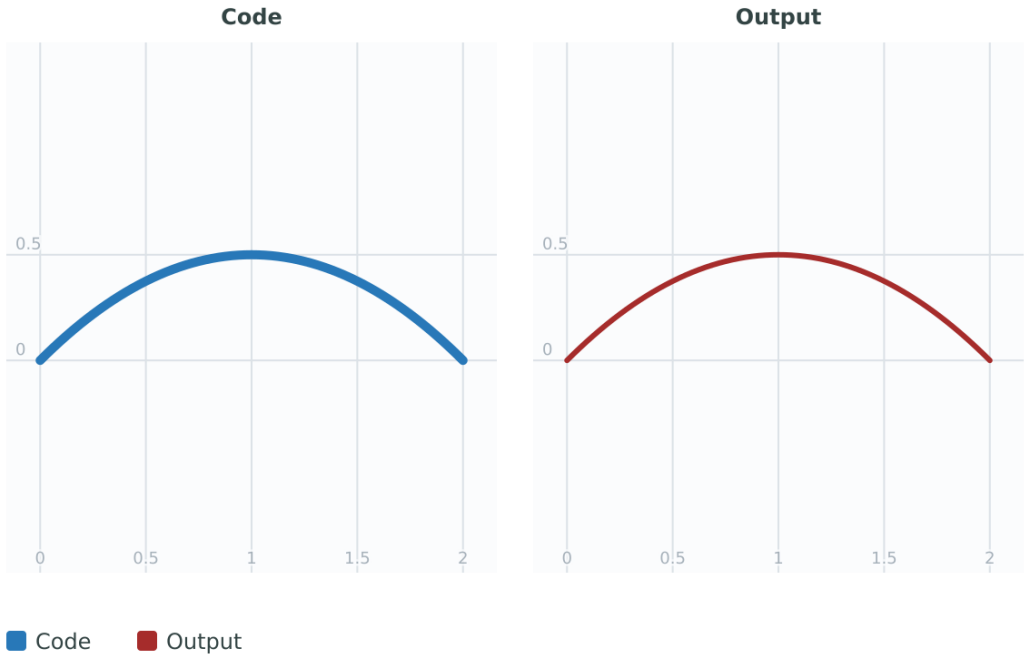
Code

```
SELECT ST_SetStartM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry, 500);
```

Output

```
NURBSCURVE M (DEGREE 2,CONTROLPOINTS M (NURBSPOINT(WEIGHTEDPOINT M (0 0 500),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (1 1 ↩
     200),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (2 0 300),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



See Also

ST_SetEndM, ST_StartM, ST_StartPoint

7.4.43 ST_SetEndM

ST_SetEndM — Sets the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.

Synopsis

geometry **ST_SetEndM**(geometry geom, float8 m);

Description

Returns a copy of the input `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry with the M coordinate of the last point set to the specified value. If the input geometry does not have an M dimension, it will be added. Returns `NULL` if the input is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

 This method supports Circular Strings and Curves.

Examples

Set end M value of a LineString

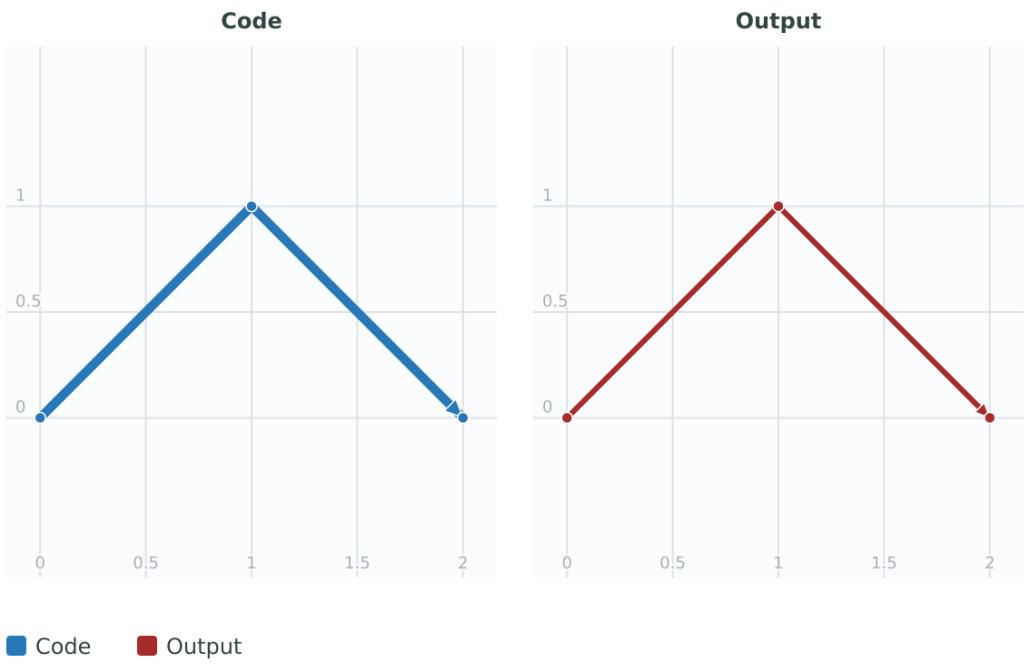
Code

```
SELECT ST_SetEndM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry, 999);
```

Output

```
LINESTRING M (0 0 10,1 1 20,2 0 999)
```

Figure



Add M dimension and set end M value

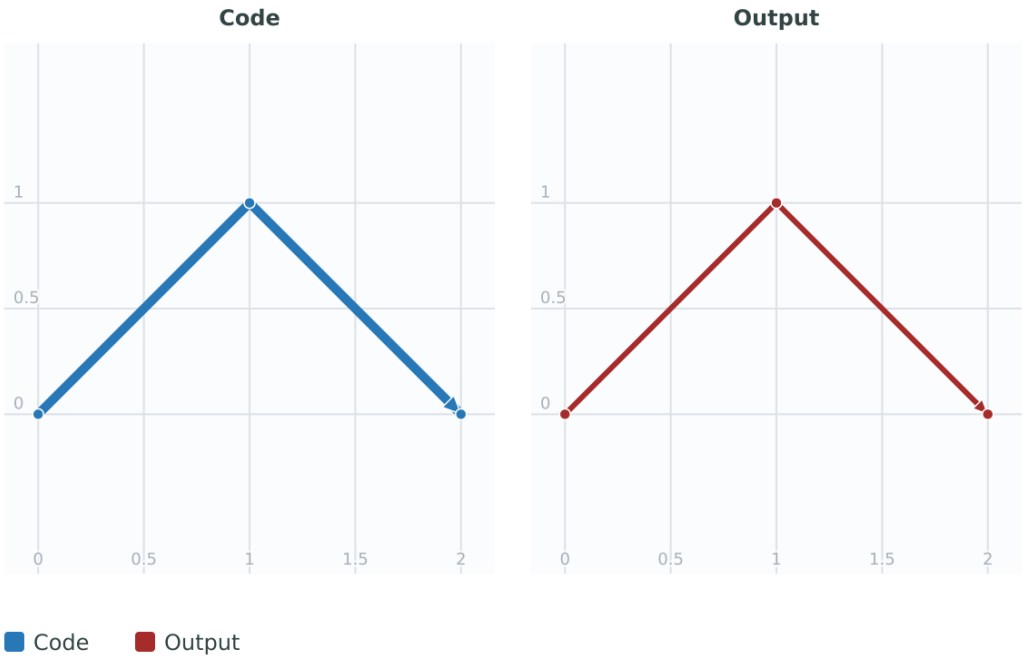
Code

```
SELECT ST_SetEndM('LINESTRING(0 0,1 1,2 0)'::geometry, 75);
```

Output

```
LINESTRING M (0 0 0,1 1 0,2 0 75)
```

Figure



Set end M value of a CircularString

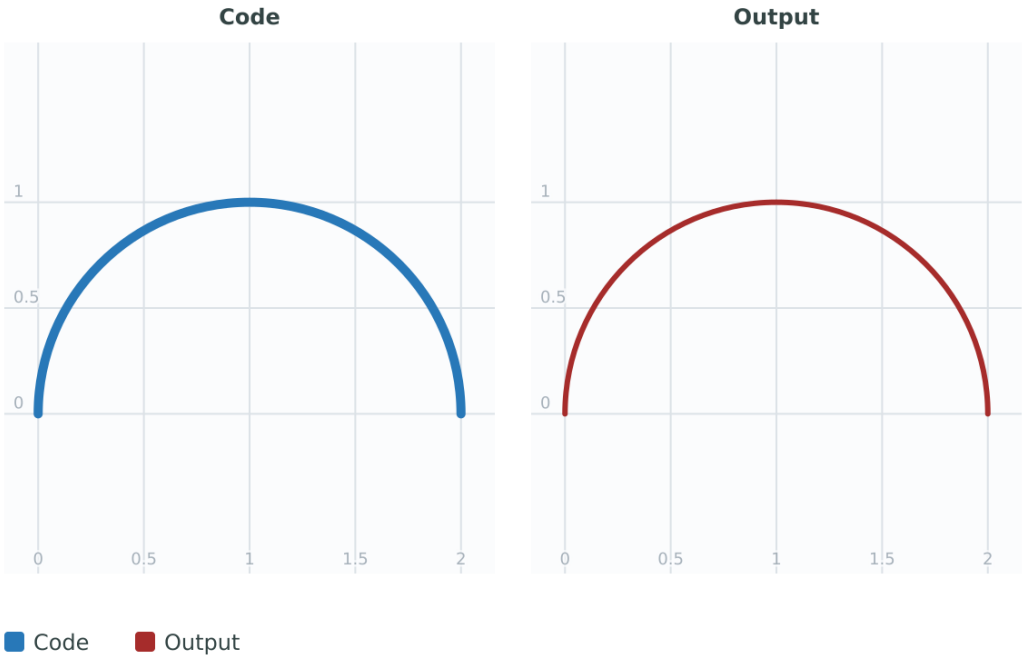
Code

```
SELECT ST_SetEndM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry, 88);
```

Output

```
CIRCULARSTRING M (0 0 5,1 1 10,2 0 88)
```

Figure



Set end M value of a NURBSCurve

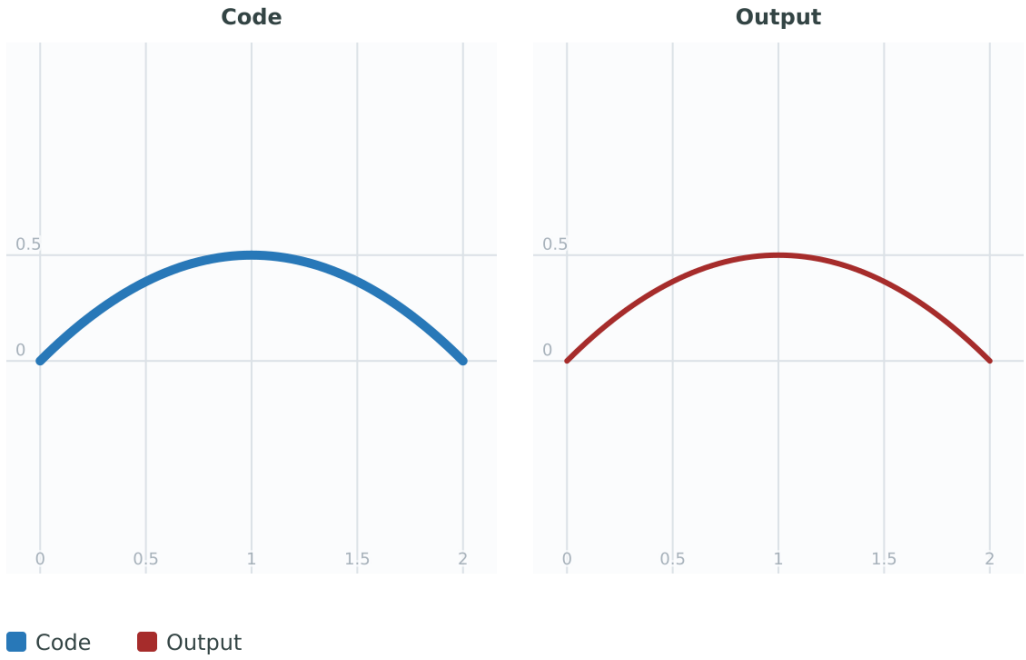
Code

```
SELECT ST_SetEndM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry, 999);
```

Output

```
NURBSCURVE M (DEGREE 2,CONTROLPOINTS M (NURBSPOINT(WEIGHTEDPOINT M (0 0 100),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (1 1 ↩
      200),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (2 0 999),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



See Also

ST_SetStartM, ST_EndM, ST_EndPoint

7.4.44 ST_Summary

ST_Summary — Returns a text summary of the contents of a geometry.

Synopsis

text **ST_Summary**(geometry g);
text **ST_Summary**(geography g);

Description

Returns a text summary of the contents of the geometry.

Flags shown square brackets after the geometry type have the following meaning:

- M: has M coordinate
- Z: has Z coordinate

- B: has a cached bounding box
- G: is geodetic (geography)
- S: has spatial reference system

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Availability: 1.2.2

Enhanced: 2.0.0 added support for geography

Enhanced: 2.1.0 S flag to denote if has a known spatial reference system

Enhanced: 2.2.0 Added support for TIN and Curves

Examples

Code

```
SELECT ST_Summary('LINESTRING(0 0,1 1)'::geometry) as geom,
ST_Summary('POLYGON((0 0,1 1,1 2,1 1,0 0))'::geography) geog;
```

Output

```
             geom              |           geog
-------------------------------+--------------------------
 LineString[B] with 2 points | Polygon[BGS] with 1 rings
                               | ring 0 has 5 points
                               :
(1 row)
```

Code

```
SELECT ST_Summary('LINESTRING(0 0 1,1 1 1)'::geography) As geog_line,
ST_Summary('SRID=4326;POLYGON((0 0 1,1 1 2,1 2 3,1 1 1,0 0 1))'::geometry) As geom_poly;
```

Output

```
          geog_line               |         geom_poly
--------------------------------- +--------------------------
 LineString[ZBGS] with 2 points | Polygon[ZBS] with 1 rings
                                 :    ring 0 has 5 points
                                 :
(1 row)
```

See Also

PostGIS_DropBBox, PostGIS_AddBBox, ST_Force3DM, ST_Force3DZ, ST_Force2D, geography

ST_IsValid, ST_IsValid, ST_IsValidReason, ST_IsValidDetail

7.4.45 ST_X

ST_X — Returns the X coordinate of a Point.

Synopsis

float **ST_X**(geometry a_point);

Description

Return the X coordinate of the point, or NULL if not available. Input must be a point.



Note
To get the minimum and maximum X value of geometry coordinates use the functions ST_XMin and ST_XMax.

This method implements the SQL/MM specification. SQL-MM 3: 6.1.3

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_X('POINT(1 2 3 4)');
```

Output

```
1
```

Code

```
SELECT ST_Y(ST_Centroid('LINESTRING(1 2 3 4,1 1 1 1)'::geometry));
```

Output

```
1.5
```

See Also

ST_Centroid, ST_GeomFromEWKT, ST_M, ST_XMax, ST_XMin, ST_Y, ST_Z

7.4.46 ST_Y

ST_Y — Returns the Y coordinate of a Point.

Synopsis

float **ST_Y**(geometry a_point);

Description

Return the Y coordinate of the point, or NULL if not available. Input must be a point.



Note
To get the minimum and maximum Y value of geometry coordinates use the functions ST_YMin and ST_YMax.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 6.1.4

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_Y('POINT(1 2 3 4)');
```

Output

```
2
```

Code

```
SELECT ST_Y(ST_Centroid('LINESTRING(1 2 3 4,1 1 1 1)'::geometry));
```

Output

```
1.5
```

See Also

ST_Centroid, ST_GeomFromEWKT, ST_M, ST_X, ST_YMax, ST_YMin, ST_Z

7.4.47 ST_Z

ST_Z — Returns the Z coordinate of a Point.

Synopsis

float **ST_Z**(geometry a_point);

Description

Return the Z coordinate of the point, or NULL if not available. Input must be a point.



Note
To get the minimum and maximum Z value of geometry coordinates use the functions ST_ZMin and ST_ZMax.

This method implements the SQL/MM specification.

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_Z('POINT(1 2 3 4)');
```

Output

```
3
```

See Also

ST_GeomFromEWKT, ST_M, ST_X, ST_Y, ST_ZMax, ST_ZMin

7.4.48 ST_ZMFlag

ST_ZMFlag — Returns a code indicating the ZM coordinate dimension of a geometry.

Synopsis

smallint **ST_ZMFlag**(geometry geomA);

Description

Returns a code indicating the ZM coordinate dimension of a geometry.

Values are: 0 = 2D, 1 = 3D-M, 2 = 3D-Z, 3 = 4D.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_ZMFlag('LINESTRING(1 2,3 4)');
```

Output

```
0
```

Code

```
SELECT ST_ZMFlag('LINESTRINGM(1 2 3, 3 4 3)');
```

Output

```
1
```

Code

```
SELECT ST_ZMFlag('CIRCULARSTRING(1 2 3,3 4 3,5 6 3)');
```

Output

```
2
```

Code

```
SELECT ST_ZMFlag('POINT(1 2 3 4)');
```

Output

```
3
```

See Also

ST_CoordDim, ST_NDims, ST_Dimension

7.4.49 ST_HasZ

ST_HasZ — Checks if a geometry has a Z dimension.

Synopsis

boolean **ST_HasZ**(geometry geom);

Description

Checks if the input geometry has a Z dimension and returns a boolean value. If the geometry has a Z dimension, it returns true; otherwise, it returns false.

Geometry objects with a Z dimension typically represent three-dimensional (3D) geometries, while those without it are two-dimensional (2D) geometries.

This function is useful for determining if a geometry has elevation or height information.

Availability: 3.5.0

 This function supports 3d and will not drop the z-index.

 This function supports M coordinates.

Examples

Code

```
SELECT ST_HasZ('POINT(1 2 3)');
```

Output

```
t
```

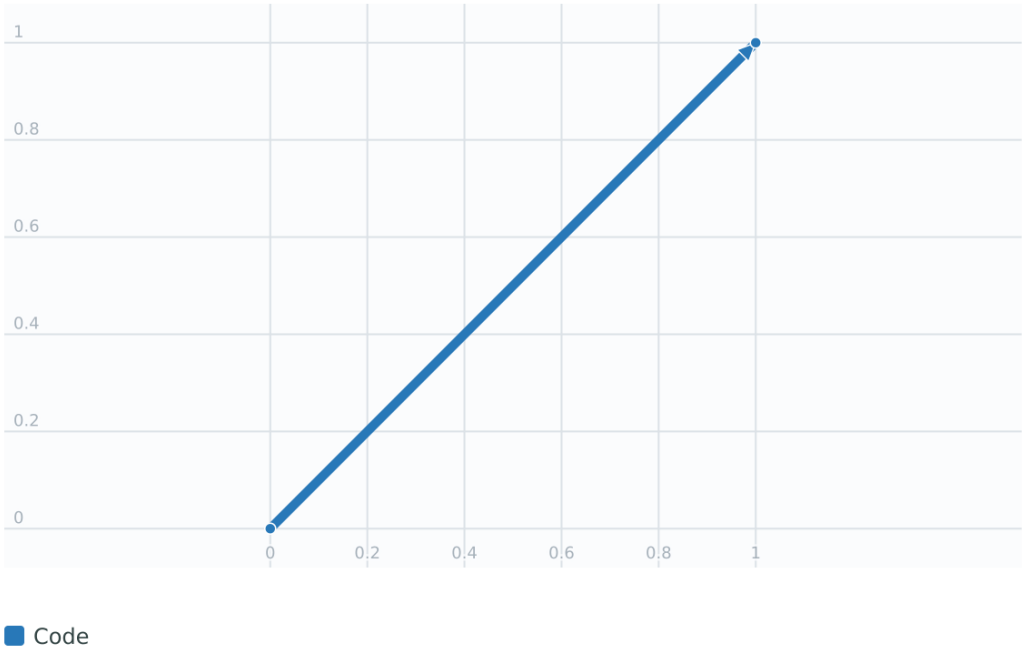
Code

```
SELECT ST_HasZ('LINESTRING(0 0,1 1)');
```

Output

```
f
```

Figure



See Also

ST_ZMFlag

ST_HasM

7.4.50 ST_HasM

ST_HasM — Checks if a geometry has an M (measure) dimension.

Synopsis

boolean **ST_HasM**(geometry geom);

Description

Checks if the input geometry has an M (measure) dimension and returns a boolean value. If the geometry has an M dimension, it returns true; otherwise, it returns false.

Geometry objects with an M dimension typically represent measurements or additional data associated with spatial features.

This function is useful for determining if a geometry includes measure information.

Availability: 3.5.0

 This function supports 3d and will not drop the z-index.

 This function supports M coordinates.

Examples

Code

```
SELECT ST_HasM('POINTM(1 2 3)');
```

Output

```
t
```

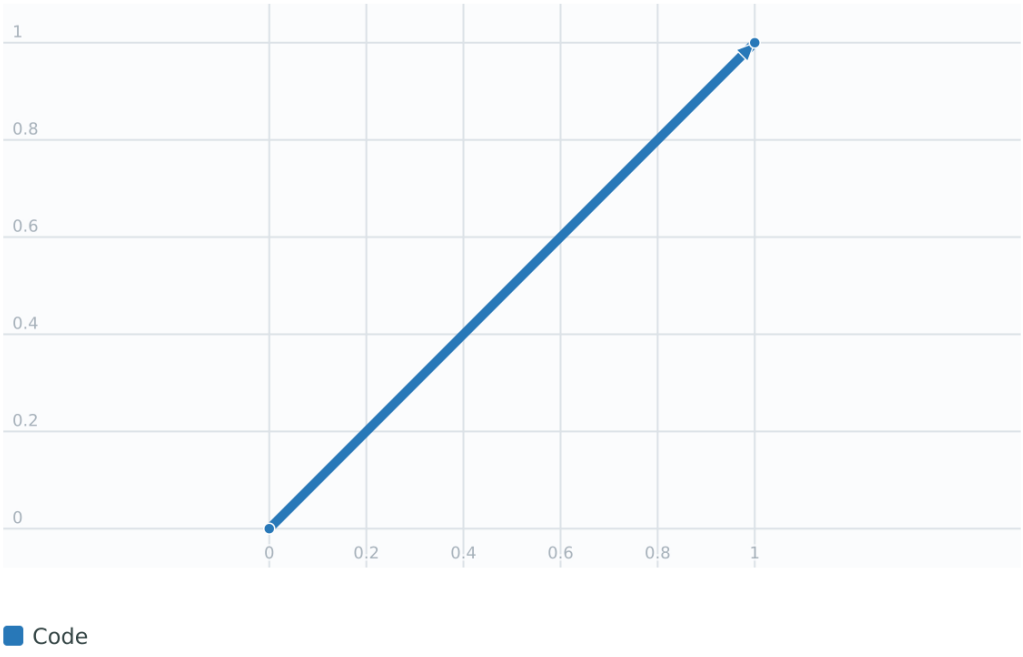
Code

```
SELECT ST_HasM('LINESTRING(0 0,1 1)');
```

Output

```
f
```

Figure



See Also

ST_ZMFlag
ST_HasZ

7.4.51 ST_ControlPoints

ST_ControlPoints — Returns the control points of a NURBS curve as a MULTIPOINT geometry.

Synopsis

geometry **ST_ControlPoints**(geometry nurbscurve);

Description

Extracts the control points from a NURBS curve and returns them as a MULTIPOINT geometry. Control points define the control polygon that influences the shape of the NURBS curve.

The function raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

Examples

This example extracts control points from a NURBS curve.

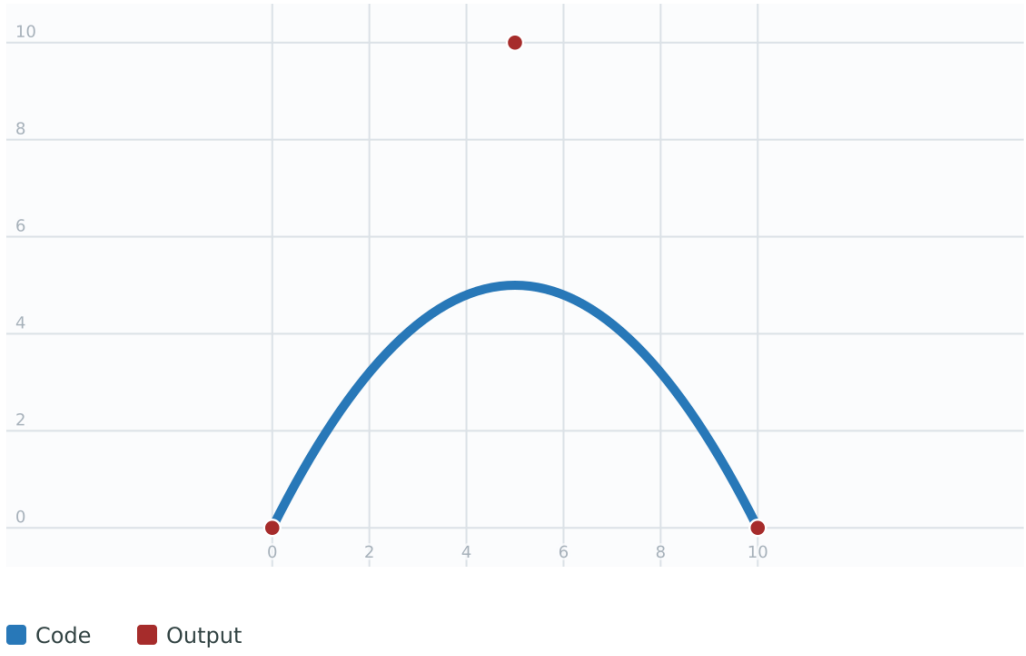
Code

```
SELECT ST_ControlPoints('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
MULTIPOINT((0 0),(5 10),(10 0))
```

Figure



This example counts the control points.

Code

```
SELECT ST_NPoints(ST_ControlPoints('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry));
```

Output

```
3
```

See Also

ST_MakeNurbsCurve, ST_NumControlPoints, ST_Weights

7.4.52 ST_Degree

ST_Degree — Returns the polynomial degree of a NURBS curve.

Synopsis

integer **ST_Degree**(geometry nurbscurve);

Description

Returns the polynomial degree of a NURBS curve as an integer. The degree determines the curve's smoothness and flexibility. Higher degrees produce smoother curves but require more control points.

The function raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

Examples

This example returns the degree of a NURBS curve.

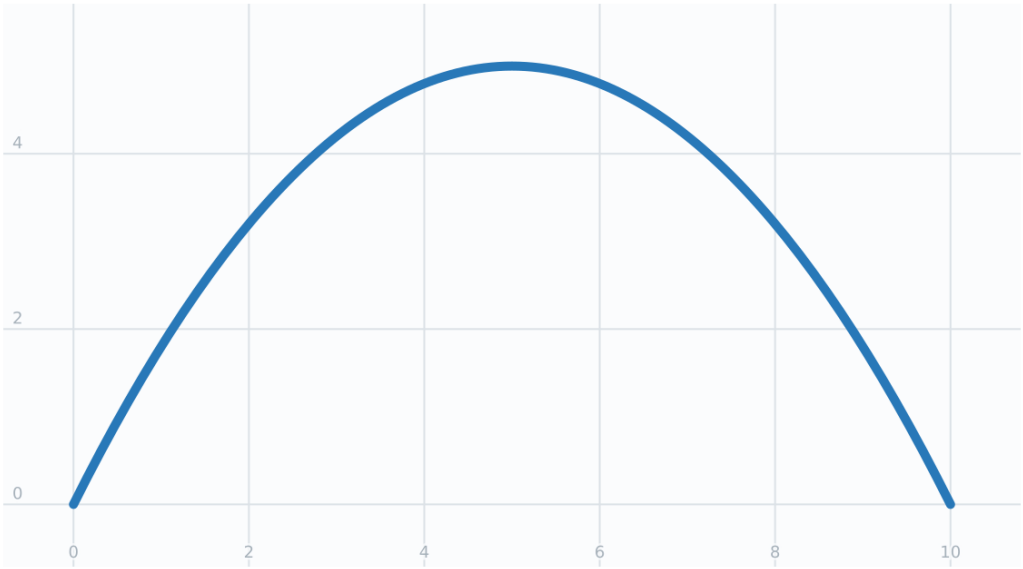
Code

```
SELECT ST_Degree('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
2
```

Figure



This example checks whether a NURBS curve is cubic.

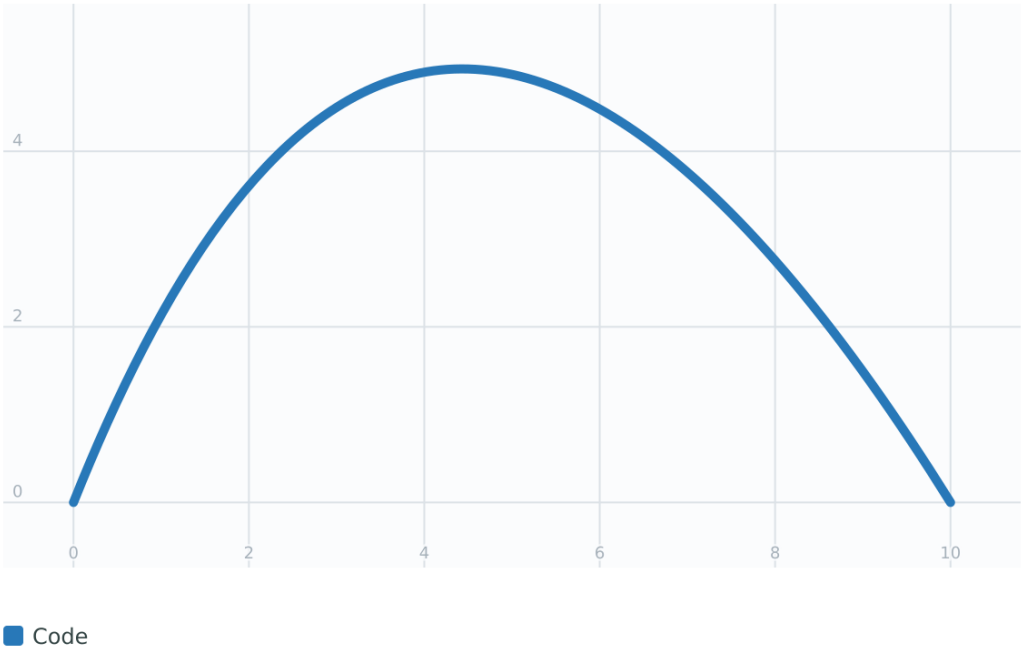
Code

```
SELECT ST_Degree('NURBSCURVE(3, (0 0, 2 5, 5 8, 10 0))'::geometry) = 3;
```

Output

```
t
```

Figure



See Also

ST_MakeNurbsCurve, ST_NumControlPoints

7.4.53 ST_Weights

ST_Weights — Returns the weight array of a rational NURBS curve.

Synopsis

float8[] **ST_Weights**(geometry nurbscurve);

Description

Returns the weight values of a rational NURBS curve as an array of double precision values. Weights control the influence of each control point on the curve shape. Higher weights pull the curve closer to the corresponding control point.

Returns NULL if the curve has no weights (non-rational B-spline) or if the input is NULL. Raises an error if the input is not a NURBS curve.

Availability: 3.7.0

Examples

This example returns weights from a rational NURBS curve.

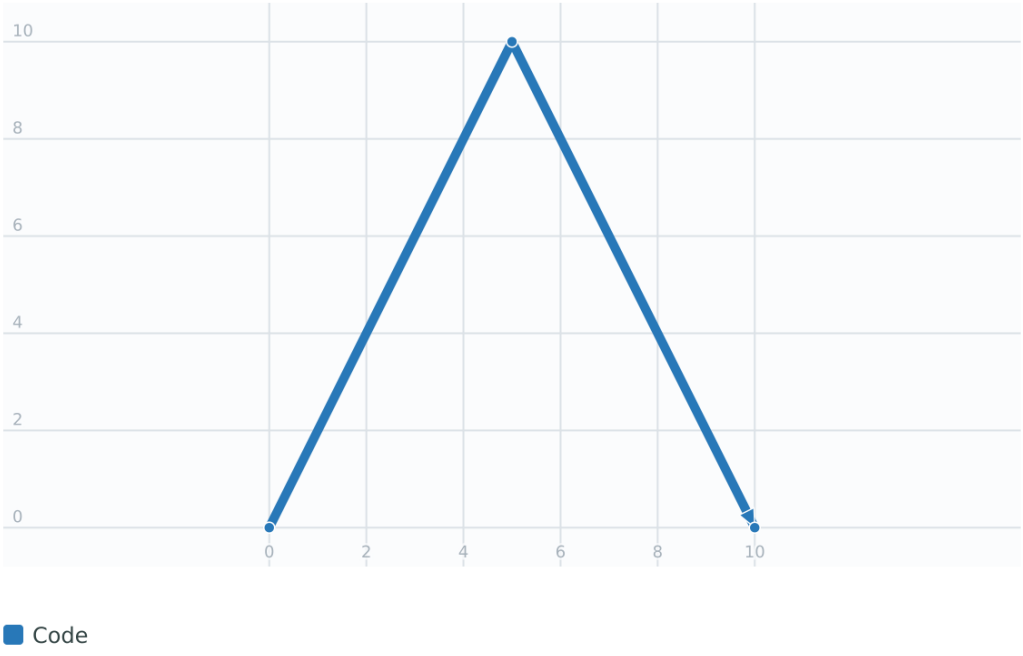
Code

```
SELECT ST_Weights(ST_MakeNurbsCurve(2, 'LINESTRING(0 0,5 10,10 0)'::geometry, ARRAY[1.0,  ↩
    2.0, 1.0]));
```

Output

```
{1,2,1}
```

Figure



This example checks the weights of a non-rational curve.

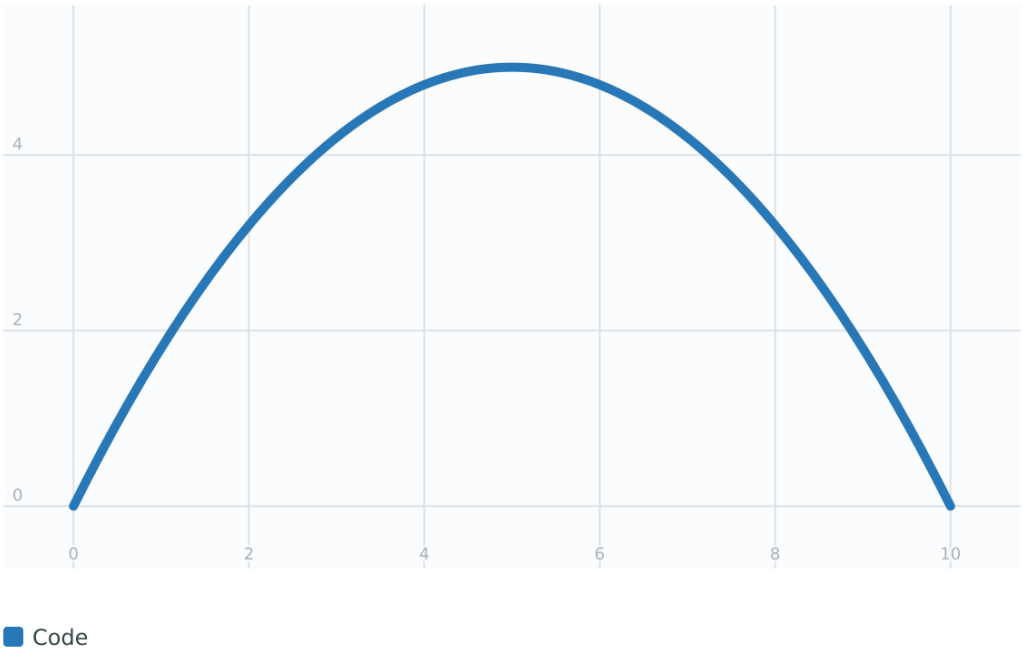
Code

```
SELECT ST_Weights('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
null
```

Figure



See Also

ST_MakeNurbsCurve, ST_NurbsCurveIsRational, ST_ControlPoints

7.4.54 ST_Knots

ST_Knots — Returns the knot vector of a NURBS curve.

Synopsis

float8[] **ST_Knots**(geometry nurbscurve);

Description

Returns the knot vector of a NURBS curve as an array of double precision values. The knot vector defines the parameter space and controls where and how the control points affect the curve. It must be a non-decreasing sequence with length equal to (number of control points + degree + 1).

Returns NULL if the curve uses the default uniform clamped knot vector or if the input is NULL. Raises an error if the input is not a NURBS curve.

Availability: 3.7.0

Examples

This example returns knots from a NURBS curve with a custom knot vector.

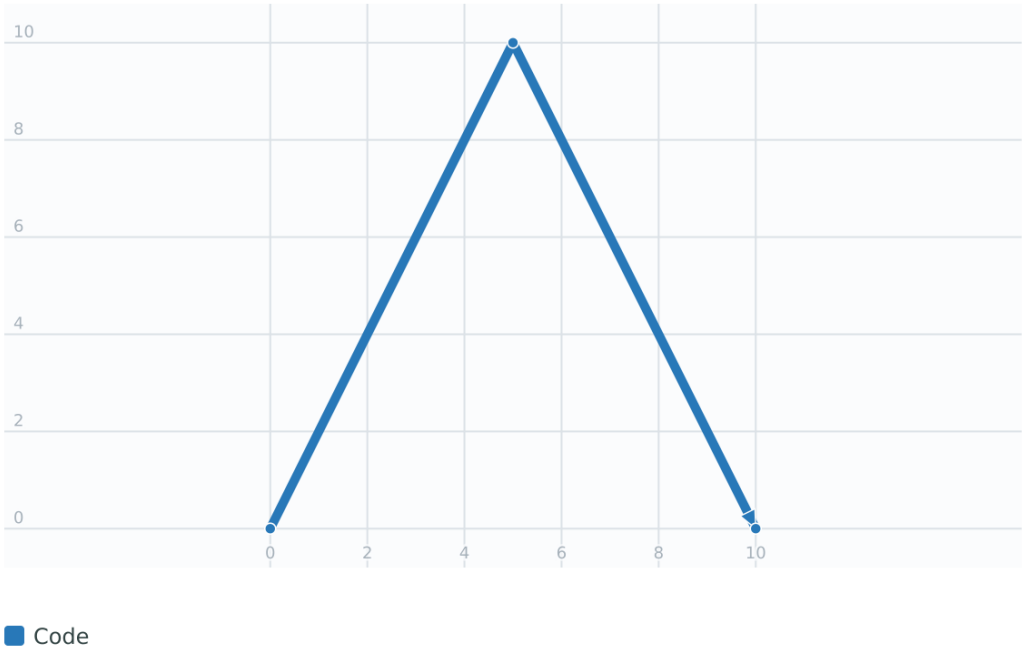
Code

```
SELECT ST_Knots(ST_MakeNurbsCurve(
        2,
        'LINESTRING(0 0,5 10,10 0)'::geometry,
        ARRAY[1.0, 2.0, 1.0],
        ARRAY[0, 0, 0, 1, 1, 1]));
```

Output

```
{0,0,0,1,1,1}
```

Figure



This example checks the knots of a curve using the default knot vector.

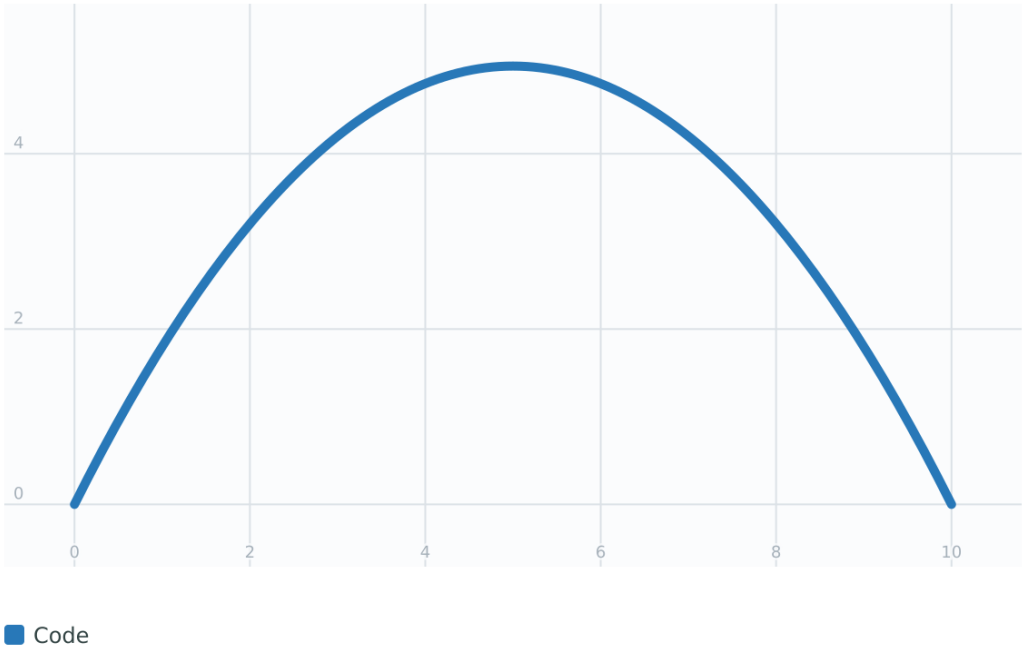
Code

```
SELECT ST_Knots('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
null
```

Figure



See Also

ST_MakeNurbsCurve, ST_Degree, ST_NumControlPoints

7.4.55 ST_NumControlPoints

ST_NumControlPoints — Returns the number of control points in a NURBS curve.

Synopsis

integer **ST_NumControlPoints**(geometry nurbscurve);

Description

Returns the number of control points in a NURBS curve. The number of control points must be at least (degree + 1) for a valid NURBS curve.

Raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

Examples

This example returns the number of control points.

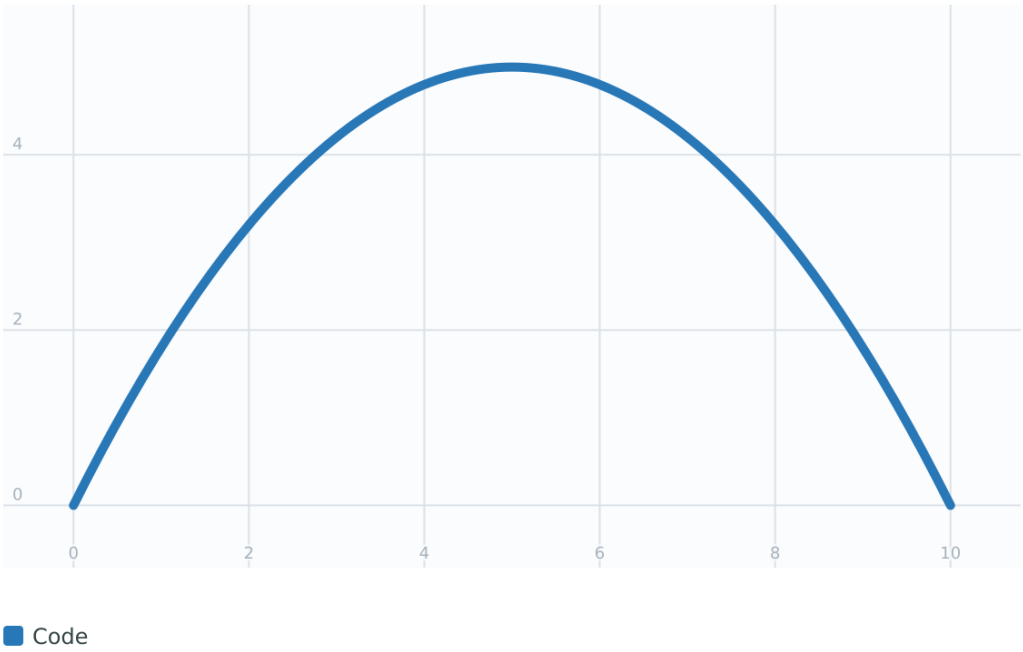
Code

```
SELECT ST_NumControlPoints('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
3
```

Figure



This example verifies that a curve has enough control points for its degree.

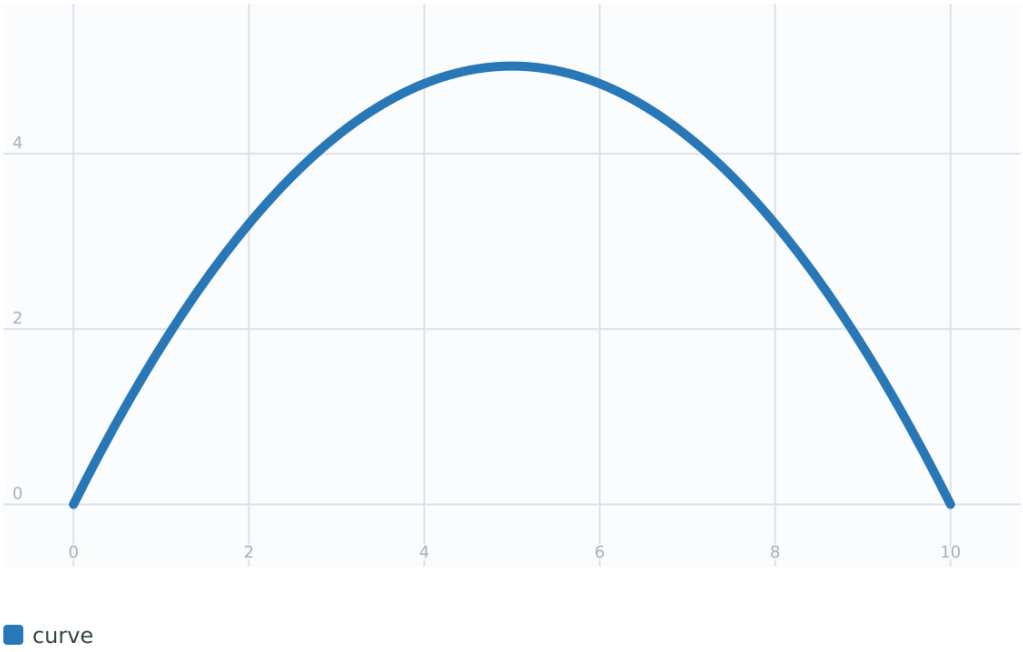
Code

```
SELECT ST_NumControlPoints(curve) >= ST_Degree(curve) + 1 AS valid
FROM (SELECT 'NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry AS curve) AS t;
```

Output

```
t
```

Figure



See Also

ST_MakeNurbsCurve, ST_ControlPoints, ST_Degree

7.4.56 ST_NurbsCurveIsRational

ST_NurbsCurveIsRational — Checks if a NURBS curve is rational (has weights).

Synopsis

boolean **ST_NurbsCurveIsRational**(geometry nurbscurve);

Description

Returns TRUE if the NURBS curve has weights assigned to its control points (rational NURBS), FALSE if it uses uniform weights (non-rational B-spline).

Rational NURBS curves provide more flexibility in curve shape control through variable weights. Non-rational B-splines use uniform weights and are a special case of NURBS curves.

Raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

Examples

This example checks whether a curve is rational.

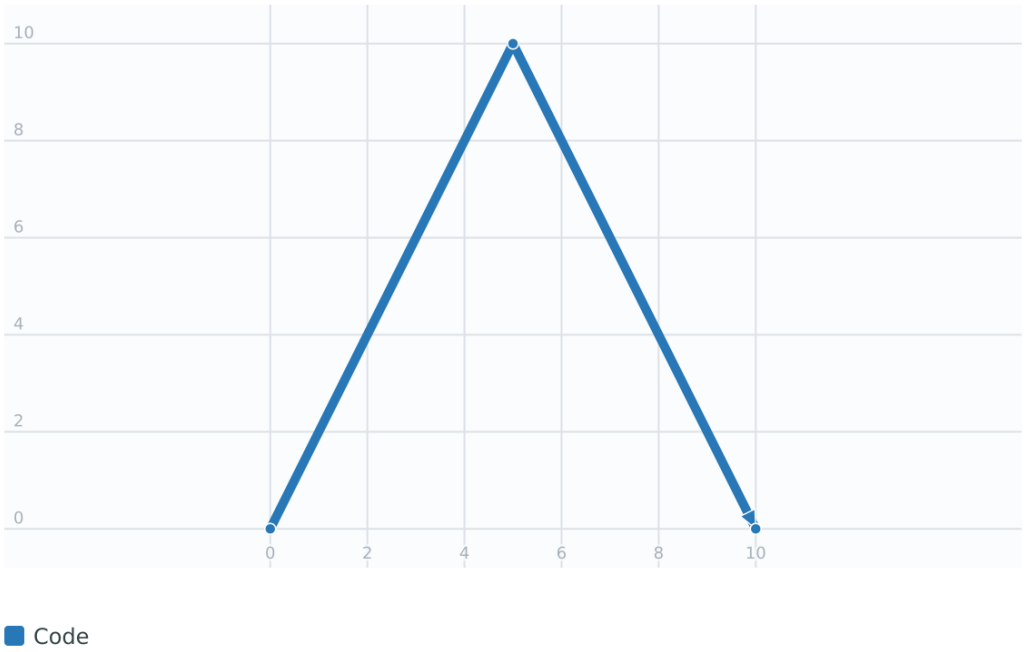
Code

```
SELECT ST_NurbsCurveIsRational(ST_MakeNurbsCurve(
        2,
        'LINESTRING(0 0,5 10,10 0)'::geometry,
        ARRAY[1.0, 2.0, 1.0]));
```

Output

```
t
```

Figure



This example checks a non-rational curve.

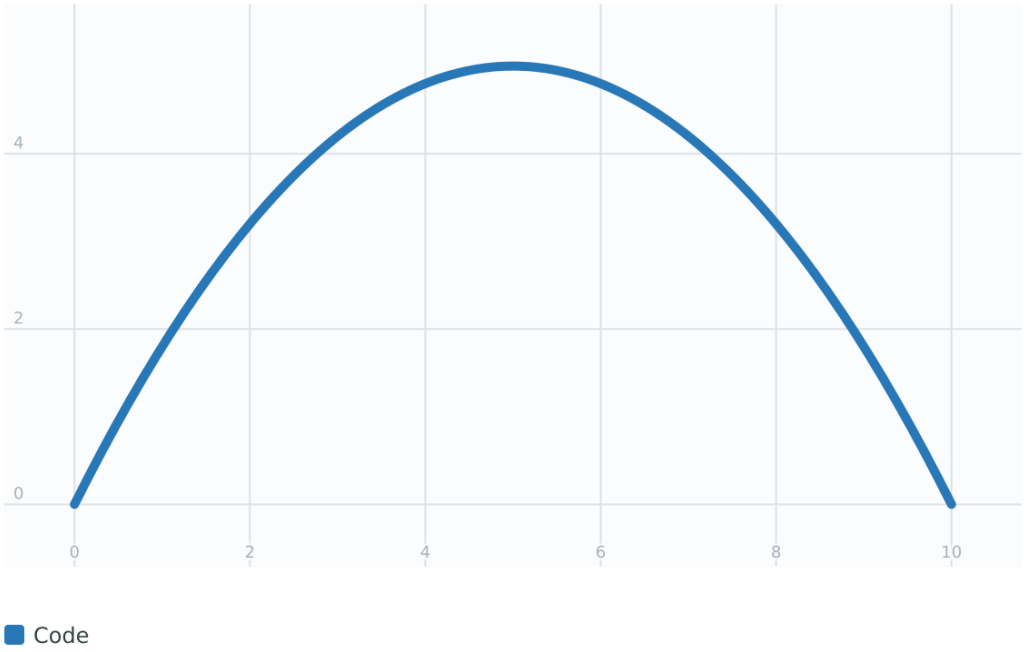
Code

```
SELECT ST_NurbsCurveIsRational('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
f
```

Figure



See Also

ST_MakeNurbsCurve, ST_Weights, ST_IsNurbsCurve

7.4.57 ST_IsNurbsCurve

ST_IsNurbsCurve — Checks if a geometry is a NURBS curve.

Synopsis

boolean **ST_IsNurbsCurve**(geometry geom);

Description

Returns TRUE if the geometry is a NURBS curve, FALSE otherwise. This function can be used to test geometry types before applying NURBS-specific functions.

Returns NULL if the input is NULL.

Availability: 3.7.0

Examples

This example checks whether a geometry is a NURBS curve.

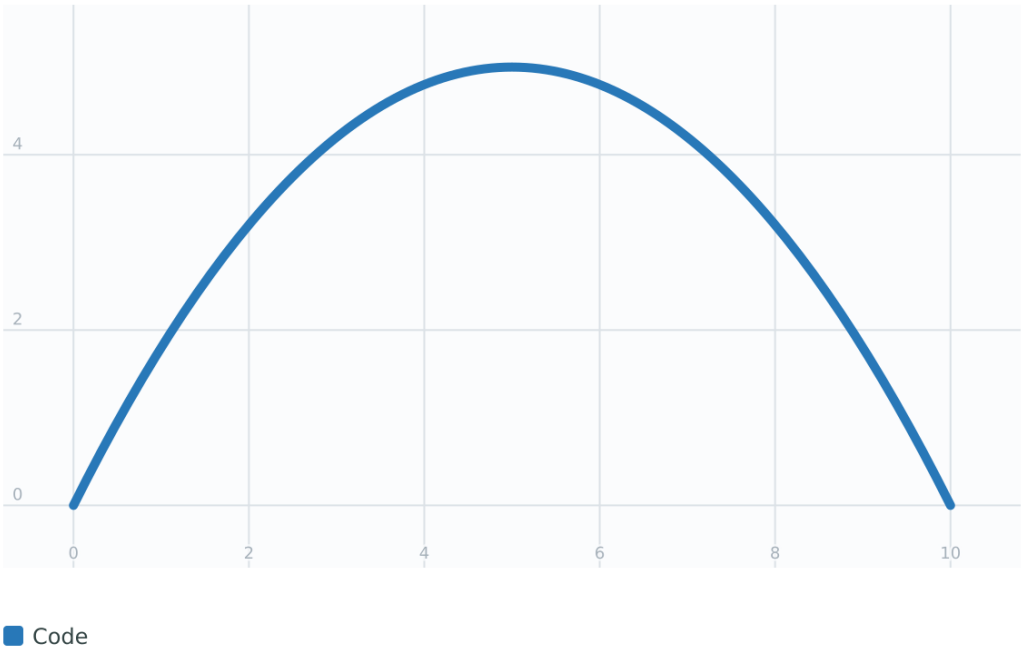
Code

```
SELECT ST_IsNurbsCurve('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
t
```

Figure



This example tests a regular LineString.

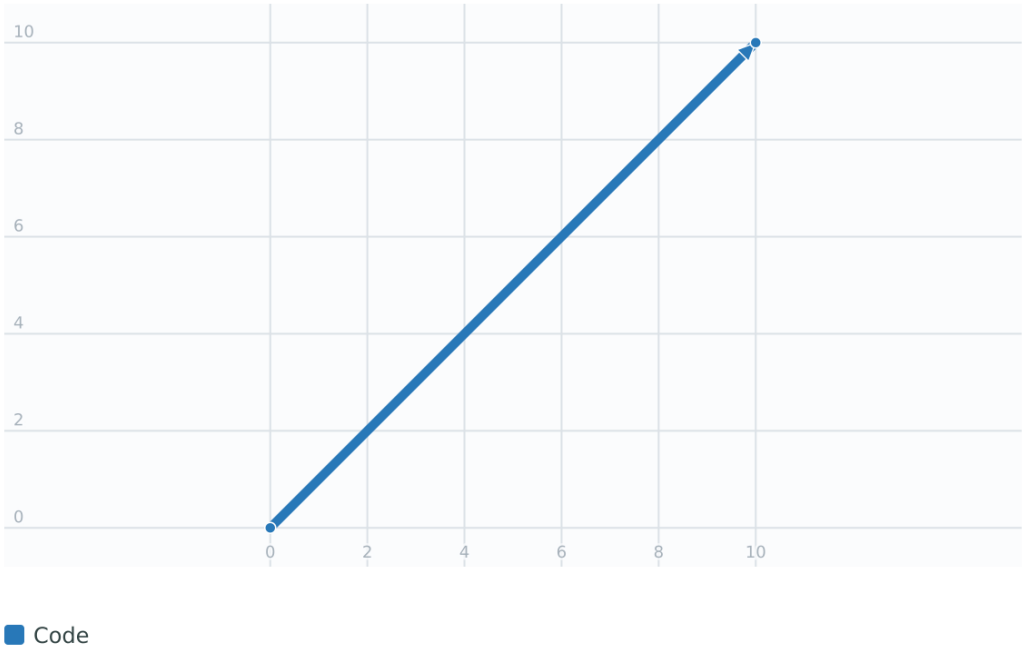
Code

```
SELECT ST_IsNurbsCurve('LINESTRING(0 0,10 10)'::geometry);
```

Output

```
f
```

Figure



See Also

ST_MakeNurbsCurve, ST_GeometryType, ST_NurbsCurveIsRational

7.4.58 ST_Evaluate

ST_Evaluate — Evaluates a NURBS curve at a specific parameter value and returns the resulting point.

Synopsis

geometry **ST_Evaluate**(geometry nurbscurve, float8 parameter);

Description

Evaluates a NURBS curve at the specified parameter value and returns a POINT geometry representing the position on the curve. The parameter typically ranges from 0.0 (start of curve) to 1.0 (end of curve), though the actual range depends on the knot vector.

Raises an error if the input is not a NURBS curve or if evaluation fails. Returns NULL if either input is NULL.

Availability: 3.7.0

Examples

This example returns the midpoint of a NURBS curve.

Code

```
SELECT ST_Evaluate('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry, 0.5);
```

This example samples multiple points along a curve.

Code

```
SELECT t, ST_Evaluate(curve, t) AS point
FROM (SELECT 'NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry AS curve) AS c,
         generate_series(0, 1, 0.25) AS t;
```

See Also

ST_MakeNurbsCurve, ST_NurbsToLineString

7.4.59 ST_NurbsToLineString

ST_NurbsToLineString — Converts a NURBS curve to a LINESTRING by uniform sampling.

Synopsis

geometry **ST_NurbsToLineString**(geometry nurbscurve, integer num_segments=32);

Description

Converts a NURBS curve to a LINESTRING geometry by creating a piecewise linear approximation. The curve is sampled at uniformly distributed parameter values and the resulting points are connected with straight line segments.

The optional `num_segments` parameter specifies the number of line segments in the output (default: 32, range: 2-10000). Higher values produce smoother approximations but increase geometry complexity.

Raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

Examples

This example converts a NURBS curve to a LineString using the default quality.

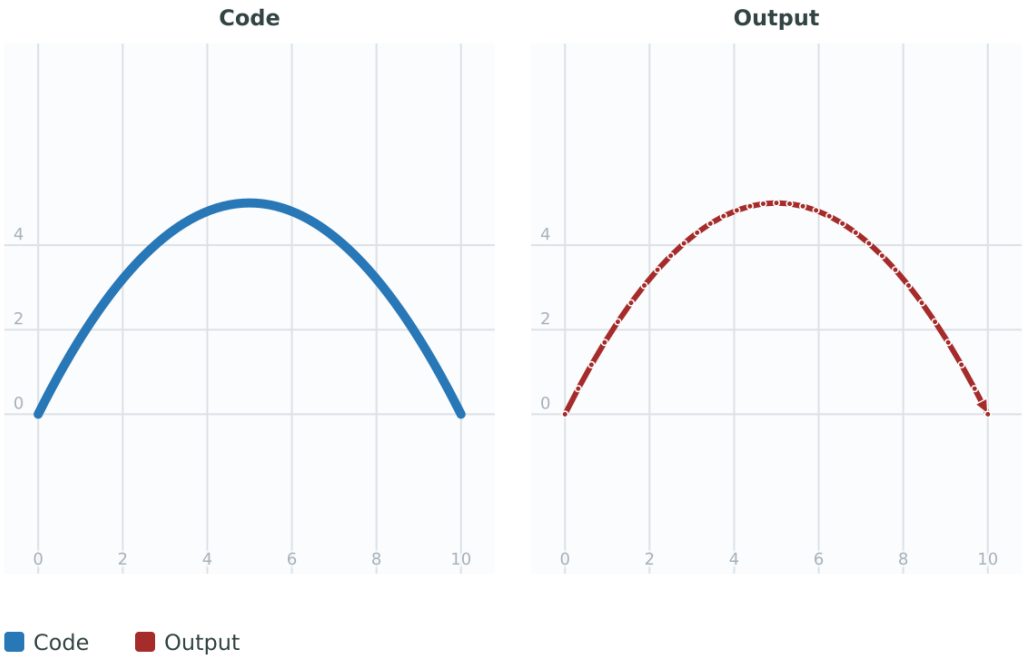
Code

```
SELECT ST_NurbsToLineString('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
LINESTRING(0 0,0.3125 0.60546875,0.625 1.171875,0.9375 1.69921875,1.25 2.1875,1.5625 2.63671875,1.875 3.046875,2.1875 ↩
    3.41796875,2.5 3.75,2.8125 4.04296875,3.125 4.296875,3.4375 4.51171875,3.75 4.6875,4.0625 4.82421875,4.375 ↩
    4.921875,4.6875 4.98046875,5 5,5.3125 4.98046875,5.625 4.921875,5.9375 4.82421875,6.25 4.6875,6.5625 ↩
    4.51171875,6.875 4.296875,7.1875 4.04296875,7.5 3.75,7.8125 3.41796875,8.125 3.046875,8.4375 2.63671875,8.75 ↩
    2.1875,9.0625 1.69921875,9.375 1.171875,9.6875 0.60546875,10 0)
```

Figure

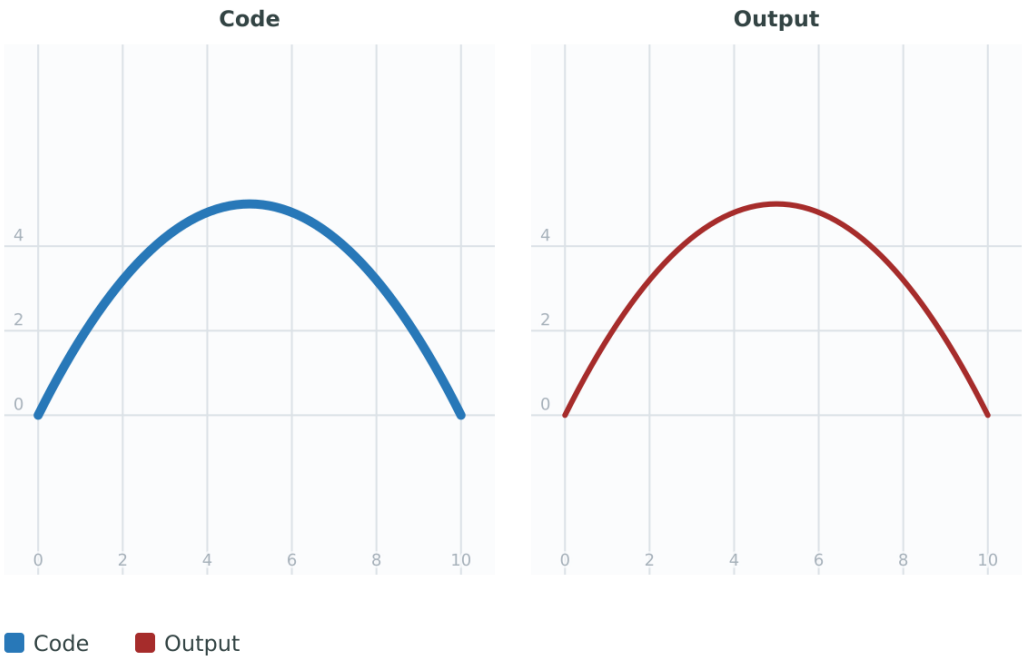


This example creates a high-quality approximation with 64 segments.

Code

```
SELECT ST_NurbsToLineString('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry, 64);
```

Figure



This example creates a lower-quality, faster approximation with 16 segments.

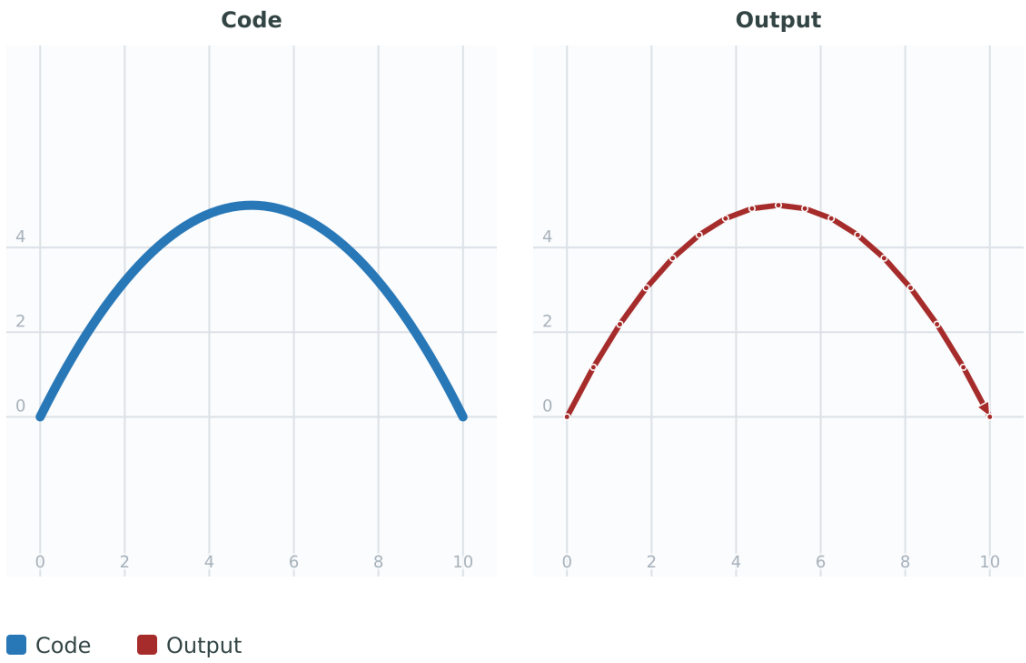
Code

```
SELECT ST_NurbsToLineString('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry, 16);
```

Output

```
LINESTRING(0 0,0.625 1.171875,1.25 2.1875,1.875 3.046875,2.5 3.75,3.125 4.296875,3.75 4.6875,4.375 4.921875,5 5,5.625 ↩
    4.921875,6.25 4.6875,6.875 4.296875,7.5 3.75,8.125 3.046875,8.75 2.1875,9.375 1.171875,10 0)
```

Figure



See Also

ST_MakeNurbsCurve, ST_Evaluate, ST_Segmentize

7.5 Geometry Editors

7.5.1 ST_AddPoint

ST_AddPoint — Add a point to a LineString.

Synopsis

geometry **ST_AddPoint**(geometry linestring, geometry point);

geometry **ST_AddPoint**(geometry linestring, geometry point, integer position = -1);

Description

Adds a point to a LineString before the index *position* (using a 0-based index). If the *position* parameter is omitted or is -1 the point is appended to the end of the LineString.

Availability: 1.1.0

This function supports 3d and will not drop the z-index.

Examples

Add a point to the end of a 3D line

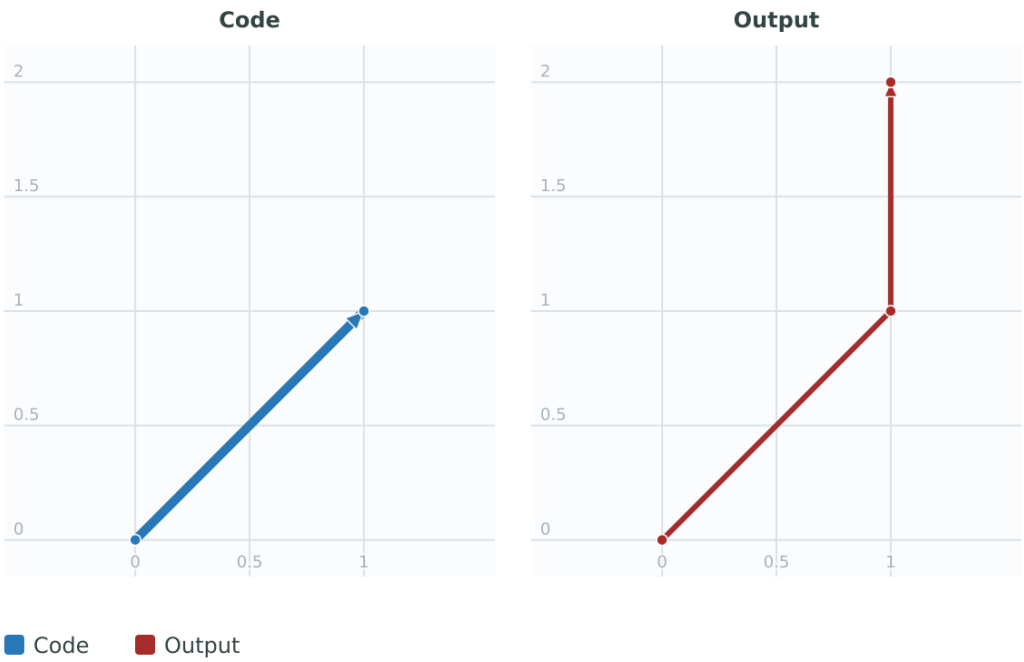
Code

```
SELECT ST_AddPoint('LINESTRING(0 0 1,1 1 1)', ST_MakePoint(1, 2, 3));
```

Output

```
LINESTRING(0 0 1,1 1 1,1 2 3)
```

Figure



Guarantee all lines in a table are closed by adding the start point of each line to the end of the line only for those that are not closed.

Code

```
UPDATE sometable
SET geom = ST_AddPoint(geom, ST_StartPoint(geom))
FROM sometable
WHERE ST_IsClosed(geom) = false;
```

See Also

ST_RemovePoint, ST_SetPoint

7.5.2 ST_CollectionExtract

ST_CollectionExtract — Given a geometry collection, returns a multi-geometry containing only elements of a specified type.

Synopsis

geometry **ST_CollectionExtract**(geometry collection);

geometry **ST_CollectionExtract**(geometry collection, integer type);

Description

Given a geometry collection, returns a homogeneous multi-geometry.

If the *type* is not specified, returns a multi-geometry containing only geometries of the highest dimension. So polygons are preferred over lines, which are preferred over points.

If the *type* is specified, returns a multi-geometry containing only that type. If there are no sub-geometries of the right type, an EMPTY geometry is returned. Only points, lines and polygons are supported. The type numbers are:

- 1 == POINT
- 2 == LINESTRING
- 3 == POLYGON

For atomic geometry inputs, the geometry is returned unchanged if the input type matches the requested type. Otherwise, the result is an EMPTY geometry of the specified type. If required, these can be converted to multi-geometries using ST_Multi.



Warning
MultiPolygon results are not checked for validity. If the polygon components are adjacent or overlapping the result will be invalid. (For example, this can occur when applying this function to an ST_Split result.) This situation can be checked with ST_IsValid and repaired with ST_MakeValid.

Availability: 1.5.0



Note
Prior to 1.5.3 this function returned atomic inputs unchanged, no matter type. In 1.5.3 non-matching single geometries returned a NULL result. In 2.0.0 non-matching single geometries return an EMPTY result of the requested type.

Examples

Extract highest-dimension type:

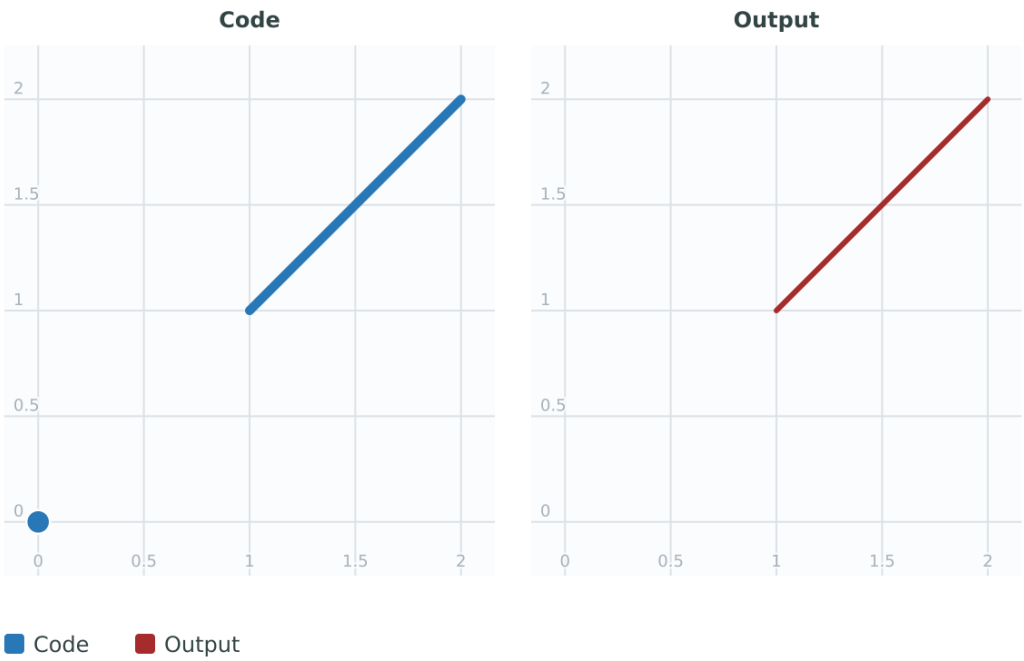
Code

```
SELECT ST_CollectionExtract('GEOMETRYCOLLECTION( POINT(0 0), LINESTRING(1 1,2 2) )');
```

Output

```
MULTILINESTRING((1 1, 2 2))
```

Figure



Extract points (type 1 == POINT):

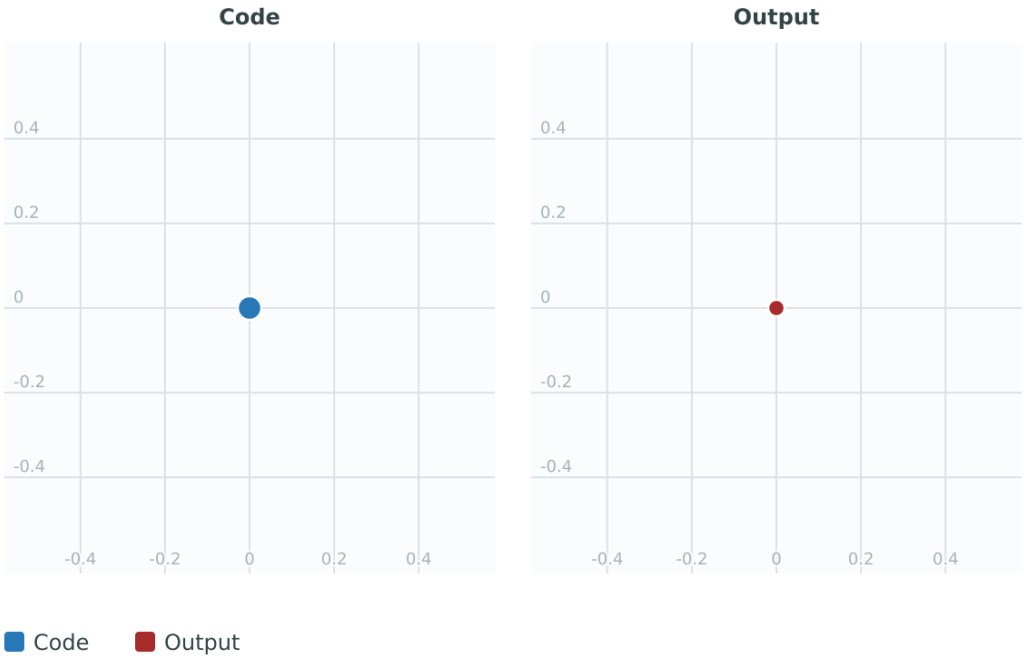
Code

```
SELECT ST_CollectionExtract('GEOMETRYCOLLECTION(GEOMETRYCOLLECTION(POINT(0 0)))',
        1);
```

Output

```
MULTIPOINT((0 0))
```

Figure



Extract lines (type 2 == LINESTRING):

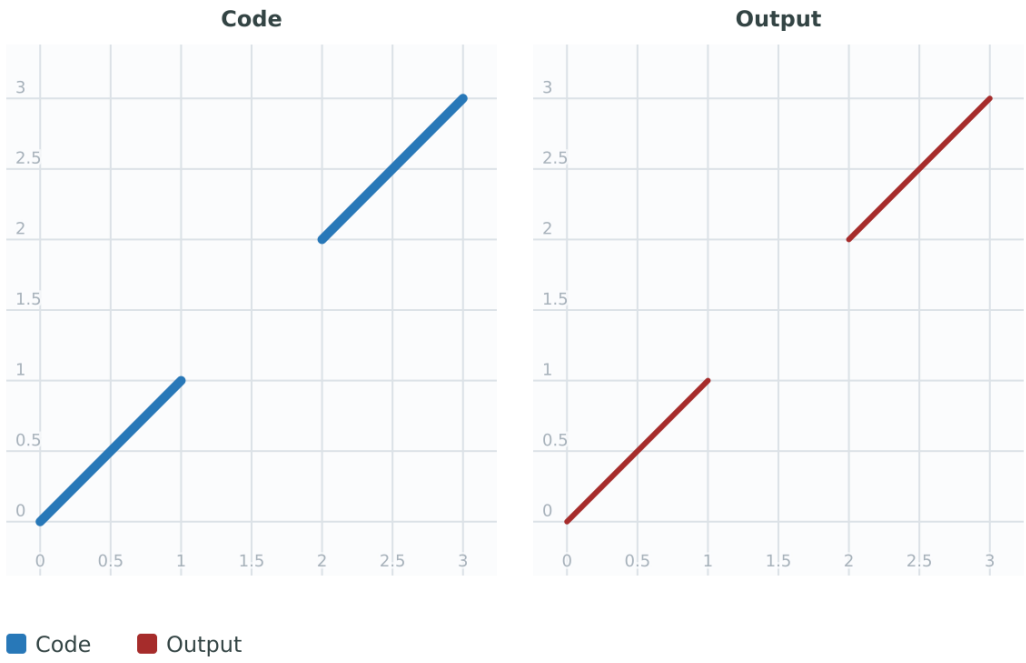
Code

```
SELECT ST_CollectionExtract('GEOMETRYCOLLECTION(GEOMETRYCOLLECTION(LINESTRING(0 0,1 1)),  ↩
    LINESTRING(2 2,3 3))',
        2);
```

Output

```
MULTILINESTRING((0 0, 1 1), (2 2, 3 3))
```

Figure



See Also

ST_CollectionHomogenize, ST_Multi, ST_IsValid, ST_MakeValid

7.5.3 ST_CollectionHomogenize

ST_CollectionHomogenize — Returns the simplest representation of a geometry collection.

Synopsis

geometry **ST_CollectionHomogenize**(geometry collection);

Description

Given a geometry collection, returns the "simplest" representation of the contents.

- Homogeneous (uniform) collections are returned as the appropriate multi-geometry.
- Heterogeneous (mixed) collections are flattened into a single GeometryCollection.

- Collections containing a single atomic element are returned as that element.
- Atomic geometries are returned unchanged. If required, these can be converted to a multi-geometry using ST_Multi.



Warning
This function does not ensure that the result is valid. In particular, a collection containing adjacent or overlapping Polygons will create an invalid MultiPolygon. This situation can be checked with ST_IsValid and repaired with ST_MakeValid.

Availability: 2.0.0

Examples

Single-element collection converted to an atomic geometry

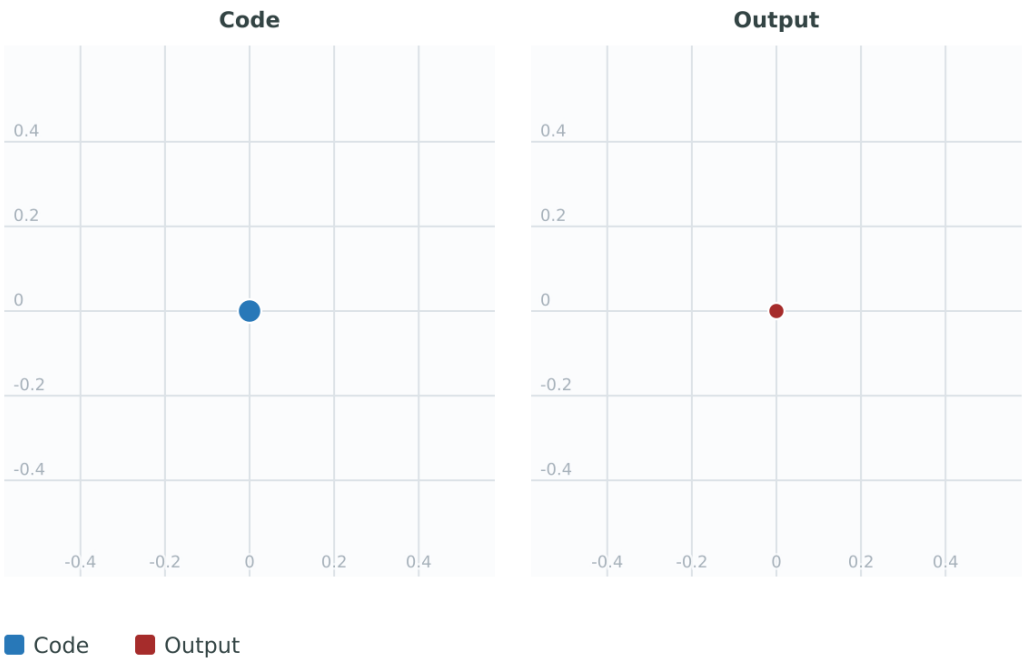
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(POINT(0 0))');
```

Output

```
POINT(0 0)
```

Figure



Nested single-element collection converted to an atomic geometry:

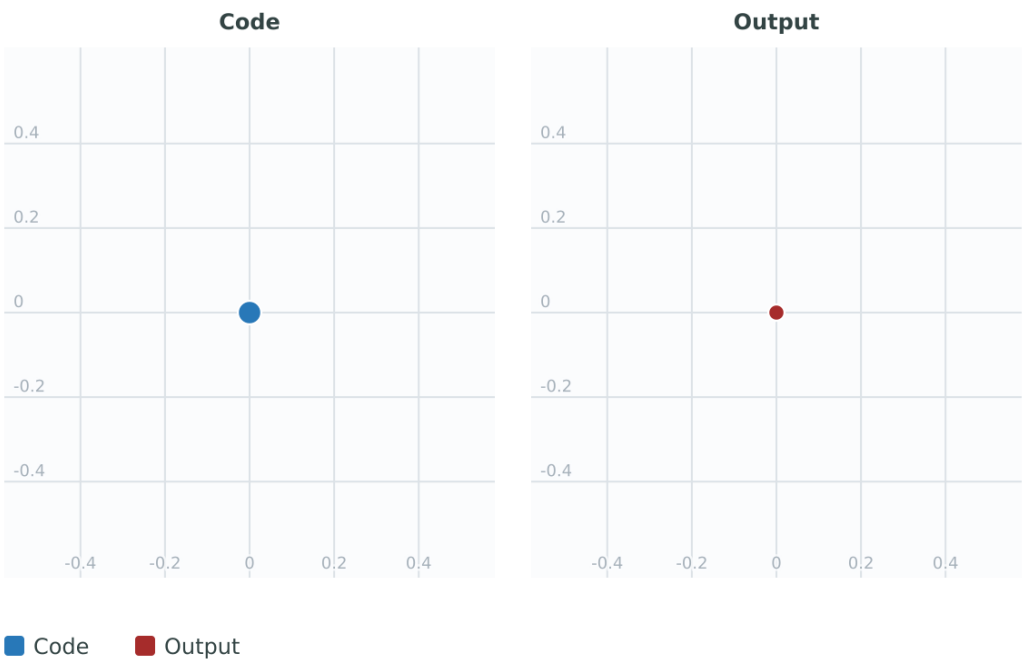
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(MULTIPOINT((0 0)))');
```

Output

```
POINT(0 0)
```

Figure



Collection converted to a multi-geometry:

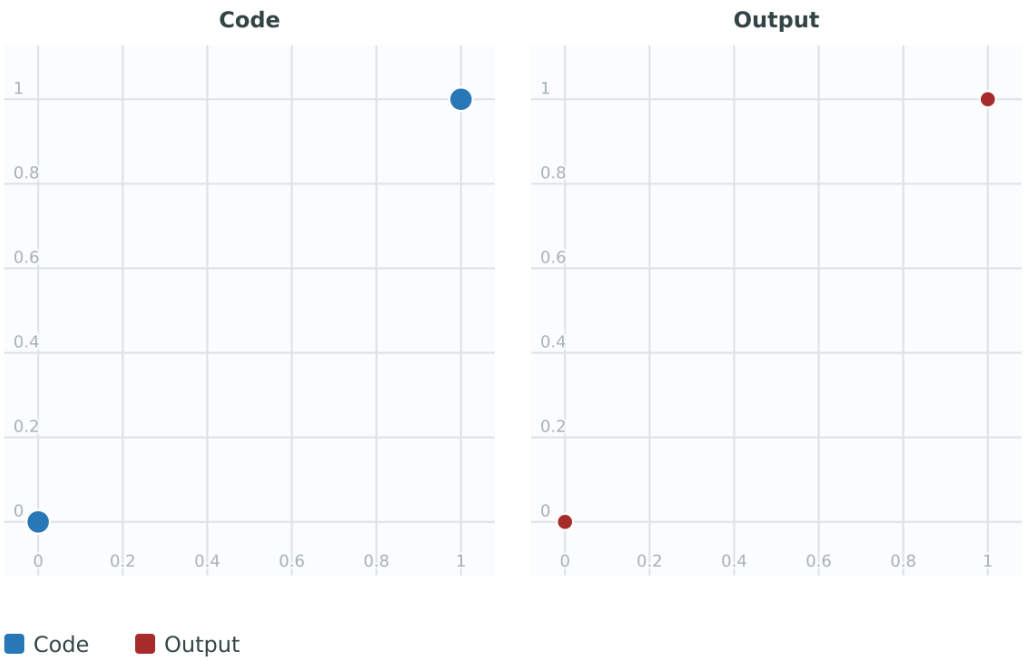
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(POINT(0 0), POINT(1 1))');
```

Output

```
MULTIPOINT((0 0),(1 1))
```

Figure



Nested heterogeneous collection flattened to a GeometryCollection:

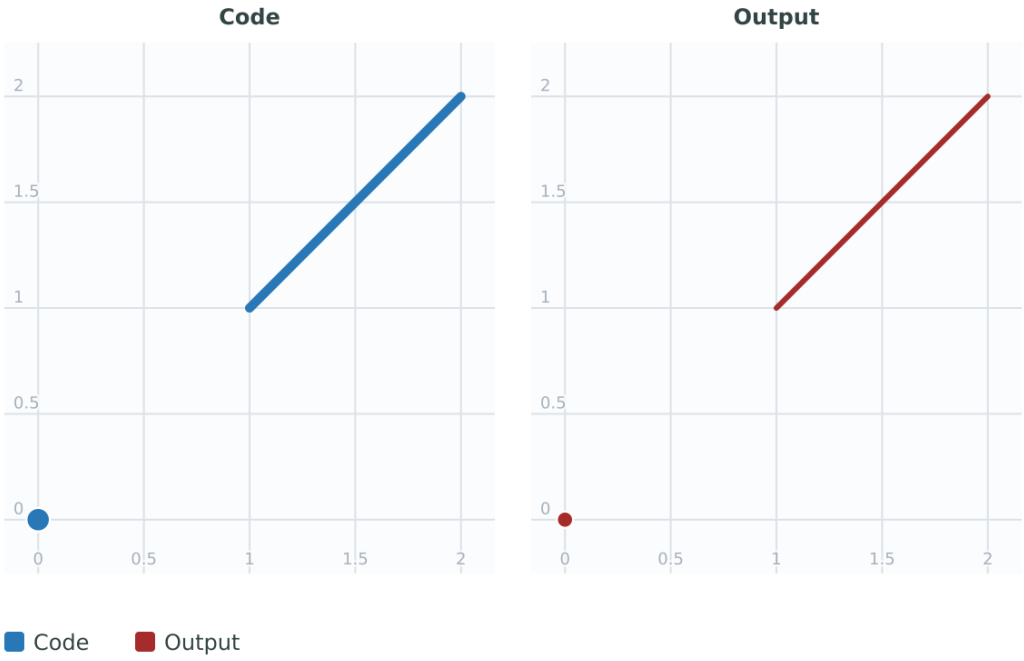
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(POINT(0 0), GEOMETRYCOLLECTION(
    LINESTRING(1 1,2 2)))');
```

Output

```
GEOMETRYCOLLECTION(POINT(0 0), LINESTRING(1 1,2 2))
```

Figure



Collection of Polygons converted to an (invalid) MultiPolygon:

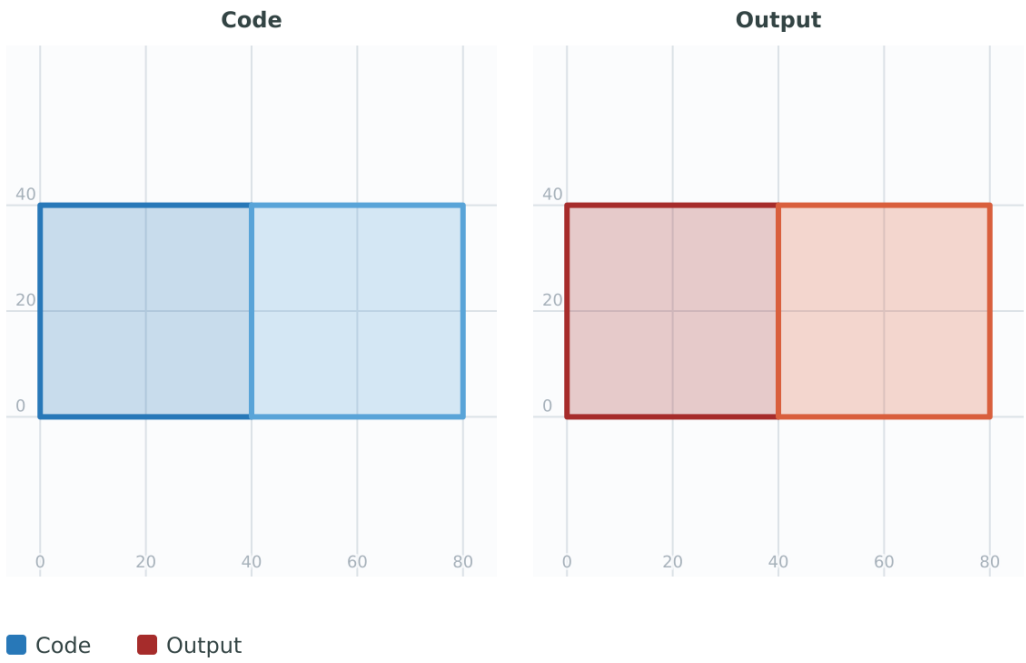
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION (POLYGON ((10 50,50 50,50 10,10 10,10 50) ↩
    ),POLYGON ((90 50,90 10,50 10,50 50,90 50)))');
```

Output

```
MULTIPOLYGON(((10 50,50 50,50 10,10 10,10 50)),((90 50,90 10,50 10,50 50,90 50)))
```

Figure



See Also

ST_CollectionExtract, ST_Multi, ST_IsValid, ST_MakeValid

7.5.4 ST_CurveToLine

ST_CurveToLine — Converts a geometry containing curves to a linear geometry.

Synopsis

geometry **ST_CurveToLine**(geometry curveGeom, float tolerance, integer tolerance_type, integer flags);

Description

Converts a CIRCULAR STRING to regular LINESTRING or CURVEPOLYGON to POLYGON or MULTISURFACE to MULTIPOLYGON. Useful for outputting to devices that can't support CIRCULARSTRING geometry types

Converts a given geometry to a linear geometry. Each curved geometry or segment is converted into a linear approximation using the given `tolerance` and options (32 segments per quadrant and no options by default).

The 'tolerance_type' argument determines interpretation of the `tolerance` argument. It can take the following values:

- 0 (default): Tolerance is max segments per quadrant.
- 1: Tolerance is max-deviation of line from curve, in source units.
- 2: Tolerance is max-angle, in radians, between generating radii.

The 'flags' argument is a bitfield. 0 by default. Supported bits are:

- 1: Symmetric (orientation independent) output.
- 2: Retain angle, avoids reducing angles (segment lengths) when producing symmetric output. Has no effect when Symmetric flag is off.

Availability: 1.3.0

Enhanced: 2.4.0 added support for max-deviation and max-angle tolerance, and for symmetric output.

Enhanced: 3.0.0 implemented a minimum number of segments per linearized arc to prevent topological collapse.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

 This method implements the SQL/MM specification. SQL-MM 3: 7.1.7

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

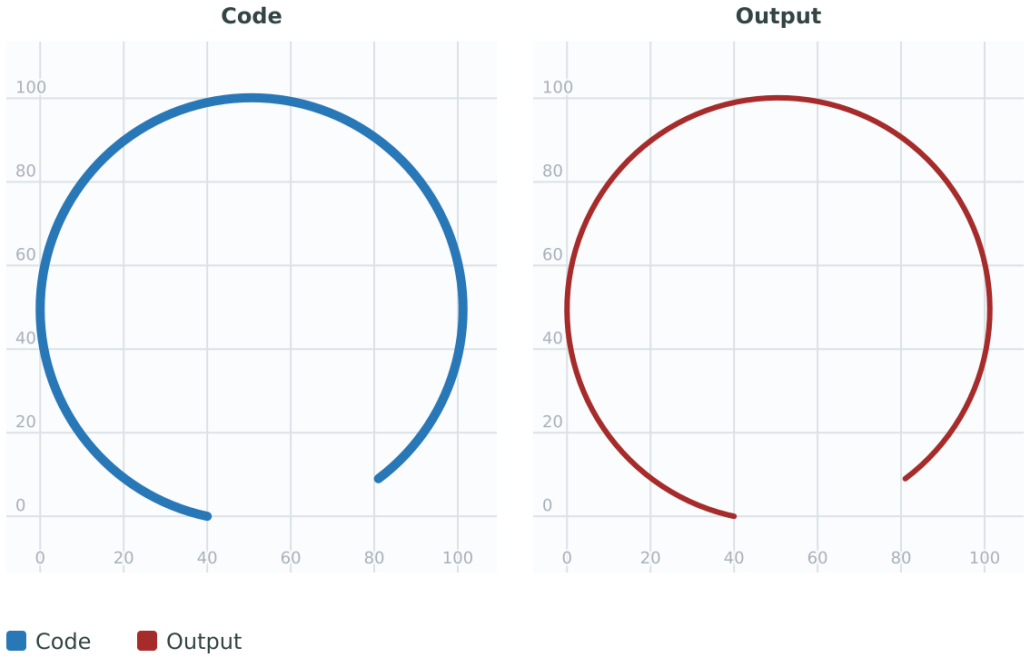
```
SELECT ST_CurveToLine('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)');
```

Output

```
 LINESTRING(220268 150415,220269.95064912 150416.539364228,220271.823415575 150418.17258804,220273.613787707 ↩
    150419.895736857,
220275.317452352 150421.704659462,220276.930305234 150423.594998003,220278.448460847 150425.562198489,
220279.868261823 150427.60152176,220281.186287736 150429.708054909,220282.399363347 150431.876723113,
220283.50456625 150434.10230186,220284.499233914 150436.379429536,220285.380970099 150438.702620341,220286.147650624 ↩
    150441.066277505,
220286.797428488 150443.464706771,220287.328738321 150445.892130112,220287.740300149 150448.342699654,
220288.031122486 150450.810511759,220288.200504713 150453.289621251,220288.248038775 150455.77405574,
220288.173610157 150458.257830005,220287.977398166 150460.734960415,220287.659875492 150463.199479347,
220287.221807076 150465.64544956,220286.664248262 150468.066978495,220285.988542259 150470.458232479,220285.196316903 ↩
    150472.81345077,
220284.289480732 150475.126959442,220283.270218395 150477.39318505,220282.140985384 150479.606668057,
220280.90450212 150481.762075989,220279.5637474 150483.85421628,220278.12195122 150485.87804878,
220276.582586992 150487.828697901,220274.949363179 150489.701464356,220273.226214362 150491.491836488,
220271.417291757 150493.195501133,220269.526953216 150494.808354014,220267.559752731 150496.326509628,
220265.520429459 150497.746310603,220263.41389631 150499.064336517,220261.245228106 150500.277412127,
220259.019649359 150501.38261503,220256.742521683 150502.377282695,220254.419330878 150503.259018879,
220252.055673714 150504.025699404,220249.657244448 150504.675477269,220247.229821107 150505.206787101,
220244.779251566 150505.61834893,220242.311439461 150505.909171266,220239.832329968 150506.078553494,
220237.347895479 150506.126087555,220234.864121215 150506.051658938,220232.386990804 150505.855446946,
220229.922471872 150505.537924272,220227.47650166 150505.099855856,220225.054972724 150504.542297043,
220222.663718741 150503.86659104,220220.308500449 150503.074365683,
220217.994991777 150502.167529512,220215.72876617 150501.148267175,
220213.515283163 150500.019034164,220211.35987523 150498.7825509,
220209.267734939 150497.441796181,220207.243902439 150496,
220205.293253319 150494.460635772,220203.420486864 150492.82741196,220201.630114732 150491.104263143,
220199.926450087 150489.295340538,220198.313597205 150487.405001997,220196.795441592 150485.437801511,
220195.375640616 150483.39847824,220194.057614703 150481.291945091,220192.844539092 150479.123276887,220191.739336189 ↩
    150476.89769814,
220190.744668525 150474.620570464,220189.86293234 150472.297379659,220189.096251815 150469.933722495,
220188.446473951 150467.535293229,220187.915164118 150465.107869888,220187.50360229 150462.657300346,
220187.212779953 150460.189488241,220187.043397726 150457.710378749,220186.995863664 150455.22594426,
220187.070292282 150452.742169995,220187.266504273 150450.265039585,220187.584026947 150447.800520653,
220188.022095363 150445.35455044,220188.579654177 150442.933021505,220189.25536018 150440.541767521,
```

```
220190.047585536 150438.18654923,220190.954421707 150435.873040558,220191.973684044 150433.60681495,
220193.102917055 150431.393331943,220194.339400319 150429.237924011,220195.680155039 150427.14578372,220197.12195122 ↩
    150425.12195122,
220198.661315447 150423.171302099,220200.29453926 150421.298535644,220202.017688077 150419.508163512,220203.826610682 ↩
    150417.804498867,
220205.716949223 150416.191645986,220207.684149708 150414.673490372,220209.72347298 150413.253689397,220211.830006129 ↩
    150411.935663483,
220213.998674333 150410.722587873,220216.22425308 150409.61738497,220218.501380756 150408.622717305,220220.824571561 ↩
    150407.740981121,
220223.188228725 150406.974300596,220225.586657991 150406.324522731,220227 150406)
```

Figure



This example uses 3D input.

Code

```
SELECT ST_CurveToLine('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)');
```

Output

```
 LINESTRING(220268 150415 1,220269.95064912 150416.539364228 1.0181172856673,
220271.823415575 150418.17258804 1.03623457133459,220273.613787707 150419.895736857 1.05435185700189,....AD INFINITUM ↩
    ....
   220225.586657991 150406.324522731 1.32611114201132,220227 150406 3)
```

Use only 2 segments to approximate quarter circle.

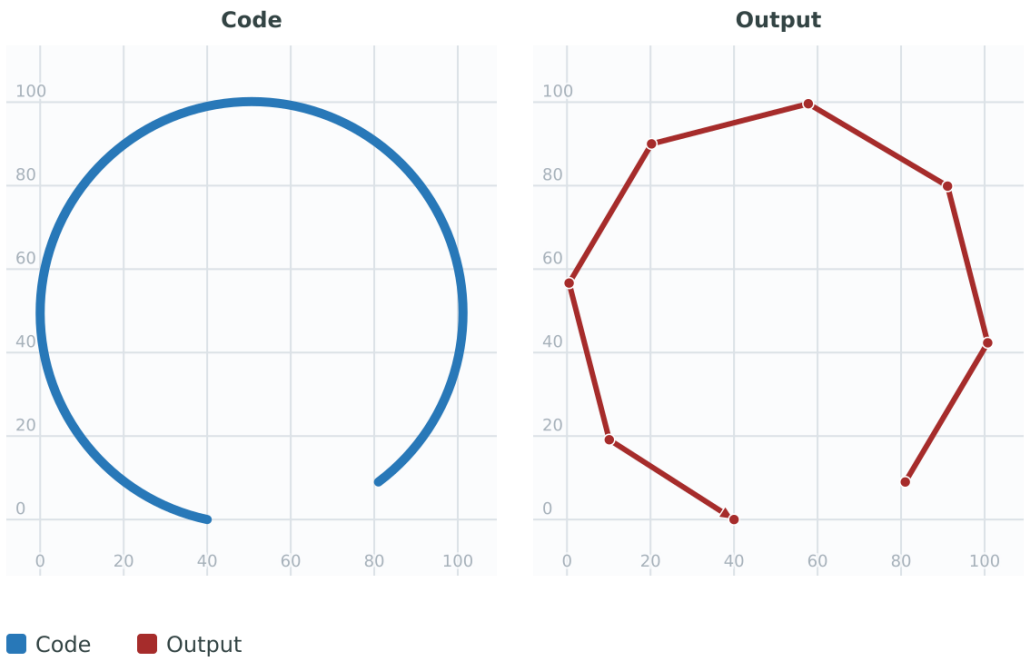
Code

```
SELECT ST_CurveToLine('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)', 2);
```

Output

```
 LINESTRING(220268 150415,220287.740300149 150448.342699654,220278.12195122 150485.87804878,
220244.779251566 150505.61834893,220207.243902439 150496,220187.50360229 150462.657300346,
220197.12195122 150425.12195122,220227 150406)
```

Figure



Ensure the approximated line is no more than 20 units away from the original curve, and make the result direction-neutral.

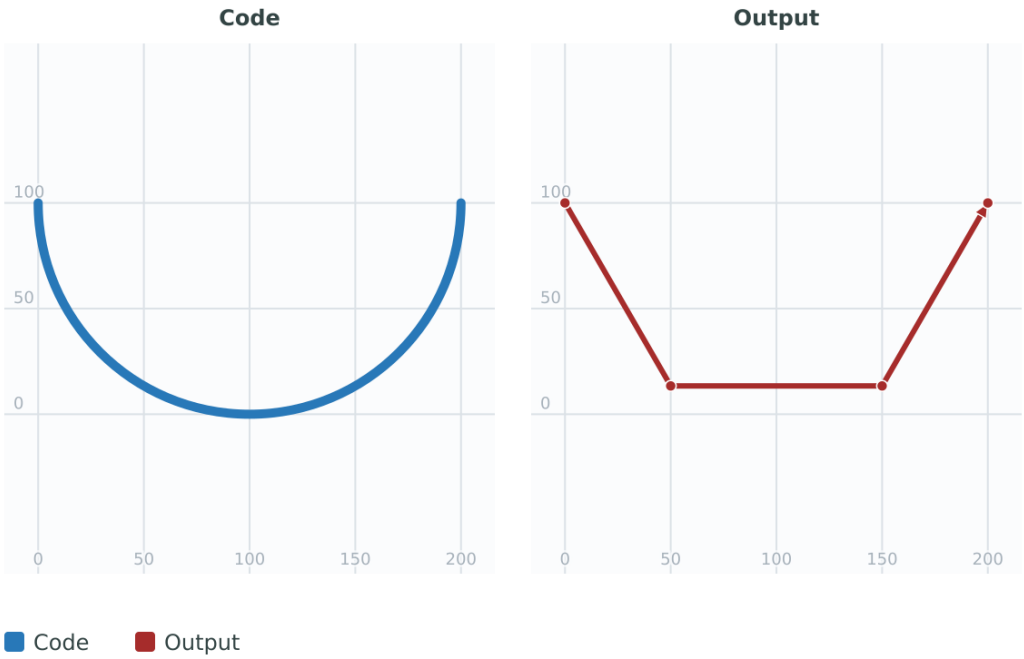
Code

```
SELECT ST_CurveToLine(
    'CIRCULARSTRING(0 0,100 -100,200 0)'::geometry,
    20,
    1,
    1
  );
```

Output

```
LINESTRING(0 0,49.99999999999996 -86.60254037844383,149.99999999999994 -86.6025403784439,200 0)
```

Figure



See Also

ST_LineToCurve

7.5.5 ST_Scroll

ST_Scroll — Change start point of a closed LineString.

Synopsis

geometry **ST_Scroll**(geometry linestring, geometry point);

Description

Changes the start/end point of a closed LineString to the given vertex *point*.

Availability: 3.2.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Examples

Make a closed line start at its 3rd vertex.

Code

```
SELECT ST_Scroll('SRID=4326;LINESTRING(0 0 0 1,10 0 2 0,5 5 4 2,0 0 0 1)', 'POINT(5 5 4 2)') ↩
    ;
```

Output

```
SRID=4326;LINESTRING(5 5 4 2,0 0 0 1,10 0 2 0,5 5 4 2)
```

Figure



See Also

ST_Normalize

7.5.6 ST_FlipCoordinates

ST_FlipCoordinates — Returns a version of a geometry with X and Y axis flipped.

Synopsis

geometry **ST_FlipCoordinates**(geometry geom);

Description

Returns a version of the given geometry with X and Y axis flipped. Useful for fixing geometries which contain coordinates expressed as latitude/longitude (Y,X).

Availability: 2.0.0

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Example

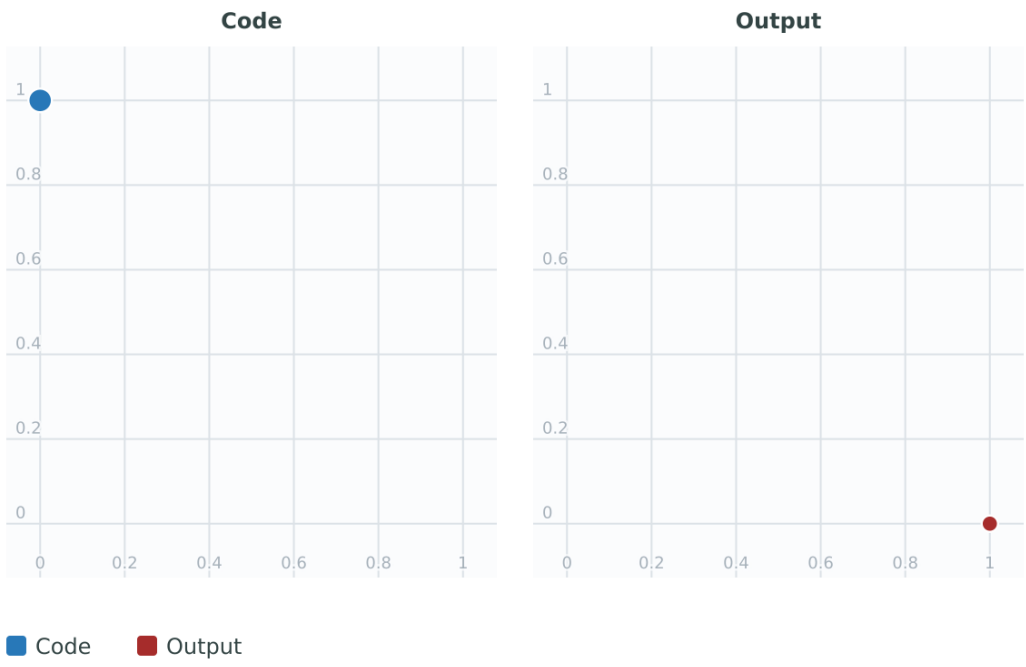
Code

```
SELECT ST_FlipCoordinates(GeomFromEWKT('POINT(1 2)'));
```

Output

```
POINT(2 1)
```

Figure



See Also

ST_SwapOrdinates

7.5.7 ST_Force2D

ST_Force2D — Force the geometries into a "2-dimensional mode".

Synopsis

geometry **ST_Force2D**(geometry geomA);

Description

Forces the geometries into a "2-dimensional mode" so that all output representations will only have the X and Y coordinates. This is useful for force OGC-compliant output (since OGC only specifies 2-D geometries).

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

Changed: 2.1.0. Up to 2.0.x this was called ST_Force_2D.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

Examples

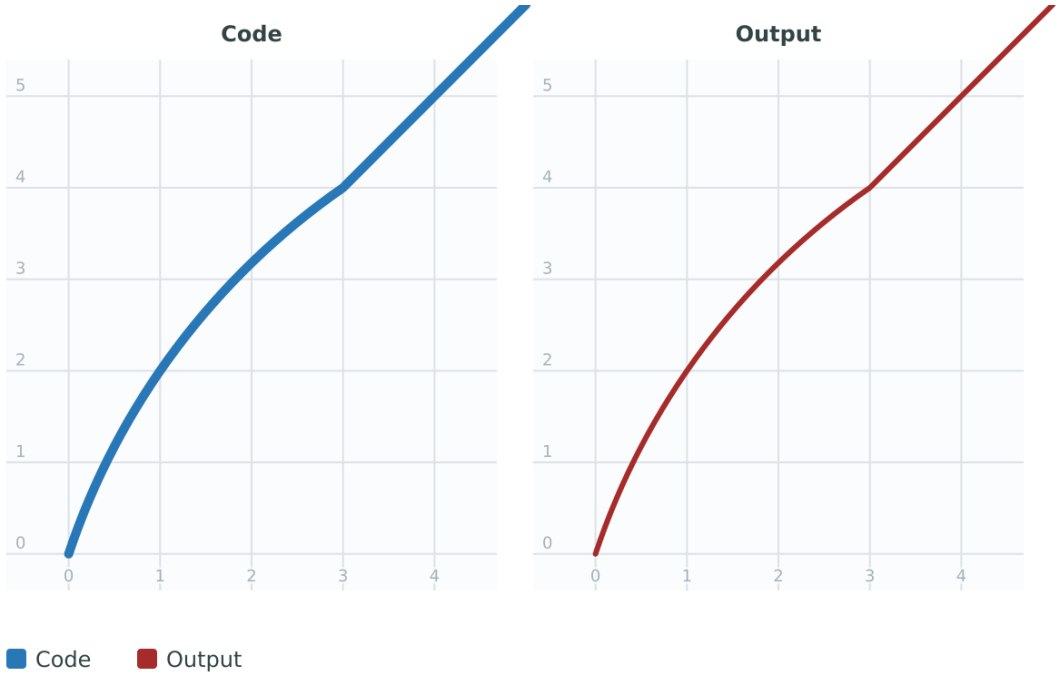
Code

```
SELECT ST_Force2D('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1,2 3,4 5,6 7,5 6)
```

Figure



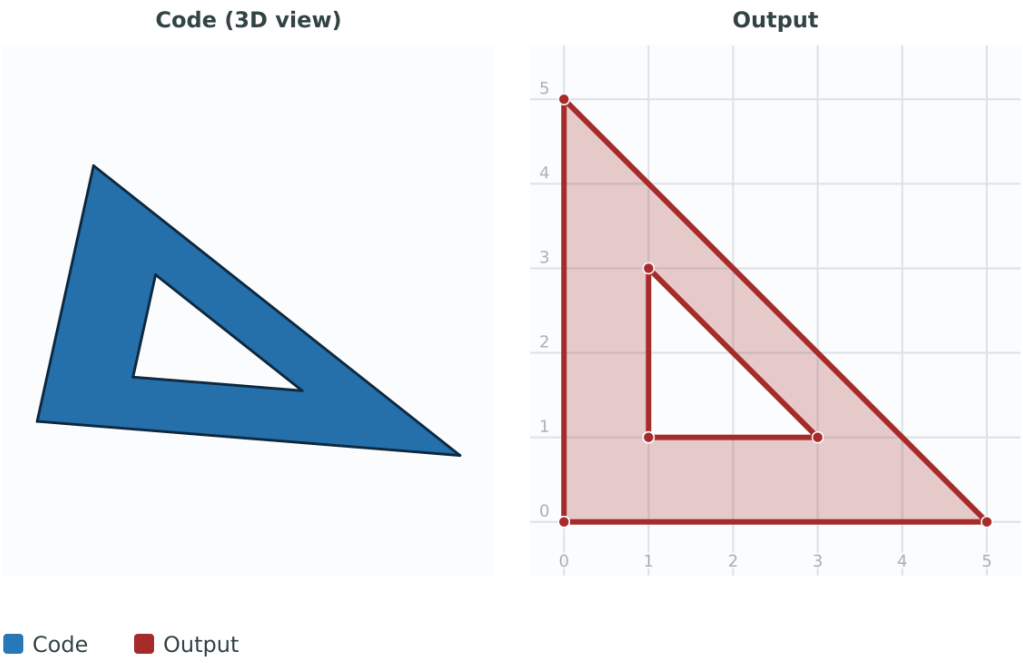
Code

```
SELECT ST_Force2D('POLYGON((0 0 2,0 5 2,5 0 2,0 0 2),(1 1 2,3 1 2,1 3 2,1 1 2))');
```

Output

```
POLYGON((0 0,0 5,5 0,0 0),(1 1,3 1,1 3,1 1))
```

Figure



See Also

ST_Force3D

7.5.8 ST_Force3D

ST_Force3D — Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.

Synopsis

geometry **ST_Force3D**(geometry geomA, float Zvalue = 0.0);

Description

Forces the geometries into XYZ mode. This is an alias for ST_Force3DZ. If a geometry has no Z component, then a `Zvalue` Z coordinate is tacked on.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

Changed: 2.1.0. Up to 2.0.x this was called ST_Force_3D.

Changed: 3.1.0. Added support for supplying a non-zero Z value.

This function supports Polyhedral surfaces.

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

Examples

Nothing happens to an already 3D geometry.

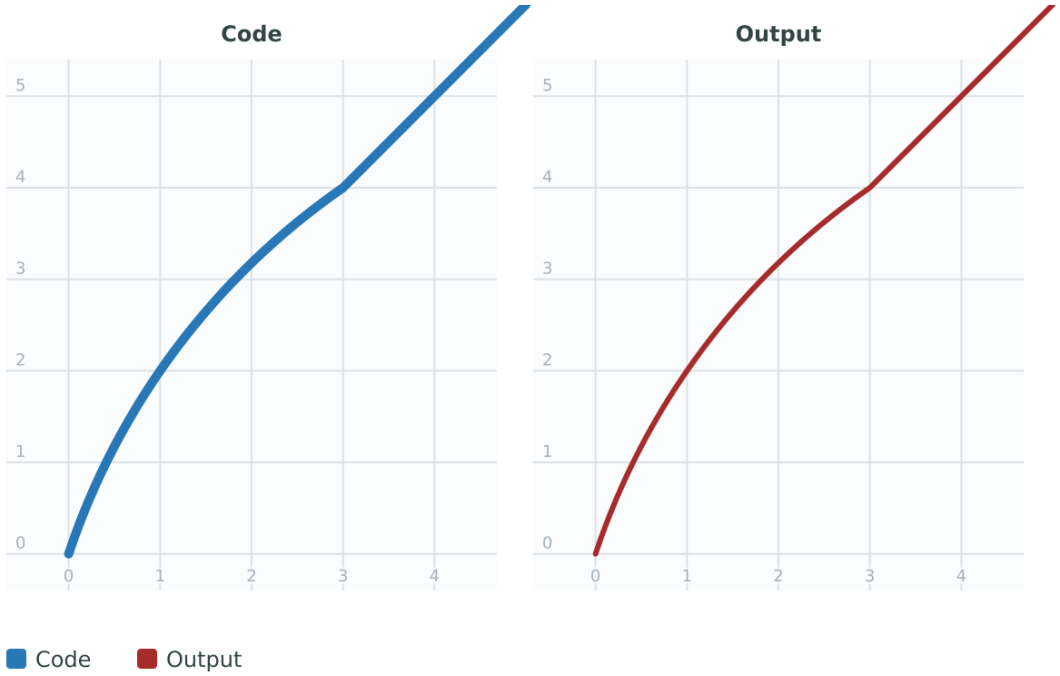
Code

```
SELECT ST_Force3D('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)
```

Figure



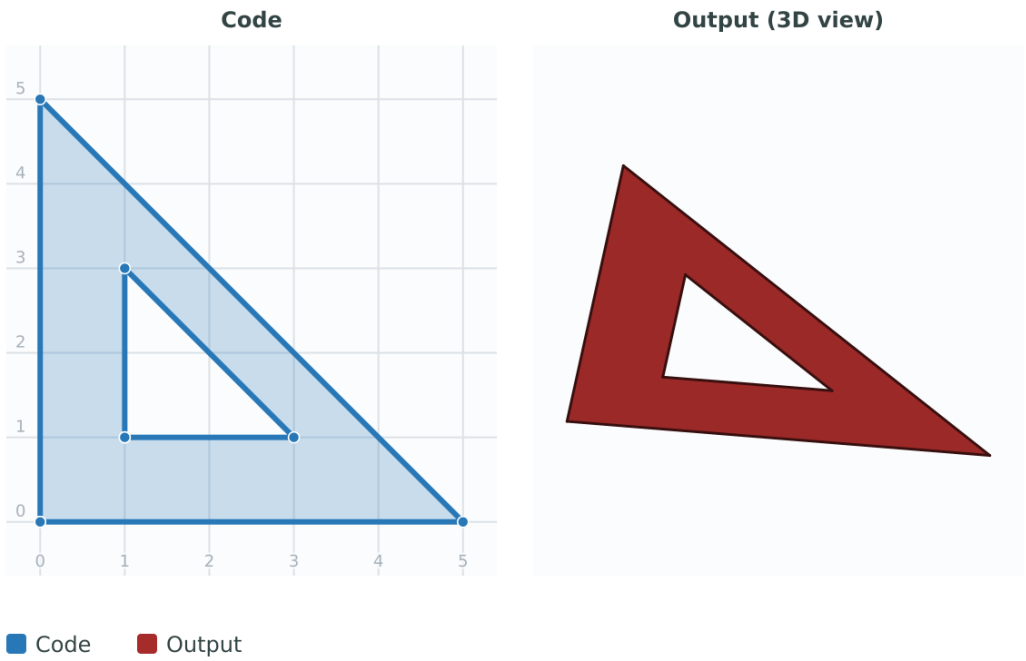
Code

```
SELECT ST_Force3D('POLYGON((0 0,0 5,5 0,0 0),(1 1,3 1,1 3,1 1))');
```

Output

```
POLYGON((0 0 0,0 5 0,5 0 0,0 0 0),(1 1 0,3 1 0,1 3 0,1 1 0))
```

Figure



See Also

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3DZ

7.5.9 ST_Force3DZ

ST_Force3DZ — Force the geometries into XYZ mode.

Synopsis

geometry **ST_Force3DZ**(geometry geomA, float Zvalue = 0.0);

Description

Forces the geometries into XYZ mode. If a geometry has no Z component, then a `Zvalue` Z coordinate is tacked on.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

Changed: 2.1.0. Up to 2.0.x this was called ST_Force_3DZ.

Changed: 3.1.0. Added support for supplying a non-zero Z value.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

Nothing happens to an already 3D geometry.

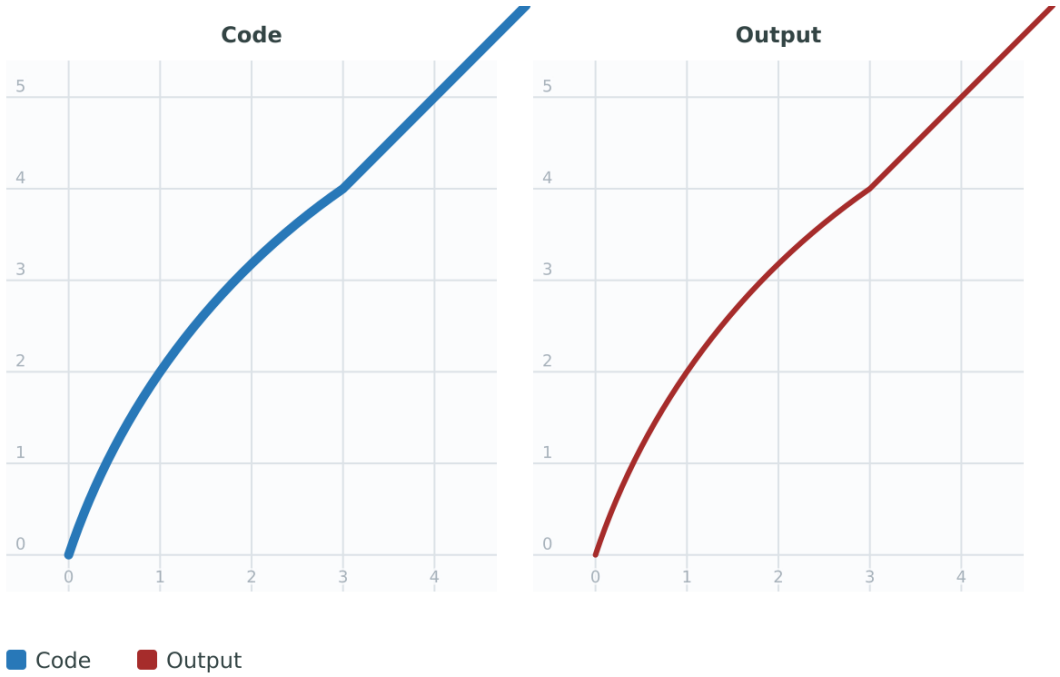
Code

```
SELECT ST_Force3DZ('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)
```

Figure



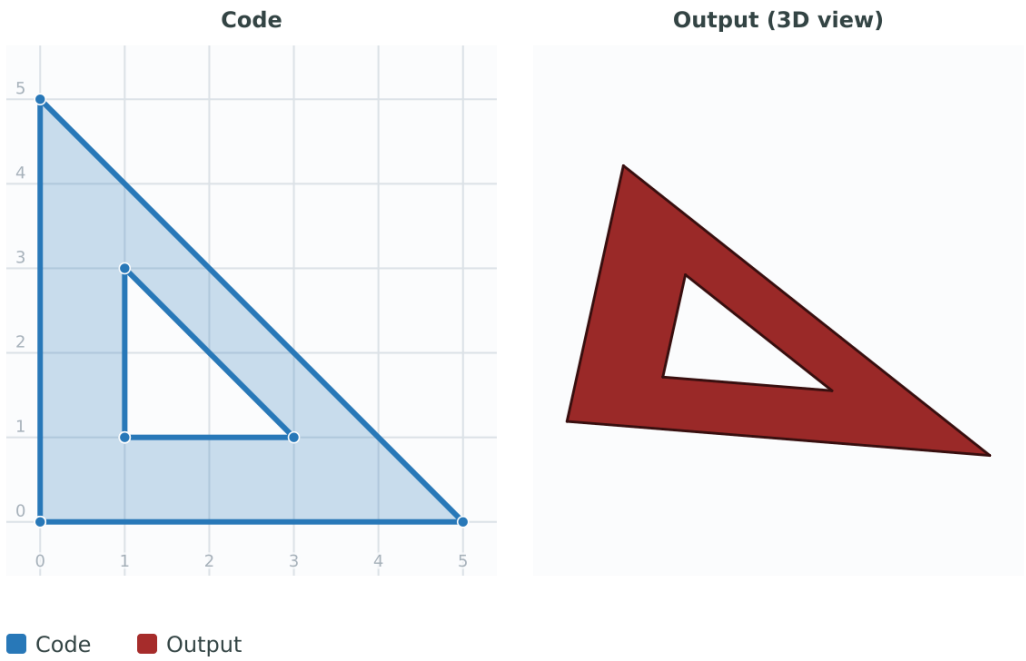
Code

```
SELECT ST_Force3DZ('POLYGON((0 0,0 5,5 0,0 0),(1 1,3 1,1 3,1 1))');
```

Output

```
POLYGON((0 0 0,0 5 0,5 0 0,0 0 0),(1 1 0,3 1 0,1 3 0,1 1 0))
```

Figure



See Also

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D

7.5.10 ST_Force3DM

ST_Force3DM — Force the geometries into XYM mode.

Synopsis

geometry **ST_Force3DM**(geometry geomA, float Mvalue = 0.0);

Description

Forces the geometries into XYM mode. If a geometry has no M component, then a `Mvalue` M coordinate is tacked on. If it has a Z component, then Z is removed

Changed: 2.1.0. Up to 2.0.x this was called ST_Force_3DM.

Changed: 3.1.0. Added support for supplying a non-zero M value.

This method supports Circular Strings and Curves.

Examples

Nothing happens to an already 3D geometry.

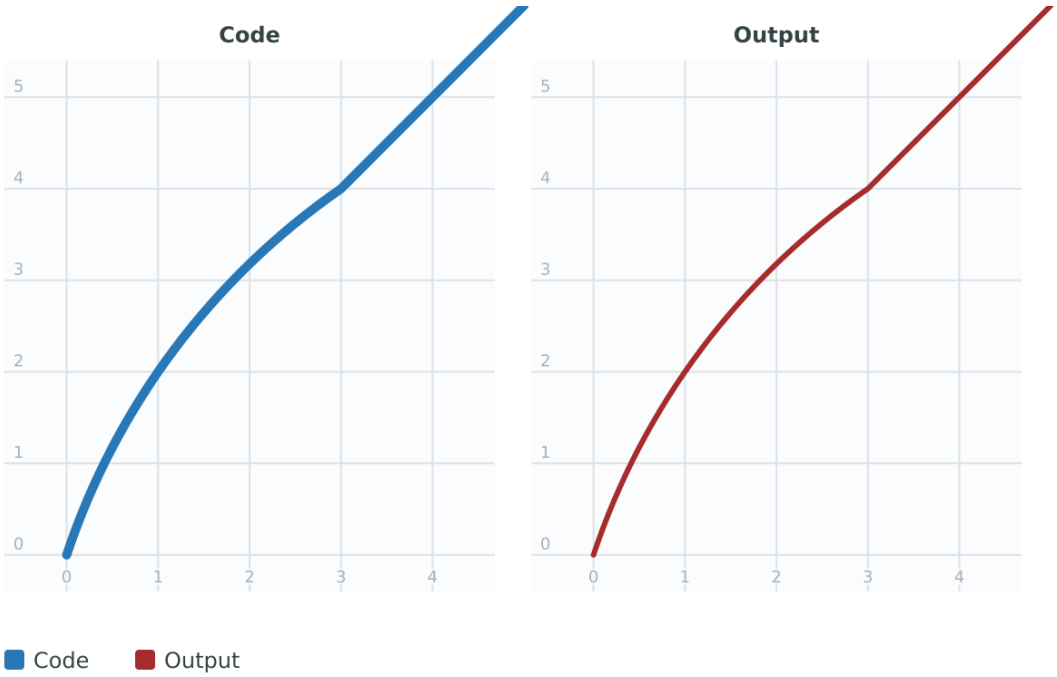
Code

```
SELECT ST_Force3DM('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRINGM(1 1 0,2 3 0,4 5 0,6 7 0,5 6 0)
```

Figure



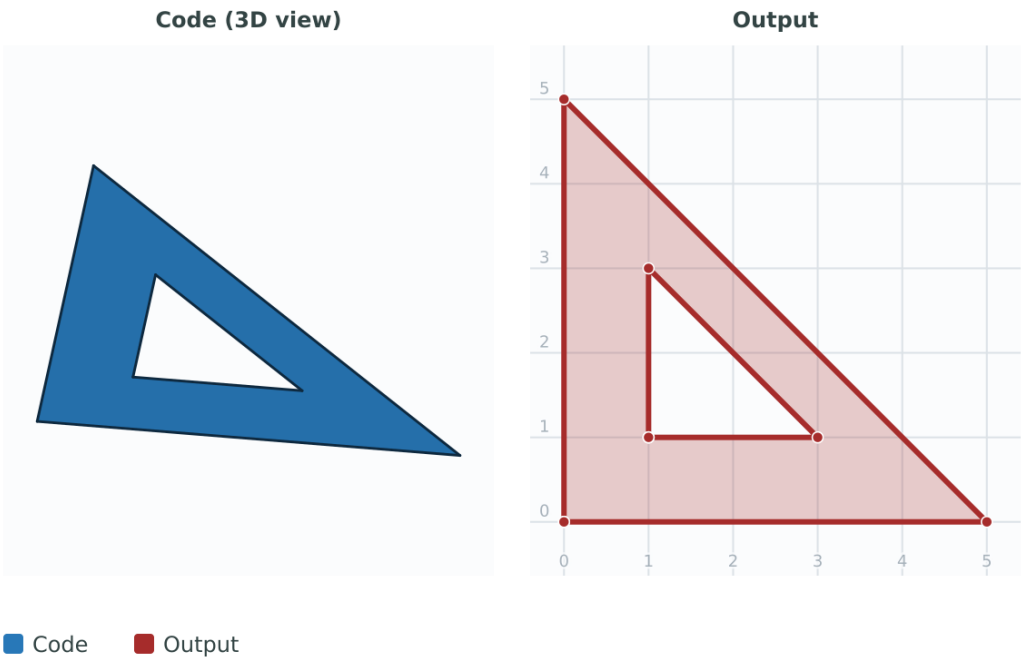
Code

```
SELECT ST_Force3DM('POLYGON((0 0 1,0 5 1,5 0 1,0 0 1),(1 1 1,3 1 1,1 3 1,1 1 1))');
```

Output

```
POLYGONM((0 0 0,0 5 0,5 0 0,0 0 0),(1 1 0,3 1 0,1 3 0,1 1 0))
```

Figure



See Also

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D, ST_GeomFromEWKT

7.5.11 ST_Force4D

ST_Force4D — Force the geometries into XYZM mode.

Synopsis

geometry **ST_Force4D**(geometry geomA, float Zvalue = 0.0, float Mvalue = 0.0);

Description

Forces the geometries into XYZM mode. `Zvalue` and `Mvalue` is tacked on for missing Z and M dimensions, respectively.

Changed: 2.1.0. Up to 2.0.x this was called ST_Force_4D.

Changed: 3.1.0. Added support for supplying non-zero Z and M values.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

Nothing happens to an already 3D geometry.

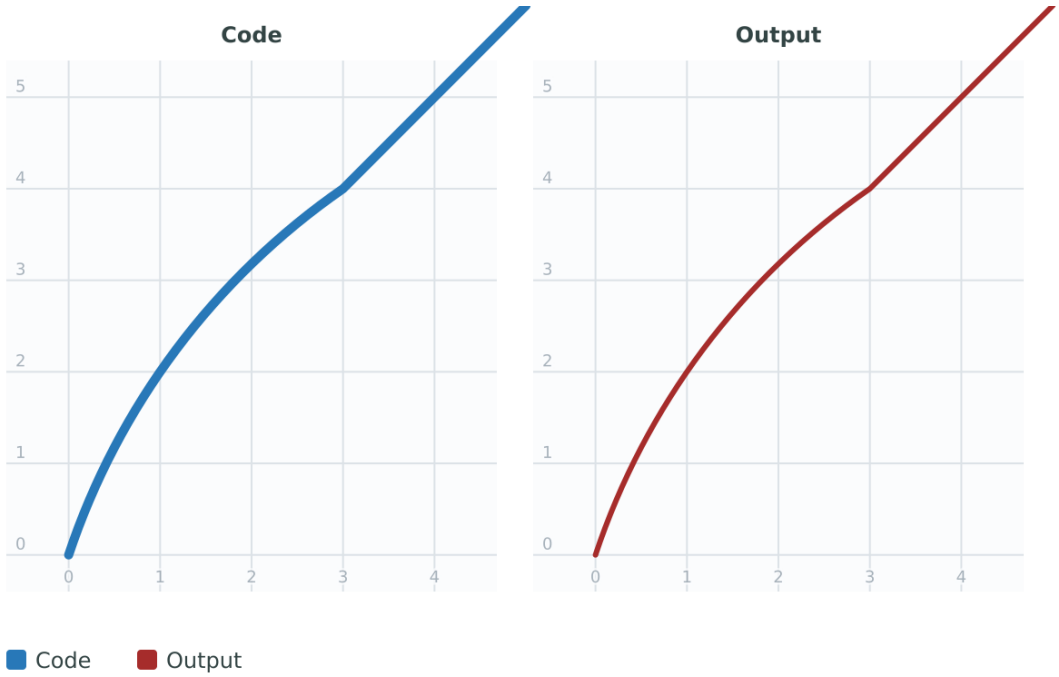
Code

```
SELECT ST_Force4D('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1 2 0,2 3 2 0,4 5 2 0,6 7 2 0,5 6 2 0)
```

Figure



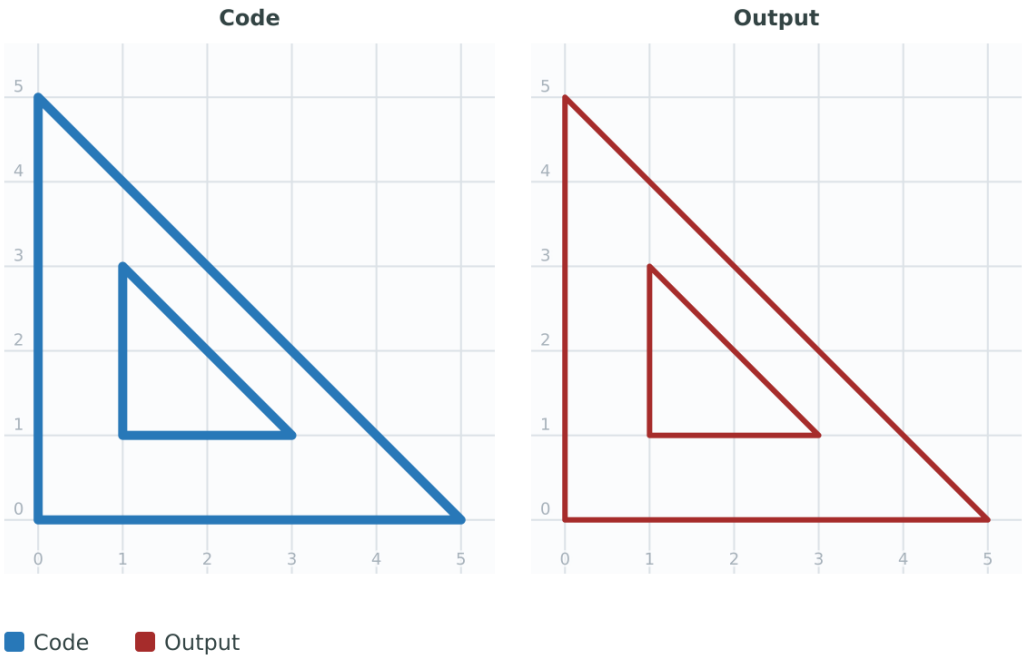
Code

```
SELECT ST_Force4D('MULTILINESTRINGM((0 0 1,0 5 2,5 0 3,0 0 4),(1 1 1,3 1 1,1 3 1,1 1 1))');
```

Output

```
MULTILINESTRING((0 0 0 1,0 5 0 2,5 0 0 3,0 0 0 4),(1 1 0 1,3 1 0 1,1 3 0 1,1 1 0 1))
```

Figure



See Also

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D

7.5.12 ST_ForceCollection

ST_ForceCollection — Convert the geometry into a GEOMETRYCOLLECTION.

Synopsis

geometry **ST_ForceCollection**(geometry geomA);

Description

Converts the geometry into a GEOMETRYCOLLECTION. This is useful for simplifying the WKB representation.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

Availability: 1.2.2, prior to 1.3.4 this function will crash with Curves. This is fixed in 1.3.4+

Changed: 2.1.0. Up to 2.0.x this was called ST_Force_Collection.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

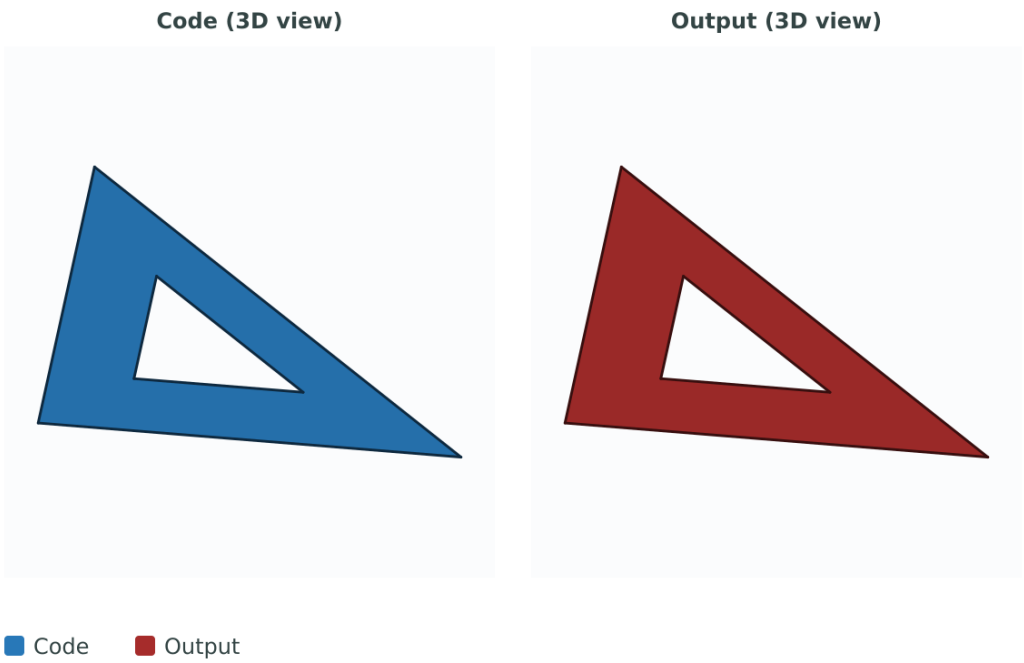
Code

```
SELECT ST_ForceCollection('POLYGON((0 0 1,0 5 1,5 0 1,0 0 1),(1 1 1,3 1 1,1 3 1,1 1 1))');
```

Output

```
GEOMETRYCOLLECTION(POLYGON((0 0 1,0 5 1,5 0 1,0 0 1),(1 1 1,3 1 1,1 3 1,1 1 1)))
```

Figure



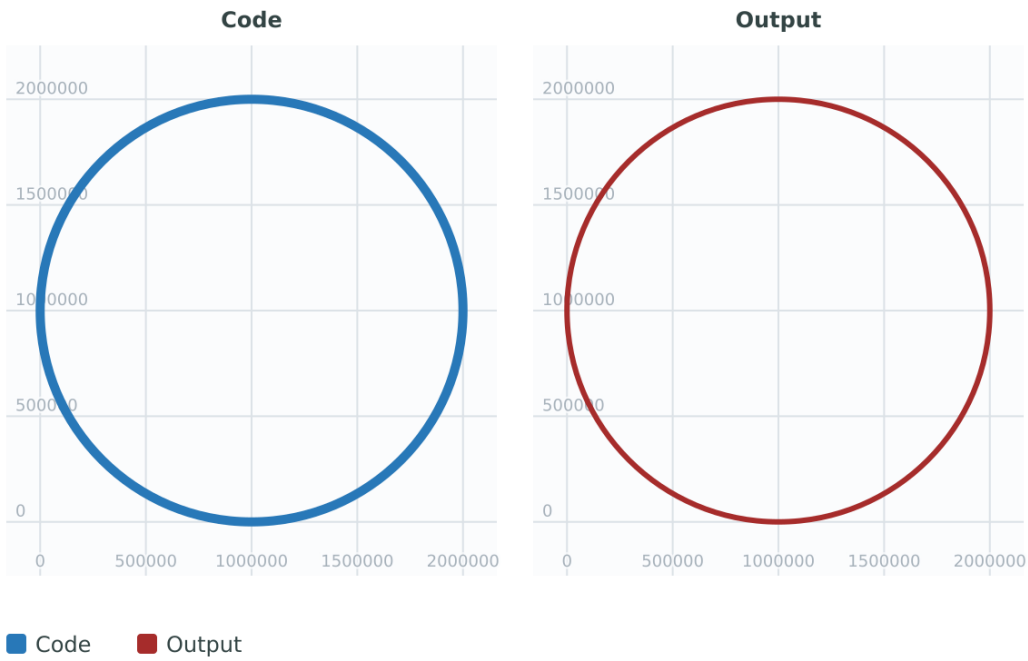
Code

```
SELECT ST_ForceCollection('CIRCULARSTRING(220227 150406,2220227 150407,220227 150406)');
```

Output

```
GEOMETRYCOLLECTION(CIRCULARSTRING(220227 150406,2220227 150407,220227 150406))
```

Figure



This example uses a PolyhedralSurface.

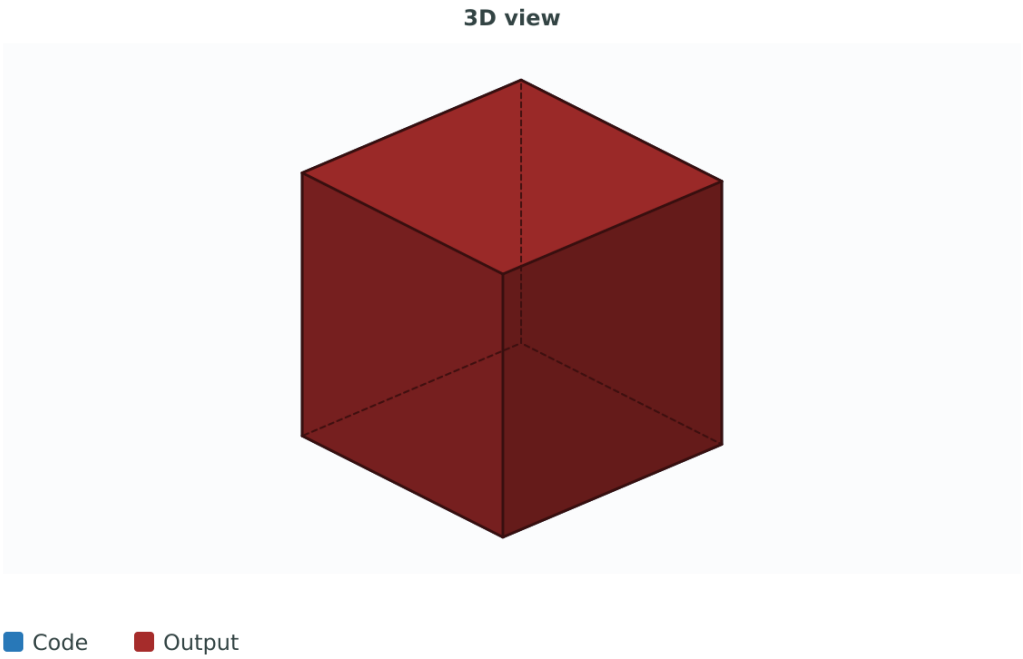
Code

```
SELECT ST_ForceCollection('POLYHEDRALSURFACE(((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
  ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
  ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
  ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
  ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
  ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)))');
```

Output

```
GEOMETRYCOLLECTION(
  POLYGON((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
  POLYGON((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
  POLYGON((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
  POLYGON((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
  POLYGON((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
  POLYGON((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)
```

Figure



See Also

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D, ST_GeomFromEWKT

7.5.13 ST_ForceCurve

ST_ForceCurve — Upcast a geometry into its curved type, if applicable.

Synopsis

geometry **ST_ForceCurve**(geometry g);

Description

Turns a geometry into its curved representation, if applicable: lines become compoundcurves, multilines become multicurves polygons become curvepolygons multipolygons become multisurfaces. If the geometry input is already a curved representation returns back same as input.

Availability: 2.2.0

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

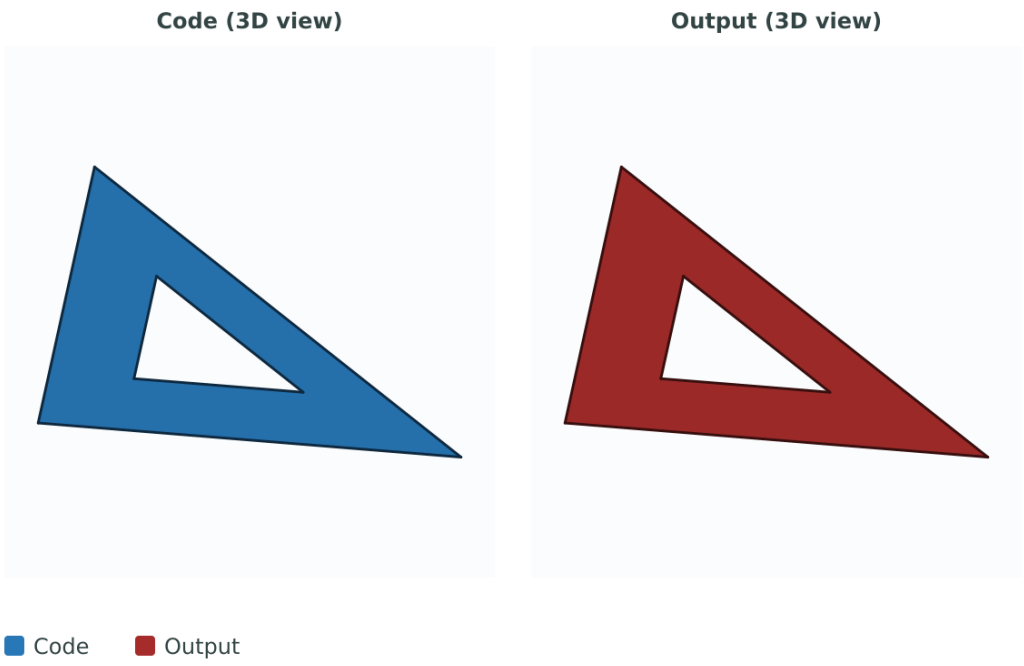
Code

```
SELECT ST_ForceCurve('POLYGON((0 0 2,5 0 2,0 5 2,0 0 2),(1 1 2,1 3 2,3 1 2,1 1 2))':: ↩
    geometry
  );
```

Output

```
CURVEPOLYGON Z ((0 0 2,5 0 2,0 5 2,0 0 2),(1 1 2,1 3 2,3 1 2,1 1 2))
```

Figure



See Also

ST_LineToCurve

7.5.14 ST_ForcePolygonCCW

ST_ForcePolygonCCW — Orients all exterior rings counter-clockwise and all interior rings clockwise.

Synopsis

geometry **ST_ForcePolygonCCW**(geometry geom);

Description

Forces (Multi)Polygons to use a counter-clockwise orientation for their exterior ring, and a clockwise orientation for their interior rings. Non-polygonal geometries are returned unchanged.

Availability: 2.4.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Examples

Force a polygon and its hole to use counter-clockwise exterior rings and clockwise interior rings.

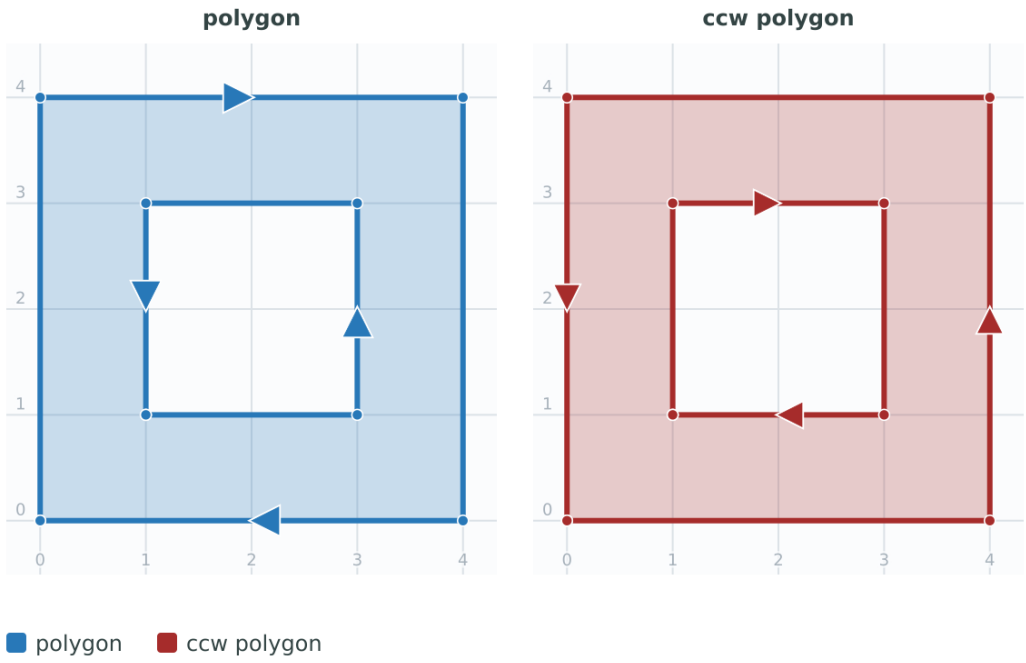
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,0 4,4 4,4 0,0 0),
                  (1 1,3 1,3 3,1 3,1 1))'::geometry AS geom
)
SELECT geom AS input_polygon,
       ST_ForcePolygonCCW(geom) AS ccw_polygon
FROM data;
```

Output

```
POLYGON((0 0,0 4,4 4,4 0,0 0),(1 1,3 1,3 3,1 3,1 1)) | POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,1 3,3 3,3 1,1 1))
```

Figure



See Also

ST_ForcePolygonCW, ST_IsPolygonCCW, ST_IsPolygonCW

7.5.15 ST_ForcePolygonCW

ST_ForcePolygonCW — Orients all exterior rings clockwise and all interior rings counter-clockwise.

Synopsis

geometry **ST_ForcePolygonCW**(geometry geom);

Description

Forces (Multi)Polygons to use a clockwise orientation for their exterior ring, and a counter-clockwise orientation for their interior rings. Non-polygonal geometries are returned unchanged.

Availability: 2.4.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Examples

Force a polygon and its hole to use clockwise exterior rings and counter-clockwise interior rings.

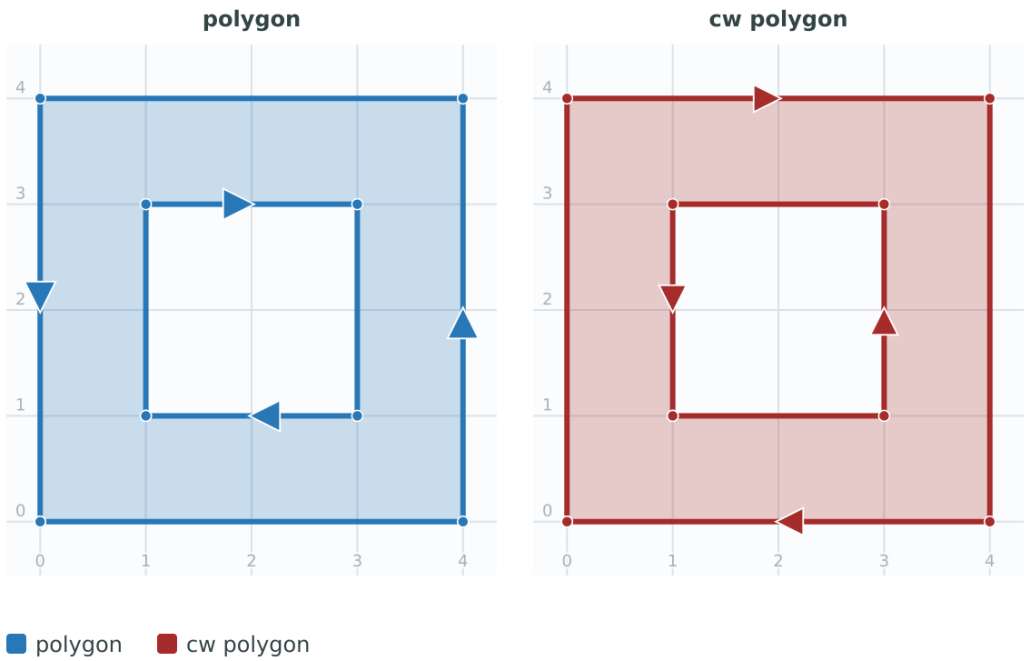
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,4 0,4 4,0 4,0 0),
                  (1 1,1 3,3 3,3 1,1 1))'::geometry AS geom
)
SELECT geom AS input_polygon,
       ST_ForcePolygonCW(geom) AS cw_polygon
FROM data;
```

Output

```
POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,1 3,3 3,3 1,1 1)) | POLYGON((0 0,0 4,4 4,4 0,0 0),(1 1,3 1,3 3,1 3,1 1))
```

Figure



See Also

ST_ForcePolygonCCW, ST_IsPolygonCCW, ST_IsPolygonCW

7.5.16 ST_ForceSFS

ST_ForceSFS — Force geometries to use SFS 1.1 or 1.2 geometry types.

Synopsis

geometry **ST_ForceSFS**(geometry geomA);
geometry **ST_ForceSFS**(geometry geomA, text version);

Description

Forces a geometry to use the geometry types defined by the requested version of the OGC Simple Features Specification. The default is SFS 1.1. Curved geometries are converted to linear representations, `TRIANGLE` is converted to `POLYGON`, and `TIN` and `POLYHEDRALSURFACE` are converted to `GEOMETRYCOLLECTION` values containing `POLYGON` elements.

If *`version`* begins with `1.2`, curved geometries are still converted to linear representations, but `TRIANGLE`, `TIN`, and `POLYHEDRALSU` are preserved. Other version values use the default SFS 1.1 behavior. Geometry collections are processed recursively, and the SRID and Z/M coordinates are preserved.

Availability: 2.1.0

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

Examples

Compare the geometry types produced for SFS 1.1 (the default) and SFS 1.2. Read each row from the input type in the direction of the result columns.

Code

```
WITH inputs(geom) AS (VALUES
    ('CIRCULARSTRING(0 0, 1 1, 2 0)'::geometry),
    ('TRIANGLE((0 0, 0 2, 2 0, 0 0))'::geometry),
    ('TIN(((0 0, 0 2, 2 0, 0 0)))'::geometry),
    ('POLYHEDRALSURFACE(((0 0, 0 2, 2 0, 0 0)))'::geometry)
)
SELECT ST_GeometryType(geom) || ' →' AS input,
       ST_GeometryType(ST_ForceSFS(geom)) AS "SFS 1.1 (default)",
       ST_GeometryType(ST_ForceSFS(geom, '1.2')) AS "SFS 1.2"
FROM inputs;
```

Output

```
         input          |   SFS 1.1 (default)   |       SFS 1.2
------------------------+-----------------------+---------------------
 ST_CircularString →    | ST_LineString         | ST_LineString
 ST_Triangle →          | ST_Polygon            | ST_Triangle
 ST_Tin →               | ST_GeometryCollection | ST_Tin
 ST_PolyhedralSurface → | ST_GeometryCollection | ST_PolyhedralSurface
(4 rows)
```

See Also

ST_CurveToLine, ST_ForceCurve

7.5.17 ST_ForceRHR

ST_ForceRHR — Force the orientation of the vertices in a polygon to follow the Right-Hand-Rule.

Synopsis

geometry **ST_ForceRHR**(geometry g);

Description

Forces the orientation of the vertices in a polygon to follow a Right-Hand-Rule, in which the area that is bounded by the polygon is to the right of the boundary. In particular, the exterior ring is orientated in a clockwise direction and the interior rings in a counter-clockwise direction. This function is a synonym for ST_ForcePolygonCW



Note
The above definition of the Right-Hand-Rule conflicts with definitions used in other contexts. To avoid confusion, it is recommended to use ST_ForcePolygonCW.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.


This function supports 3d and will not drop the z-index.


This function supports Polyhedral surfaces.

Examples

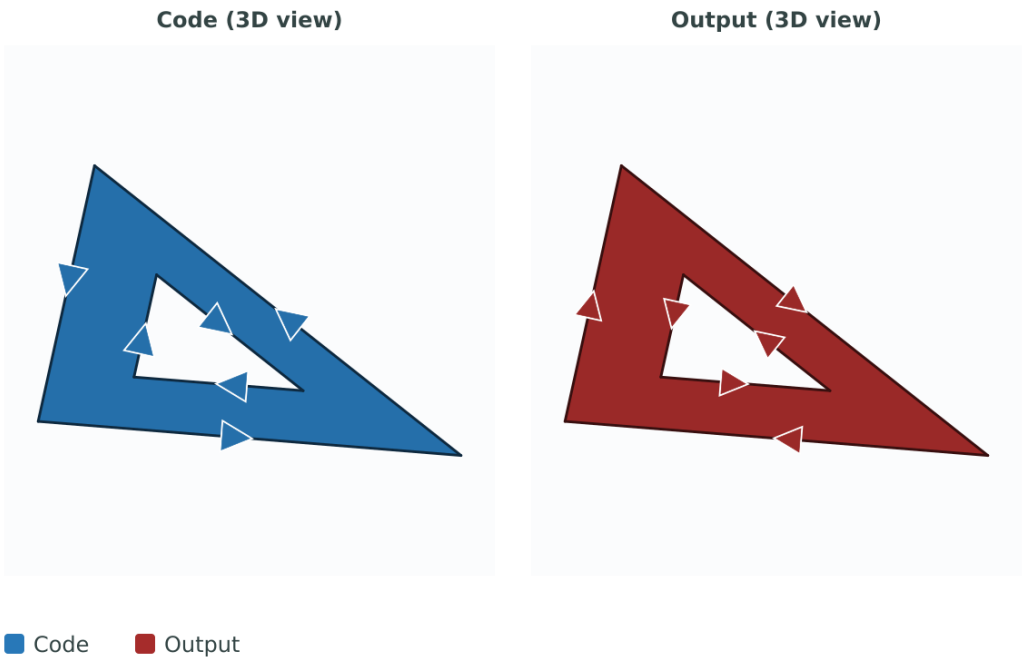
Code

```
SELECT ST_ForceRHR('POLYGON((0 0 2,5 0 2,0 5 2,0 0 2),(1 1 2,1 3 2,3 1 2,1 1 2))'
  );
```

Output

```
POLYGON((0 0 2,0 5 2,5 0 2,0 0 2),(1 1 2,3 1 2,1 3 2,1 1 2))
```

Figure



See Also

ST_ForcePolygonCCW, ST_ForcePolygonCW, ST_IsPolygonCCW, ST_IsPolygonCW, ST_BuildArea, ST_Polygonize, ST_Reverse

7.5.18 ST_LineExtend

ST_LineExtend — Returns a line extended forwards and backwards by specified distances.

Synopsis

geometry **ST_LineExtend**(geometry line, float distance_forward, float distance_backward=0.0);

Description

Returns a line extended forwards and backwards by adding new start (and end) points at the given distance(s). A distance of zero does not add a point. Only non-negative distances are allowed. The direction(s) of the added point(s) is determined by the first (and last) two distinct points of the line. Duplicate points are ignored.

Availability: 3.4.0

Example: Extends a line 5 units forward and 6 units backward

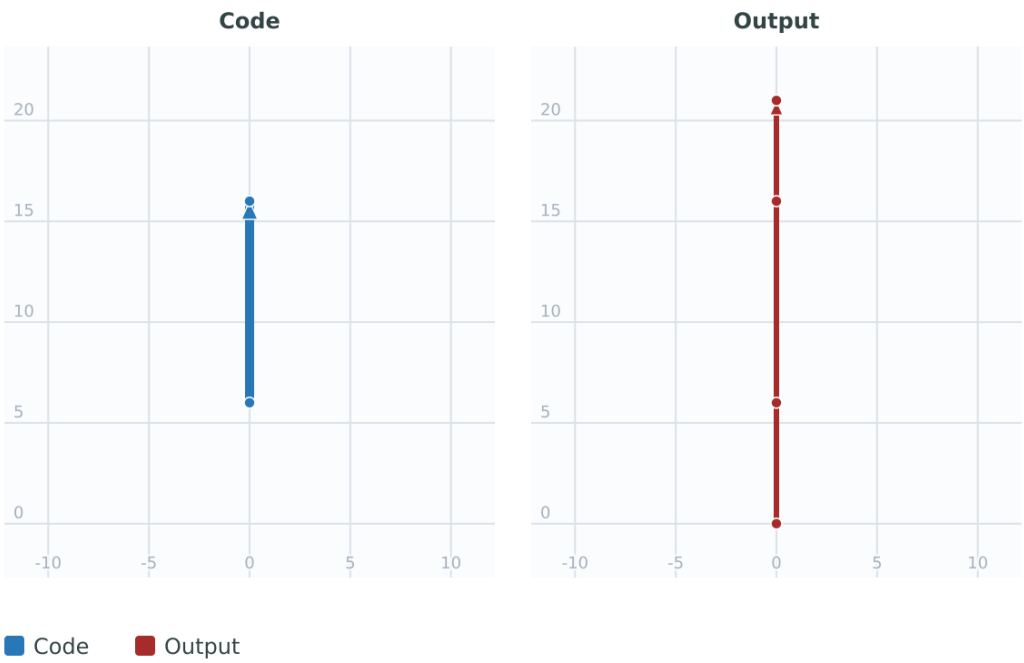
Code

```
SELECT ST_LineExtend('LINESTRING(0 0,0 10)'::geometry, 5, 6);
```

Output

```
LINESTRING(0 -6,0 0,0 10,0 15)
```

Figure



See Also

ST_LineSubstring, ST_LocateAlong, ST_Project

7.5.19 ST_LineToCurve

ST_LineToCurve — Converts a linear geometry to a curved geometry.

Synopsis

geometry **ST_LineToCurve**(geometry geomANoncircular);

Description

Converts plain LINESTRING/POLYGON to CIRCULAR STRINGs and Curved Polygons. Note much fewer points are needed to describe the curved equivalent.

Note
If the input LINESTRING/POLYGON is not curved enough to clearly represent a curve, the function will return the same input geometry.

Availability: 1.3.0

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

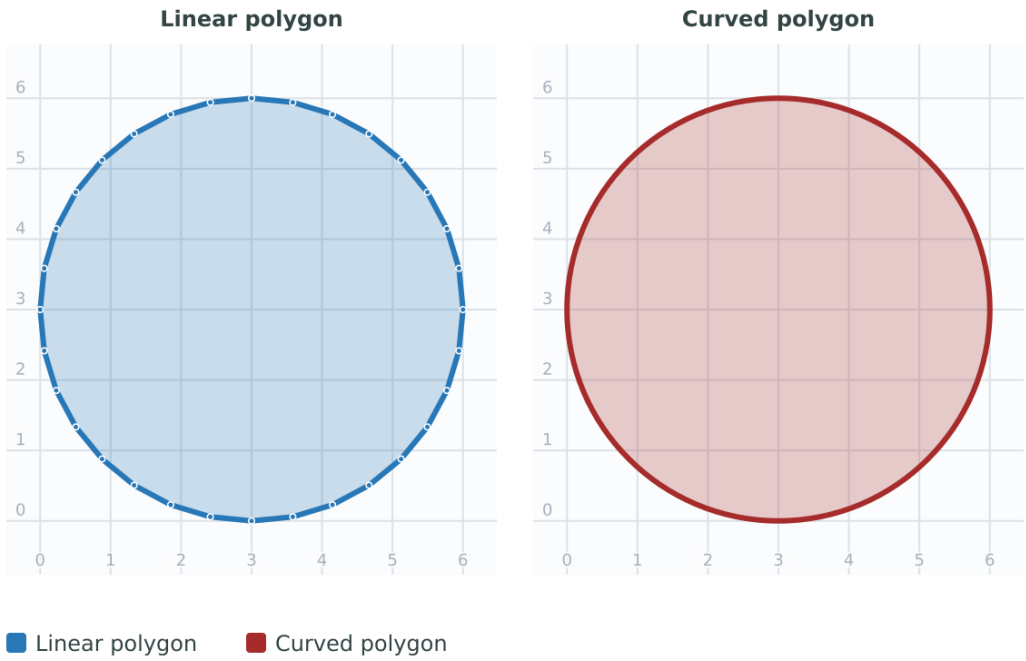
Examples

This example converts a linear Polygon approximating a circle to a CurvePolygon.

Code

```
WITH data AS (
    SELECT ST_Buffer('POINT(1 3)'::geometry, 3) AS geom
)
SELECT geom AS "input_Linear polygon",
       ST_LineToCurve(geom) AS "Curved polygon"
FROM data;
```

Figure

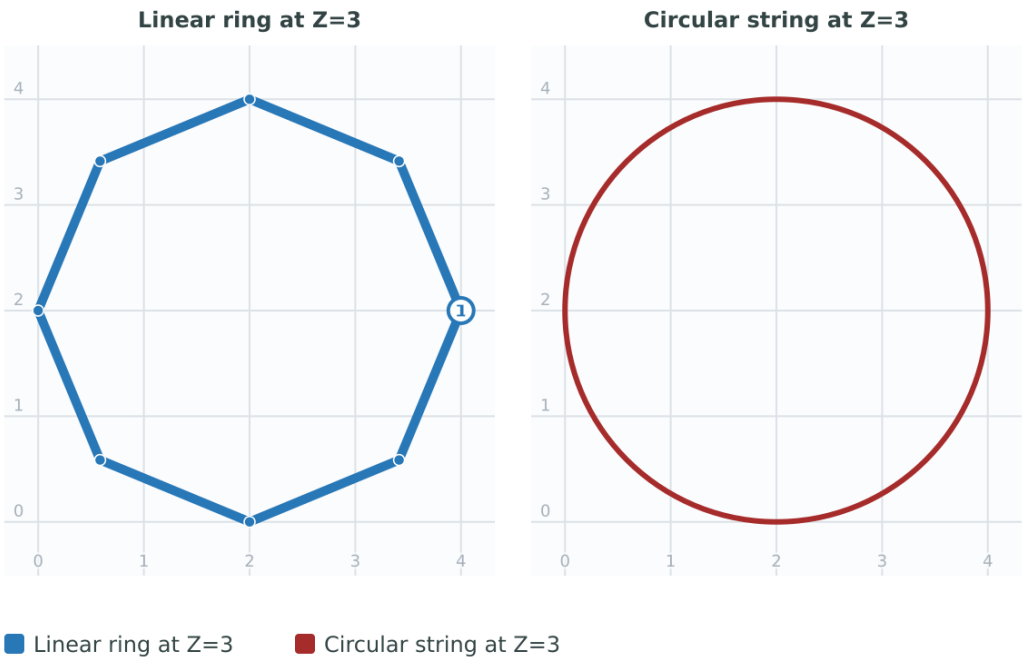


This example converts a 3D linear ring to a CircularString while preserving Z.

Code

```
WITH data AS (
    SELECT ST_Translate(
        ST_Force3D(ST_Boundary(ST_Buffer(ST_Point(1, 3), 2, 2))),
        0, 0, 3
    ) AS geom
)
SELECT geom AS "input_Linear ring at Z=3",
       ST_LineToCurve(geom) AS "Circular string at Z=3"
FROM data;
```

Figure



See Also

ST_CurveToLine

7.5.20 ST_Multi

ST_Multi — Return the geometry as a MULTI* geometry.

Synopsis

geometry **ST_Multi**(geometry geom);

Description

Returns the geometry as a MULTI* geometry collection. If the geometry is already a collection, it is returned unchanged.

Examples

A Polygon is converted to a single-element MultiPolygon. The coordinates are unchanged.

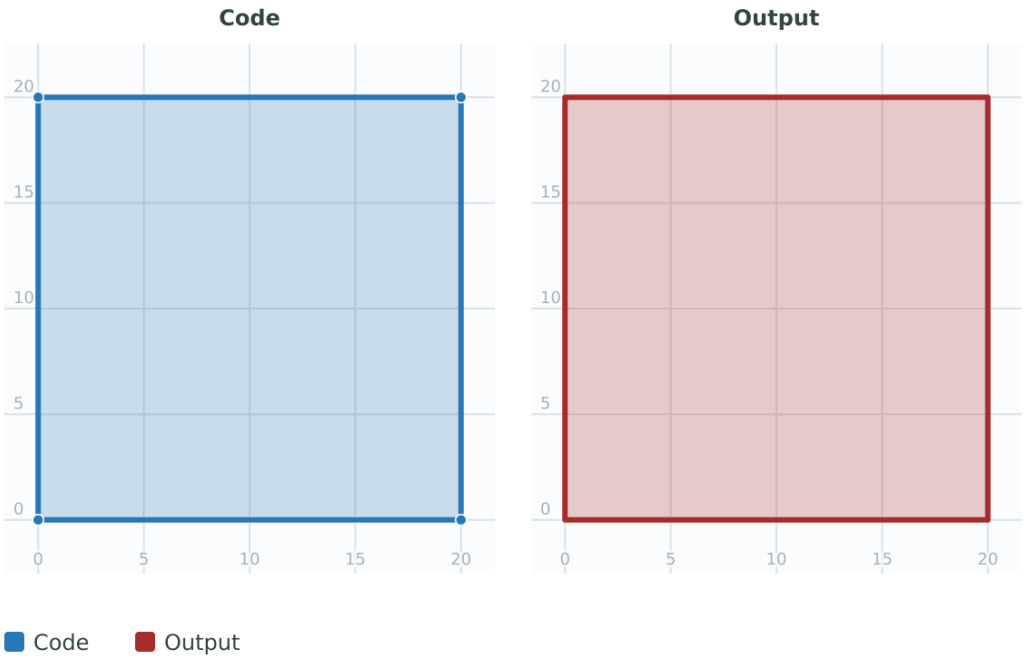
Code

```
SELECT ST_Multi('POLYGON ((10 30,30 30,30 10,10 10,10 30))');
```

Output

```
MULTIPOLYGON(((10 30,30 30,30 10,10 10,10 30)))
```

Figure



See Also

ST_AsText

7.5.21 ST_Normalize

ST_Normalize — Return the geometry in its canonical form.

Synopsis

geometry **ST_Normalize**(geometry geom);

Description

Returns the geometry in its normalized/canonical form. May reorder vertices in polygon rings, rings in a polygon, elements in a multi-geometry complex.

Mostly only useful for testing purposes (comparing expected and obtained results).

Availability: 2.3.0

Examples

Code

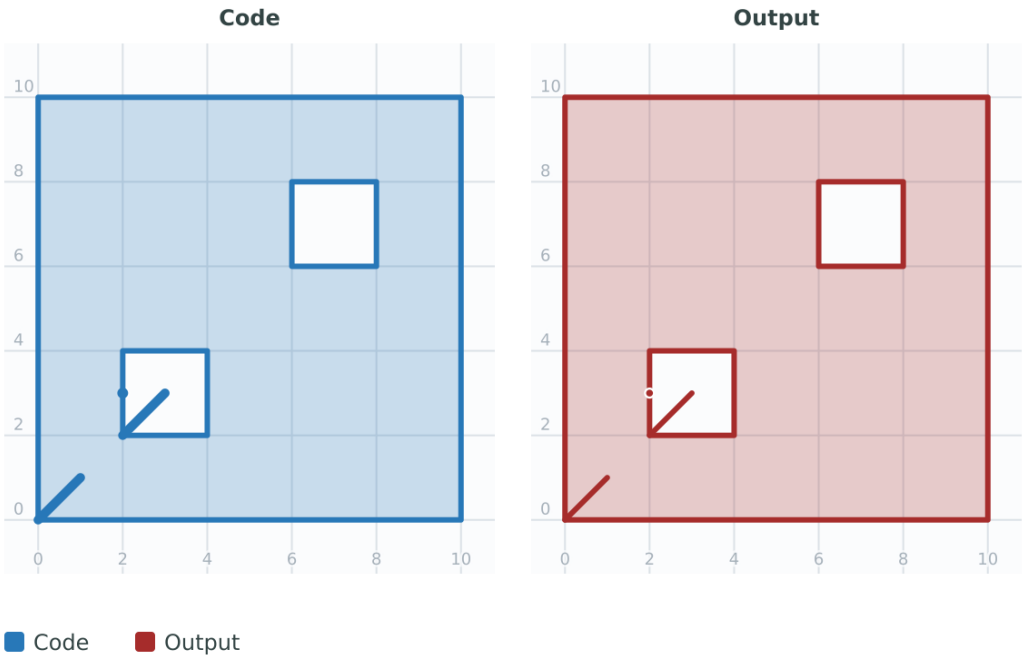
```
SELECT ST_Normalize('GEOMETRYCOLLECTION(
    POINT(2 3),
    MULTILINESTRING((0 0,1 1),(2 2,3 3)),
    POLYGON(
      (0 10,0 0,10 0,10 10,0 10),
      (4 2,2 2,2 4,4 4,4 2),
```

```
        (6 8,8 8,8 6,6 6,6 8)
    )
  )');
```

Output

```
GEOMETRYCOLLECTION(POLYGON((0 0,0 10,10 10,10 0,0 0),(6 6,8 6,8 8,6 8,6 6),(2 2,4 2,4 4,2 4,2 2)), MULTILINESTRING((2 ↩
    2,3 3),(0 0,1 1)), POINT(2 3))
```

Figure



See Also

ST_Equals,

7.5.22 ST_Project

ST_Project — Returns a point projected from a start point by a distance and bearing (azimuth).

Synopsis

geometry **ST_Project**(geometry g1, float distance, float azimuth);
geometry **ST_Project**(geometry g1, geometry g2, float distance);
geography **ST_Project**(geography g1, float distance, float azimuth);
geography **ST_Project**(geography g1, geography g2, float distance);

Description

Returns a point projected from a point along a geodesic using a given distance and azimuth (bearing). This is known as the direct geodesic problem.

The two-point version uses the path from the first to the second point to implicitly define the azimuth and uses the distance as before.

The distance is given in meters. Negative values are supported.

The azimuth (also known as heading or bearing) is given in radians. It is measured clockwise from true north.

- North is azimuth zero (0 degrees)
- East is azimuth $\pi/2$ (90 degrees)
- South is azimuth π (180 degrees)
- West is azimuth $3\pi/2$ (270 degrees)

Negative azimuth values and values greater than 2π (360 degrees) are supported.

Availability: 2.0.0

Enhanced: 2.4.0 Allow negative distance and non-normalized azimuth.

Enhanced: 3.4.0 Allow geometry arguments and two-point form omitting azimuth.

Example: Projected point at 100,000 meters and bearing 45 degrees

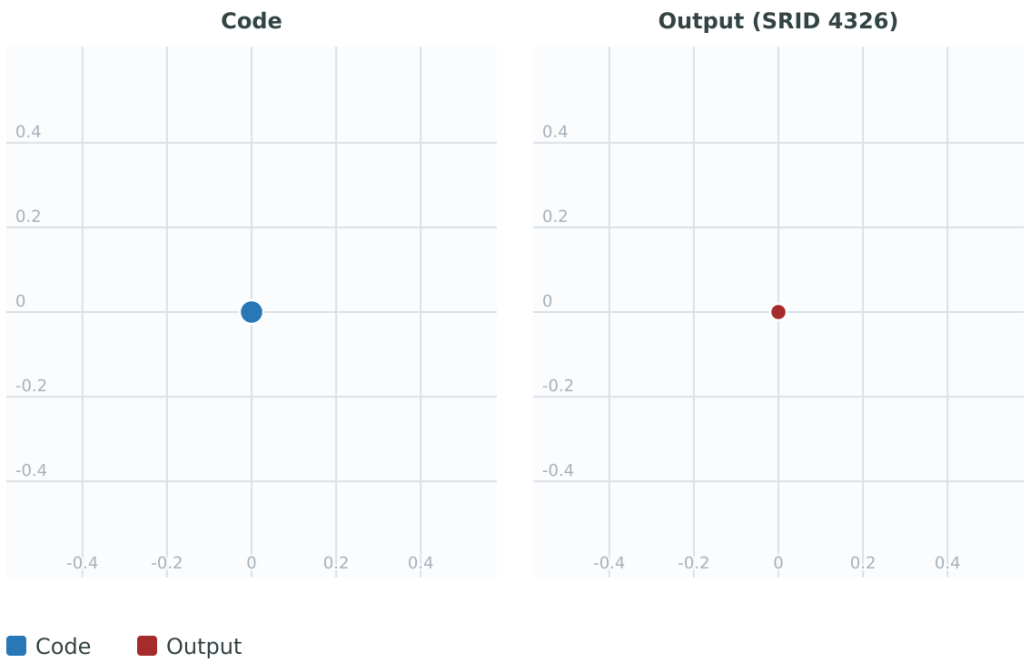
Code

```
SELECT ST_Project('POINT(0 0)'::geography, 100000, radians(45.0));
```

Output

```
POINT(0.635231029125537 0.639472334729198)
```

Figure



See Also

ST_Azimuth, ST_Distance, PostgreSQL function radians()

7.5.23 ST_QuantizeCoordinates

ST_QuantizeCoordinates — Sets least significant bits of coordinates to zero

Synopsis

geometry **ST_QuantizeCoordinates**(geometry g, int prec_x, int prec_y = NULL, int prec_z = NULL, int prec_m = NULL);

Description

`ST_QuantizeCoordinates` determines the number of bits (`N`) required to represent a coordinate value with a specified number of digits after the decimal point, and then sets all but the `N` most significant bits to zero. The resulting coordinate value will still round to the original value, but will have improved compressiblity. This can result in a significant disk usage reduction provided that the geometry column is using a compressible storage type. The function allows specification of a different number of digits after the decimal point in each dimension; unspecified dimensions are assumed to have the precision of the `x` dimension. Negative digits are interpreted to refer digits to the left of the decimal point, (i.e., `prec_x=-2` will preserve coordinate values to the nearest 100.

The coordinates produced by `ST_QuantizeCoordinates` are independent of the geometry that contains those coordinates and the relative position of those coordinates within the geometry. As a result, existing topological relationships between geometries are unaffected by use of this function. The function may produce invalid geometry when it is called with a number of digits lower than the intrinsic precision of the geometry.

Availability: 2.5.0

Technical Background

PostGIS stores all coordinate values as double-precision floating point integers, which can reliably represent 15 significant digits. However, PostGIS may be used to manage data that intrinsically has fewer than 15 significant digits. An example is TIGER data, which is provided as geographic coordinates with six digits of precision after the decimal point (thus requiring only nine significant digits of longitude and eight significant digits of latitude.)

When 15 significant digits are available, there are many possible representations of a number with 9 significant digits. A double precision floating point number uses 52 explicit bits to represent the significand (mantissa) of the coordinate. Only 30 bits are needed to represent a mantissa with 9 significant digits, leaving 22 insignificant bits; we can set their value to anything we like and still end up with a number that rounds to our input value. For example, the value 100.123456 can be represented by the floating point numbers closest to 100.123456000000, 100.123456000001, and 100.123456432199. All are equally valid, in that `ST_AsText(geom, 6)` will return the same result with any of these inputs. As we can set these bits to any value, `ST_QuantizeCoordinates` sets the 22 insignificant bits to zero. For a long coordinate sequence this creates a pattern of blocks of consecutive zeros that is compressed by PostgreSQL more efficiently.



Note
Only the on-disk size of the geometry is potentially affected by `ST_QuantizeCoordinates`. ST_MemSize, which reports the in-memory usage of the geometry, will return the the same value regardless of the disk space used by a geometry.

Examples

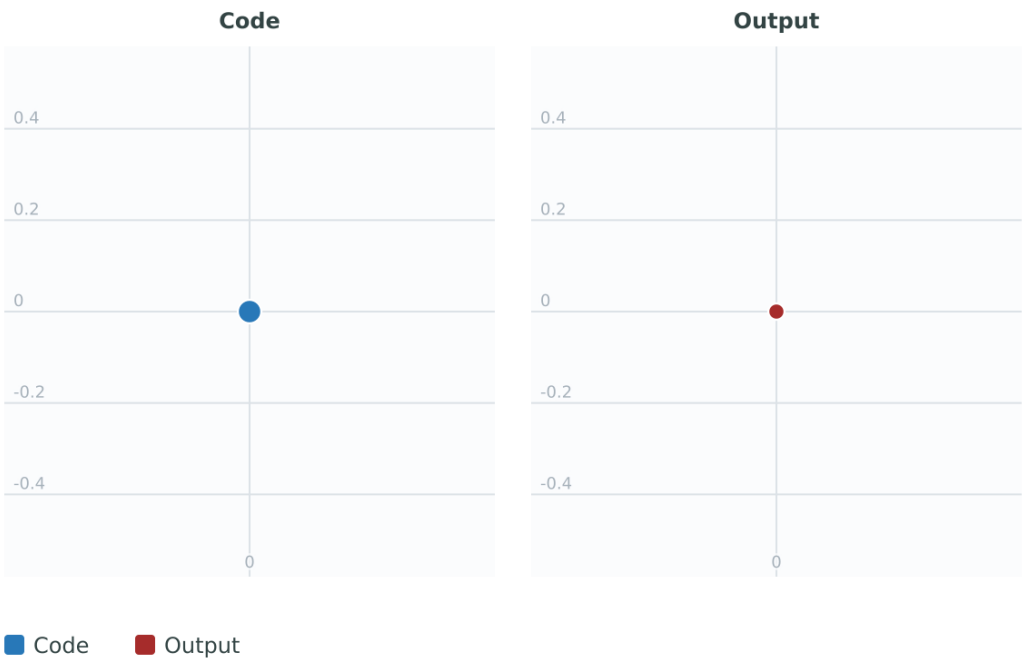
Code

```
SELECT ST_QuantizeCoordinates('POINT (100.123456 0)'::geometry, 4);
```

Output

```
POINT(100.12344360351562 0)
```

Figure



Code

```
WITH test AS (SELECT 'POINT (123.456789123456 123.456789123456)'::geometry AS geom)
SELECT
  digits,
  encode(ST_QuantizeCoordinates(geom, digits), 'hex'),
  ST_QuantizeCoordinates(geom, digits)
FROM test, generate_series(15, -15, -1) AS digits;
```

Output

```
 digits |                   encode                   |                 st_astext
--------+--------------------------------------------+-------------------------------------------
     15 | 01010000005f9a72083cdd5e405f9a72083cdd5e40 | POINT(123.456789123456 123.456789123456)
     14 | 01010000005f9a72083cdd5e405f9a72083cdd5e40 | POINT(123.456789123456 123.456789123456)
     13 | 01010000005e9a72083cdd5e405e9a72083cdd5e40 | POINT(123.45678912345599 123.45678912345599)
     12 | 0101000000409a72083cdd5e40409a72083cdd5e40 | POINT(123.45678912345556 123.45678912345556)
     11 | 0101000000009a72083cdd5e40009a72083cdd5e40 | POINT(123.45678912345466 123.45678912345466)
     10 | 0101000000009872083cdd5e40009872083cdd5e40 | POINT(123.45678912344738 123.45678912344738)
      9 | 0101000000008072083cdd5e40008072083cdd5e40 | POINT(123.45678912336007 123.45678912336007)
      8 | 0101000000000070083cdd5e40000070083cdd5e40 | POINT(123.45678912103176 123.45678912103176)
      7 | 0101000000000060083cdd5e40000060083cdd5e40 | POINT(123.4567891061306 123.4567891061306)
      6 | 0101000000000000083cdd5e40000000083cdd5e40 | POINT(123.45678901672363 123.45678901672363)
      5 | 0101000000000000003cdd5e40000000003cdd5e40 | POINT(123.456787109375 123.456787109375)
      4 | 0101000000000000003cdd5e40000000003cdd5e40 | POINT(123.456787109375 123.456787109375)
      3 | 0101000000000000038dd5e400000000038dd5e40 | POINT(123.45654296875 123.45654296875)
      2 | 0101000000000000000dd5e400000000000dd5e40 | POINT(123.453125 123.453125)
      1 | 0101000000000000000dc5e400000000000dc5e40 | POINT(123.4375 123.4375)
      0 | 0101000000000000000c05e400000000000c05e40 | POINT(123 123)
     -1 | 0101000000000000000005e40000000000000005e40 | POINT(120 120)
     -2 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -3 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -4 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -5 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -6 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -7 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -8 | 0101000000000000000005840000000000000005840 | POINT(96 96)
     -9 | 0101000000000000000005840000000000000005840 | POINT(96 96)
    -10 | 0101000000000000000005840000000000000005840 | POINT(96 96)
    -11 | 0101000000000000000005840000000000000005840 | POINT(96 96)
```

```
    -12 | 0101000000000000000000584000000000000005840 | POINT(96 96)
    -13 | 0101000000000000000000584000000000000005840 | POINT(96 96)
    -14 | 0101000000000000000000584000000000000005840 | POINT(96 96)
    -15 | 0101000000000000000000584000000000000005840 | POINT(96 96)
(31 rows)
```

See Also

ST_SnapToGrid

7.5.24 ST_RemovePoint

ST_RemovePoint — Remove a point from a linestring.

Synopsis

geometry **ST_RemovePoint**(geometry linestring, integer offset);

Description

Removes a point from a LineString, given its index (0-based). Useful for turning a closed line (ring) into an open linestring.

Enhanced: 3.2.0

Availability: 1.1.0

 This function supports 3d and will not drop the z-index.

Examples

Guarantees no lines are closed by removing the end point of closed lines (rings). Assumes geom is of type LINESTRING

Code

```
UPDATE sometable
    SET geom = ST_RemovePoint(geom, ST_NPoints(geom) - 1)
    FROM sometable
    WHERE ST_IsClosed(geom);
```

See Also

ST_AddPoint, ST_NPoints, ST_NumPoints

7.5.25 ST_RemoveRepeatedPoints

ST_RemoveRepeatedPoints — Returns a version of a geometry with duplicate points removed.

Synopsis

geometry **ST_RemoveRepeatedPoints**(geometry geom, float8 tolerance = 0.0);

Description

Returns a version of the given geometry with duplicate consecutive points removed. The function processes only (Multi)LineStrings, (Multi)Polygons and MultiPoints but it can be called with any kind of geometry. Elements of GeometryCollections are processed individually. The endpoints of LineStrings are preserved.

If a non-zero *tolerance* parameter is provided, vertices within the tolerance distance of one another are considered to be duplicates. The distance is computed in 2D (XY plane).

Enhanced: 3.2.0

Availability: 2.2.0

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

Examples

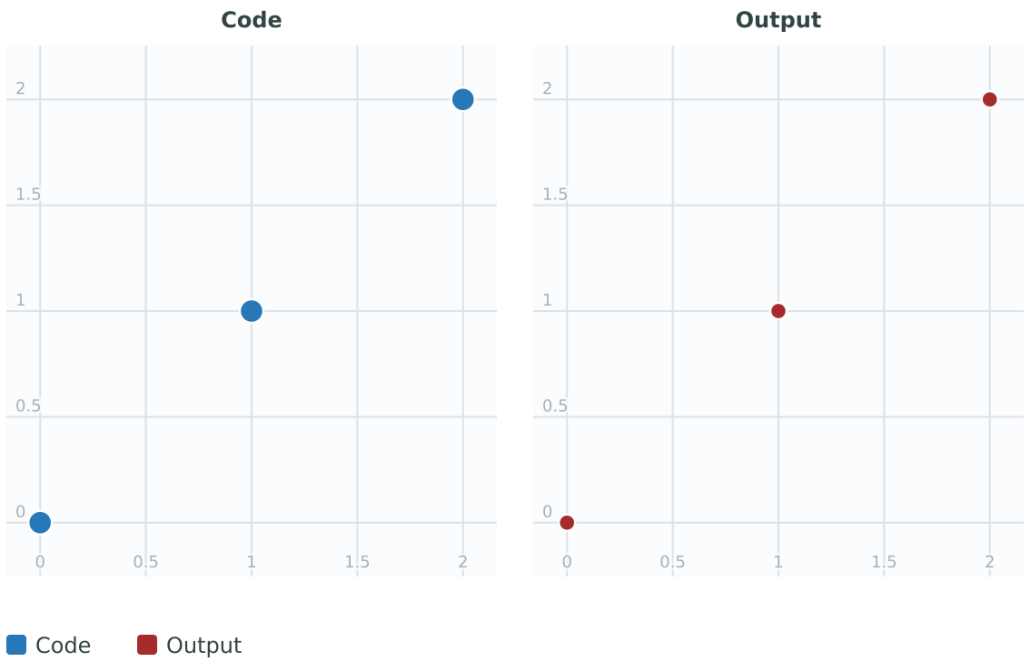
Code

```
SELECT ST_RemoveRepeatedPoints('MULTIPOINT ((1 1),(2 2),(3 3),(2 2))');
```

Output

```
MULTIPOINT((1 1),(2 2),(3 3))
```

Figure



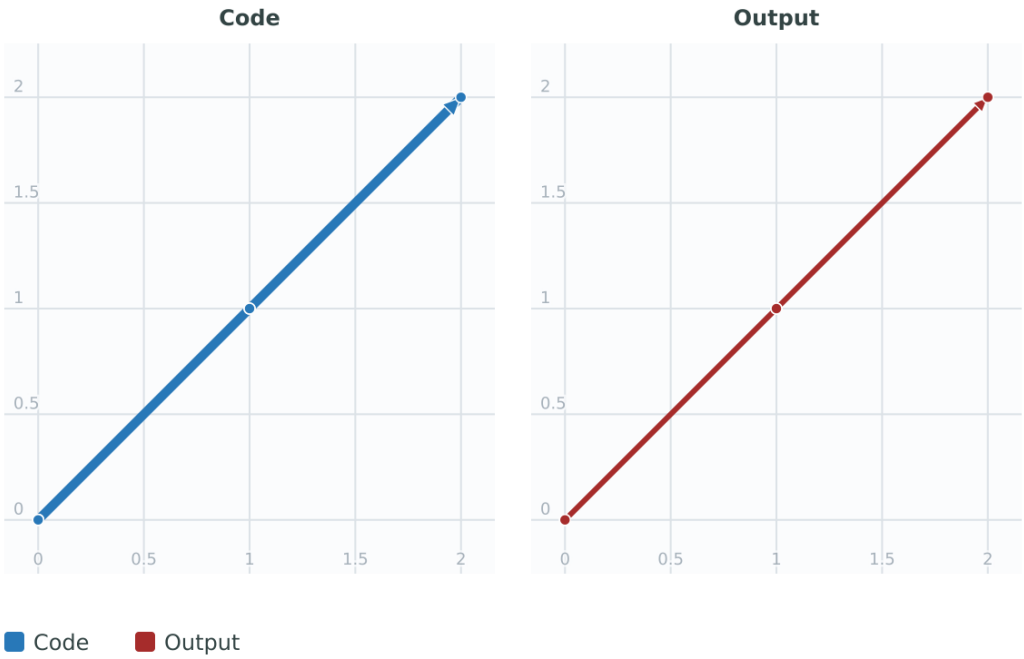
Code

```
SELECT ST_RemoveRepeatedPoints('LINESTRING (0 0,0 0,1 1,0 0,1 1,2 2)');
```

Output

```
LINESTRING(0 0,1 1,0 0,1 1,2 2)
```

Figure



Example: Collection elements are processed individually.

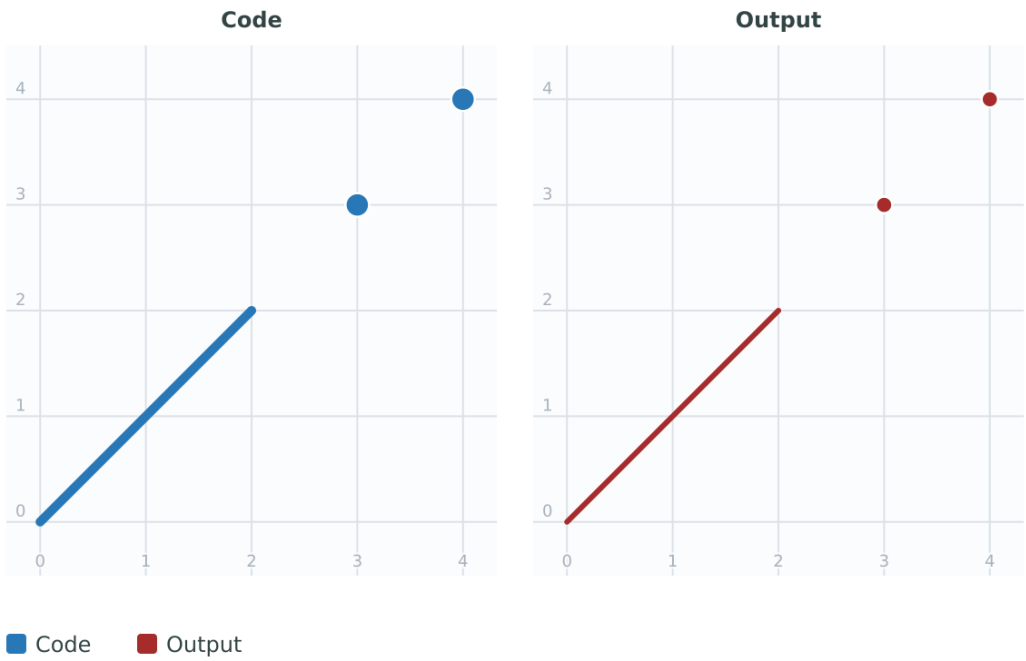
Code

```
SELECT ST_RemoveRepeatedPoints('GEOMETRYCOLLECTION (LINESTRING (1 1,2 2,2 2,3 3),POINT (4 4) ↩
    ,POINT (4 4),POINT (5 5))');
```

Output

```
GEOMETRYCOLLECTION(LINESTRING(1 1,2 2,3 3), POINT(4 4), POINT(4 4), POINT(5 5))
```

Figure



Example: Repeated point removal with a distance tolerance.

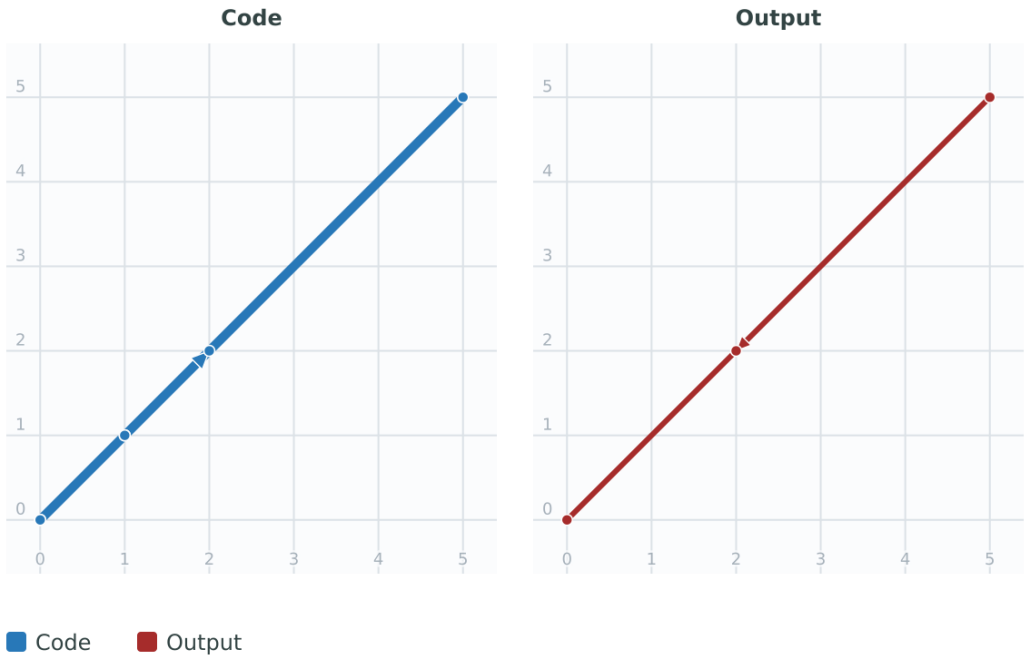
Code

```
SELECT ST_RemoveRepeatedPoints('LINESTRING (0 0,0 0,1 1,5 5,1 1,2 2)', 2);
```

Output

```
LINESTRING(0 0,5 5,2 2)
```

Figure



See Also

ST_Simplify

7.5.26 ST_RemoveIrrelevantPointsForView

ST_RemoveIrrelevantPointsForView — Removes points that are irrelevant for rendering a specific rectangular view of a geometry.

Synopsis

geometry **ST_RemoveIrrelevantPointsForView**(geometry geom, box2d bounds, boolean cartesian_hint = false);

Description

Returns a geometry without points being irrelevant for rendering the geometry within a given rectangular view.

This function can be used to quickly preprocess geometries that should be rendered only within certain bounds.

Only geometries of type (MULTI)POLYGON and (MULTI)LINESTRING are evaluated. Other geometries keep unchanged.

In contrast to `ST_ClipByBox2D()` this function

- sorts out points without computing new intersection points which avoids rounding errors and usually increases performance,
- returns a geometry with equal or similar point number,
- leads to the same rendering result within the specified view, and
- may introduce self-intersections which would make the resulting geometry invalid (see example below).

If `cartesian_hint` is set to `true`, the algorithm applies additional optimizations involving cartesian math to further reduce the resulting point number. Please note that using this option might introduce rendering artifacts if the resulting coordinates are projected into another (non-cartesian) coordinate system before rendering.



Warning
For polygons, this function does currently not ensure that the result is valid. This situation can be checked with ST_IsValid and repaired with ST_MakeValid.

Availability: 3.5.0

Examples

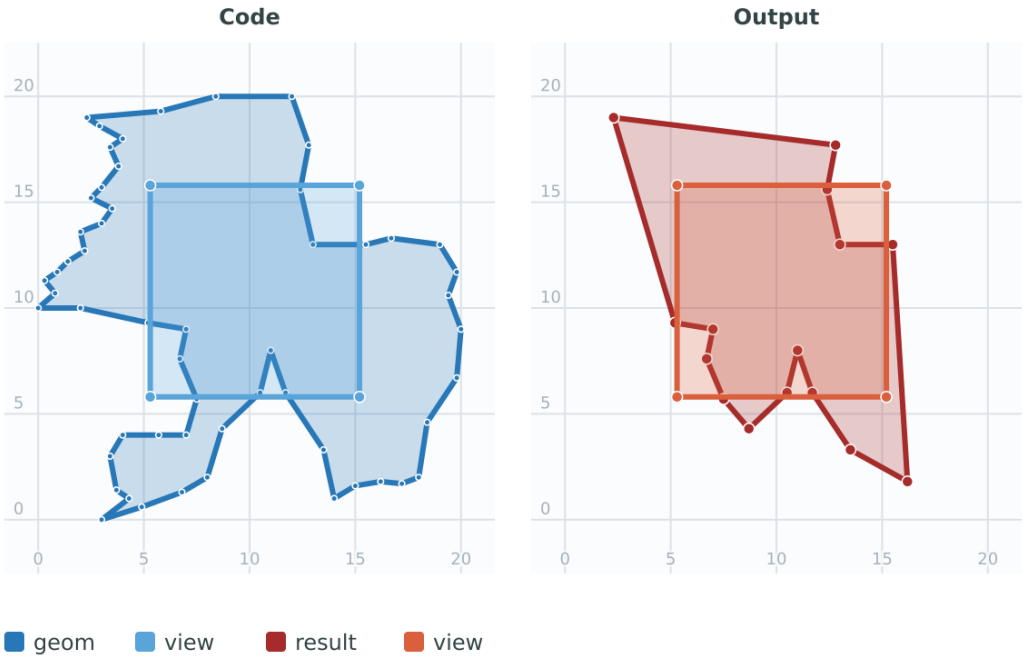
The input polygon has 55 points. The result retains 15 points, while rendering the same shape inside the view envelope.

Code

```
WITH data(geom, view) AS (VALUES (
  'POLYGON((3.2 14.2,6.8 14.2,7.6 11.9,7.2 9.8,7.8 7.2,10.3 7.2,11.5 7.5,13.8 7.2,14.6 ↩
   5.9,14.2 4.8,14.8 3.2,14.6 0.9,13.2 -1.2,12.8 -3.8,12 -4.1,11 -4,9.8 -4.2,8.8 -4.8,8.3 ↩
   -2.5,6.5 0.2,5.8 2.2,5.3 0.2,3.5 -1.5,2.8 -3.8,1.6 -4.5,-0.3 -5.2,-2.2 -5.8,-0.9 ↩
   -4.8,-1.5 -4.4,-1.8 -2.8,-1.2 -1.8,0.5 -1.8,1.8 -1.8,2.3 -0.1,1.5 1.8,1.8 3.2,0 3.5,-3.2 ↩
    4.2,-5.2 4.2,-4.4 4.9,-4.9 5.5,-4.3 5.9,-3.8 6.4,-3 6.9,-3.2 7.8,-2.2 8.2,-1.7 8.9,-2.7 ↩
    9.4,-2.2 9.9,-1.4 10.9,-1.8 11.8,-1.2 12.2,-2.3 12.8,-2.9 13.2,0.6 13.5,3.2 14.2))':: ↩
   geometry,
```

```
  ST_MakeEnvelope(0.1, 0, 10, 10)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view, true) AS result,
       view AS view
FROM data;
```

Figure

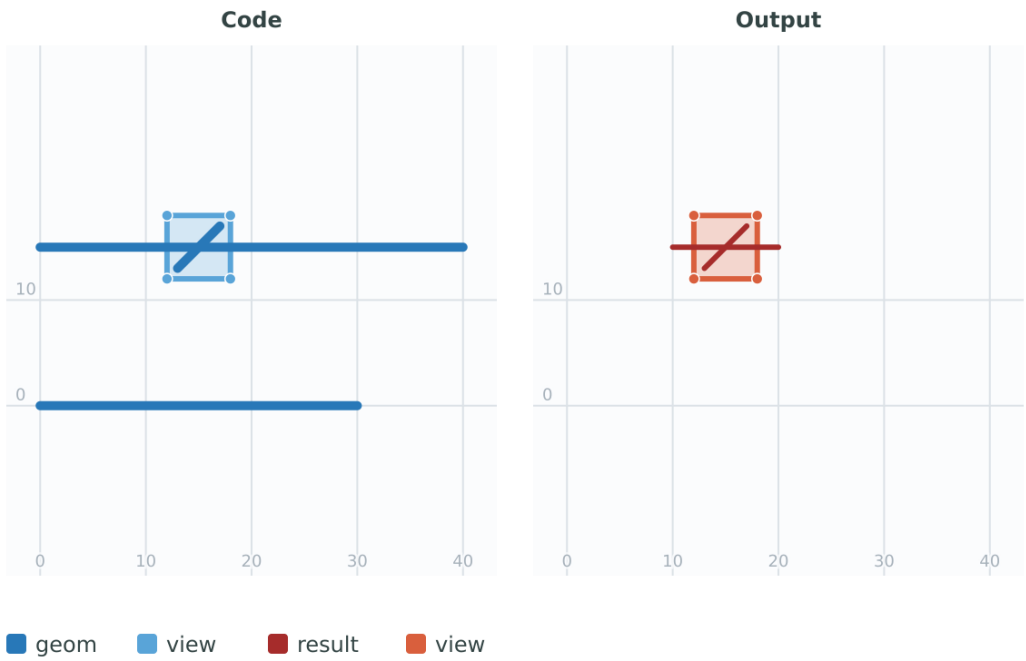


Only linework affecting the view is retained; existing vertices are reused.

Code

```
WITH data(geom, view) AS (VALUES (
  'MULTILINESTRING((0 0,10 0,20 0,30 0),(0 15,5 15,10 15,15 15,20 15,25 15,30 15,40 15),(13 ↩
    13,15 15,17 17))'::geometry,
  ST_MakeEnvelope(12, 12, 18, 18)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view, true) AS result,
       view AS view
FROM data;
```

Figure



A line with no segment relevant to the view becomes empty.

Code

```
SELECT ST_RemoveIrrelevantPointsForView('LINESTRING(0 0,10 0,20 0,30 0)',
ST_MakeEnvelope(12, 12, 18, 18), true);
```

Output

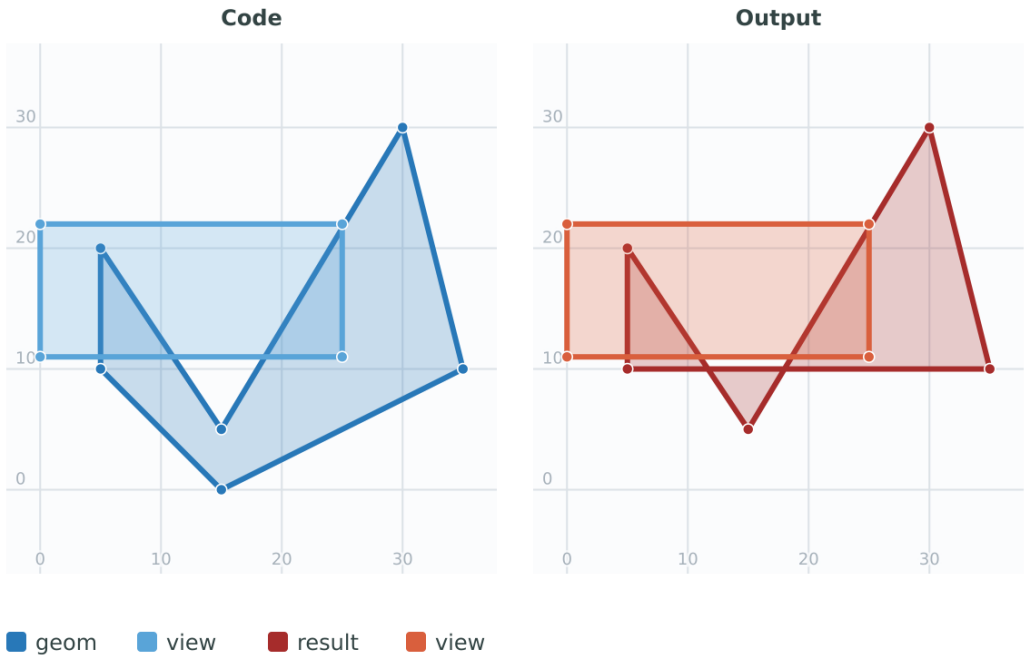
```
LINESTRING EMPTY
```

Removing vertices without computing intersections can introduce self-intersections in the result.

Code

```
WITH data(geom, view) AS (VALUES (
  'POLYGON((0 10,0 0,10 -10,30 0,25 20,10 -5,0 10))'::geometry,
  ST_MakeEnvelope(-5, 1, 20, 12)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view, true) AS result,
       view AS view
FROM data;
```

Figure

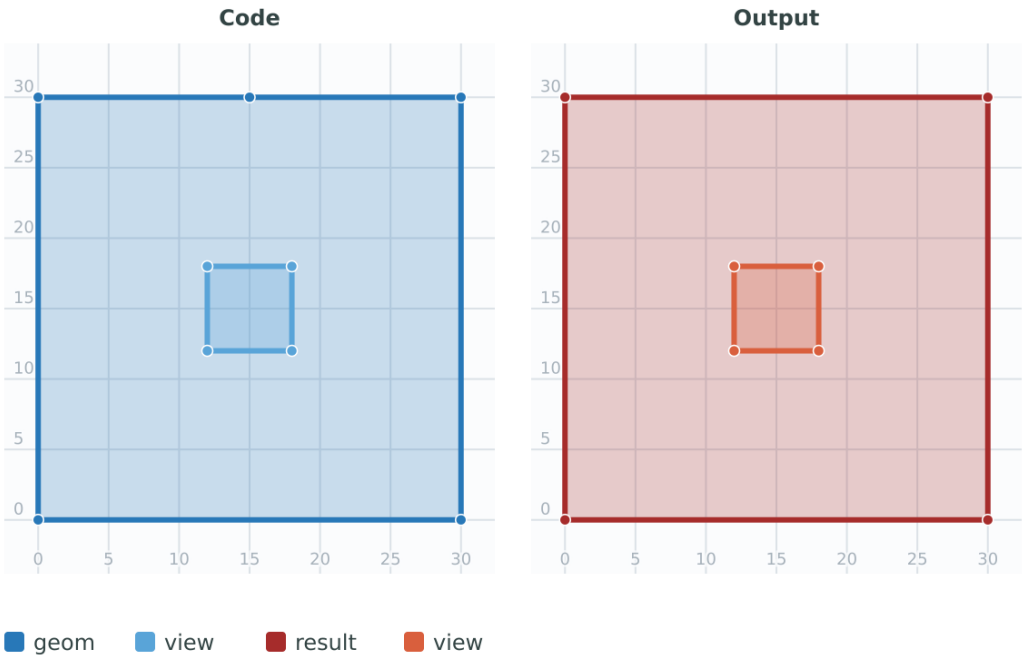


With the default *cartesian_hint*, the function removes only points known to be irrelevant to the view.

Code

```
WITH data(geom, view) AS (VALUES (
  'POLYGON((0 30,15 30,30 30,30 0,0 0,0 30))'::geometry,
  ST_MakeEnvelope(12, 12, 18, 18)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view) AS result,
       view AS view
FROM data;
```

Figure



See Also

ST_ClipByBox2D, ST_Intersection

7.5.27 ST_RemoveSmallParts

ST_RemoveSmallParts — Removes small parts (polygon rings or linestrings) of a geometry.

Synopsis

geometry **ST_RemoveSmallParts**(geometry geom, double precision minSizeX, double precision minSizeY);

Description

Returns a geometry without small parts (exterior or interior polygon rings, or linestrings).

This function can be used as preprocessing step for creating simplified maps, e. g. to remove small islands or holes.

It evaluates only geometries of type (MULTI)POLYGON and (MULTI)LINESTRING. Other geometries remain unchanged.

If *minSizeX* is greater than 0, parts are sorted out if their width is smaller than *minSizeX*.

If *minSizeY* is greater than 0, parts are sorted out if their height is smaller than *minSizeY*.

Both *minSizeX* and *minSizeY* are measured in coordinate system units of the geometry.

For polygon types, evaluation is done separately for each ring which can lead to one of the following results:

- the original geometry,
- a POLYGON with all rings with less vertices,
- a POLYGON with a reduced number of interior rings (having possibly less vertices),

- a POLYGON EMPTY, or
- a MULTIPOLYGON with a reduced number of polygons (having possibly less interior rings or vertices), or
- a MULTIPOLYGON EMPTY.

For linestring types, evaluation is done for each linestring which can lead to one of the following results:

- the original geometry,
- a LINESTRING with a reduced number of vertices,
- a LINESTRING EMPTY,
- a MULTILINESTRING with a reduced number of linestrings (having possibly less vertices), or
- a MULTILINESTRING EMPTY.

Availability: 3.5.0

Examples

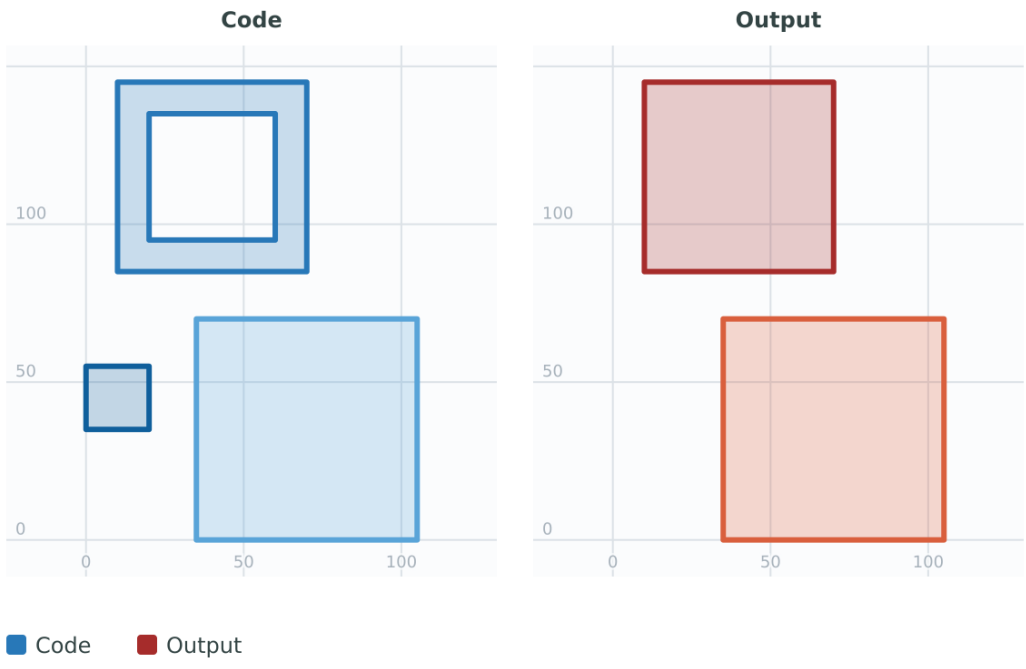
Code

```
SELECT ST_RemoveSmallParts('MULTIPOLYGON(
    ((60 160,120 160,120 220,60 220,60 160),(70 170,70 210,110 210,110 170,70 170)),
    ((85 75,155 75,155 145,85 145,85 75)),
    ((50 110,70 110,70 130,50 130,50 110)))',
    50, 50);
```

Output

```
MULTIPOLYGON(((60 160,120 160,120 220,60 220,60 160)),((85 75,155 75,155 145,85 145,85 75)))
```

Figure



Code

```
SELECT ST_RemoveSmallParts('LINESTRING(10 10,20 20)',
    50, 50);
```

Output

```
LINESTRING EMPTY
```

7.5.28 ST_Reverse

ST_Reverse — Return the geometry with vertex order reversed.

Synopsis

geometry **ST_Reverse**(geometry g1);

Description

Can be used on any geometry and reverses the order of the vertices.

Enhanced: 2.4.0 support for curves was introduced.

 This function supports 3d and will not drop the z-index.

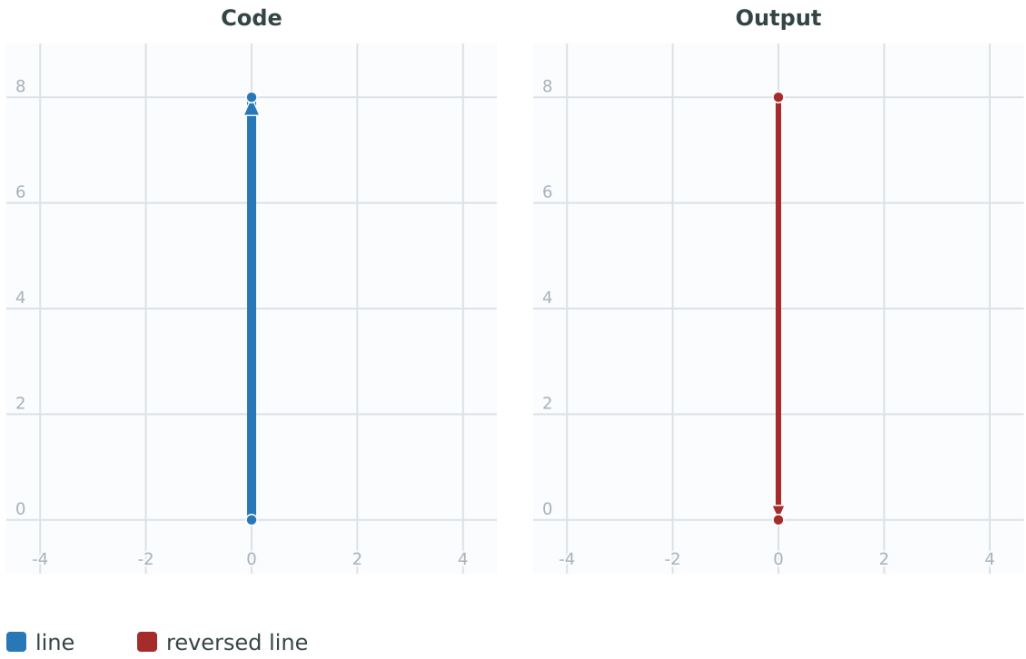
 This function supports Polyhedral surfaces.

Examples

Code

```
WITH data(geom) AS (VALUES (
  ST_MakeLine(ST_Point(1, 2), ST_Point(1, 10))
))
SELECT geom AS input_line,
       ST_Reverse(geom) AS reversed_line
FROM data;
```

Figure



7.5.29 ST_Segmentize

ST_Segmentize — Returns a modified geometry/geography having no segment longer than a given distance.

Synopsis

geometry **ST_Segmentize**(geometry geom, float max_segment_length);
geography **ST_Segmentize**(geography geog, float max_segment_length);

Description

Returns a modified geometry/geography having no segment longer than `max_segment_length`. Length is computed in 2D. Segments are always split into equal-length subsegments.

- For geometry, the maximum length is in the units of the spatial reference system.
- For geography, the maximum length is in meters. Distances are computed on the sphere. Added vertices are created along the spherical great-circle arcs defined by segment endpoints.



Note
This only shortens long segments. It does not lengthen segments shorter than the maximum length.



Warning
For inputs containing long segments, specifying a relatively short `max_segment_length` can cause a very large number of vertices to be added. This can happen unintentionally if the argument is specified accidentally as a number of segments, rather than a maximum length.

Availability: 1.2.2

Enhanced: 3.0.0 Segmentize geometry now produces equal-length subsegments

Enhanced: 2.3.0 Segmentize geography now produces equal-length subsegments

Enhanced: 2.1.0 support for geography was introduced.

Changed: 2.1.0 As a result of the introduction of geography support, the usage `ST_Segmentize('LINESTRING(1 2, 3 4)', 0.5)` causes an ambiguous function error. The input needs to be properly typed as a geometry or geography. Use ST_GeomFromText, ST_GeogFromText or a cast to the required type (e.g. `ST_Segmentize('LINESTRING(1 2, 3 4)'::geometry, 0.5) )`

Examples

Segmentizing a line. Long segments are split evenly, and short segments are not split.

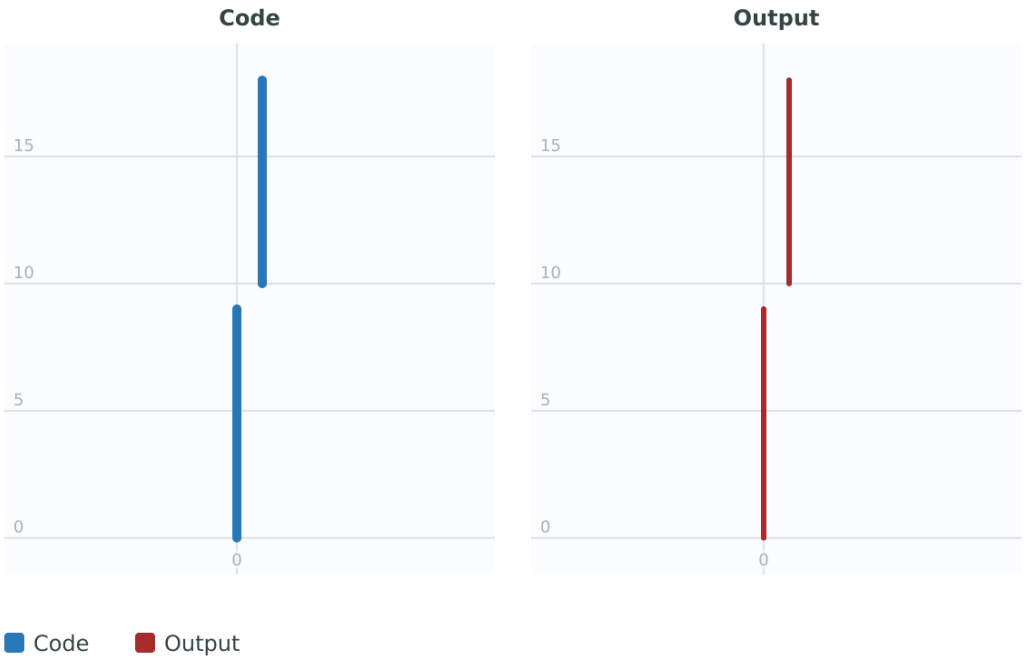
Code

```
SELECT ST_Segmentize('MULTILINESTRING((0 0,0 1,0 9),(1 10,1 18))'::geometry,
5);
```

Output

```
MULTILINESTRING((0 0,0 1,0 5,0 9),(1 10,1 14,1 18))
```

Figure



Segmentizing a polygon:

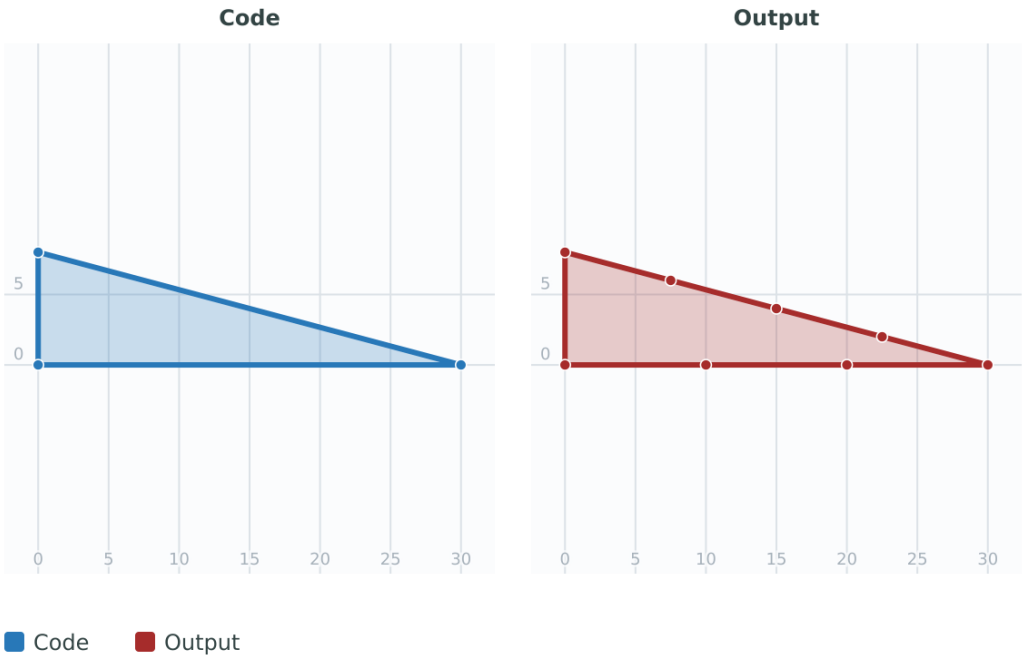
Code

```
SELECT ST_Segmentize('POLYGON((0 0,0 8,30 0,0 0))'::geometry,
        10);
```

Output

```
POLYGON((0 0,0 8,7.5 6,15 4,22.5 2,30 0,20 0,10 0,0 0))
```

Figure



Segmentizing a geographic line, using a maximum segment length of 2000 kilometers. Vertices are added along the great-circle arc connecting the endpoints.

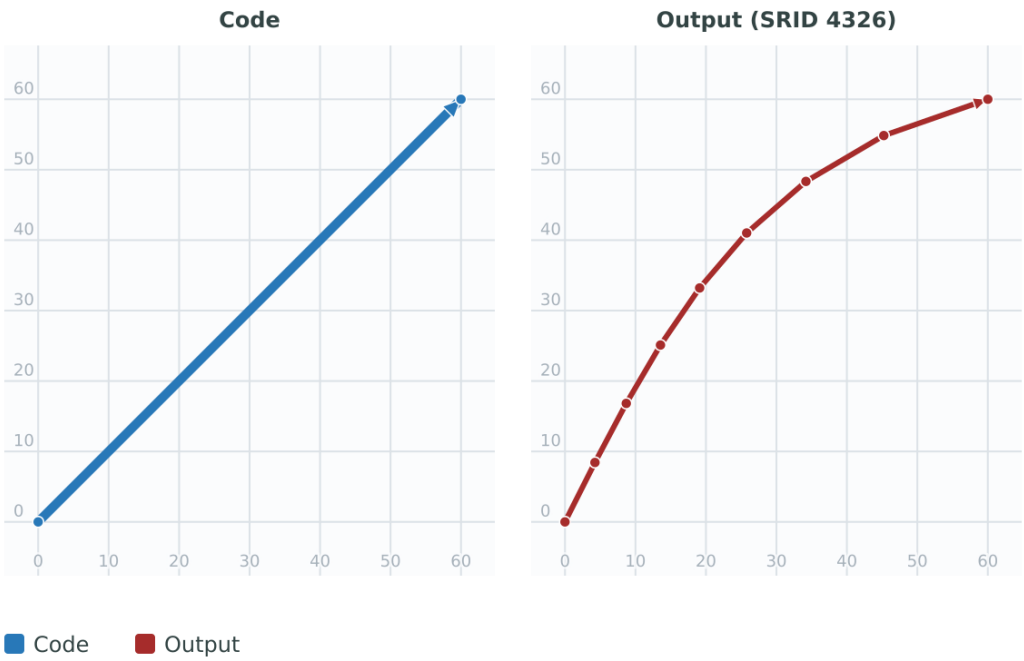
Code

```
SELECT ST_Segmentize('LINESTRING (0 0,60 60)'::geography), 2000000);
```

Output

```
LINESTRING(0 0,4.252632294621186 8.43596525986862,8.69579947419404 16.824093489701564,13.550465473227048 ↩
    25.107950473646188,19.1066053508691 33.21091076089908,25.779290201459894 41.01711439406505,34.188839517966954 ↩
    48.337222885886,45.238153936612264 54.84733442373889,60 60)
```

Figure



See Also

ST_LineSubstring

7.5.30 ST_SetPoint

ST_SetPoint — Replace point of a linestring with a given point.

Synopsis

geometry **ST_SetPoint**(geometry linestring, integer zerobasedposition, geometry point);

Description

Replace point N of linestring with given point. Index is 0-based.Negative index are counted backwards, so that -1 is last point. This is especially useful in triggers when trying to maintain relationship of joints when one vertex moves.

Availability: 1.1.0

Updated 2.3.0 : negative indexing

This function supports 3d and will not drop the z-index.

Examples

Change first point in line string from -1 3 to -1 1.

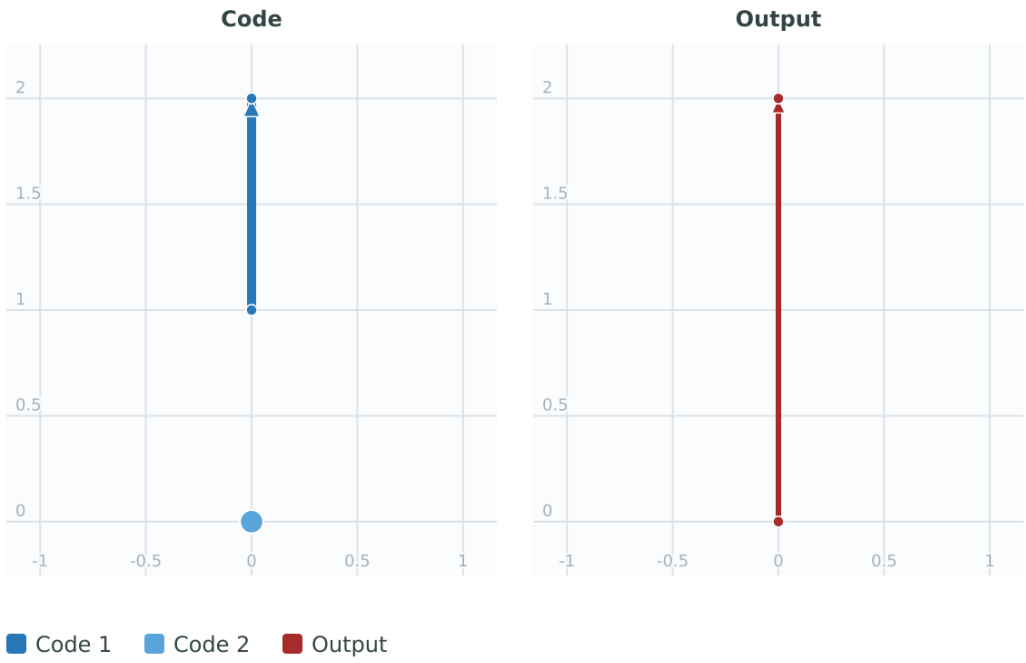
Code

```
SELECT ST_SetPoint('LINESTRING(-1 2,-1 3)', 0, 'POINT(-1 1)');
```

Output

```
LINESTRING(-1 1,-1 3)
```

Figure



Change last point in a line string (let's play with 3D linestring this time)

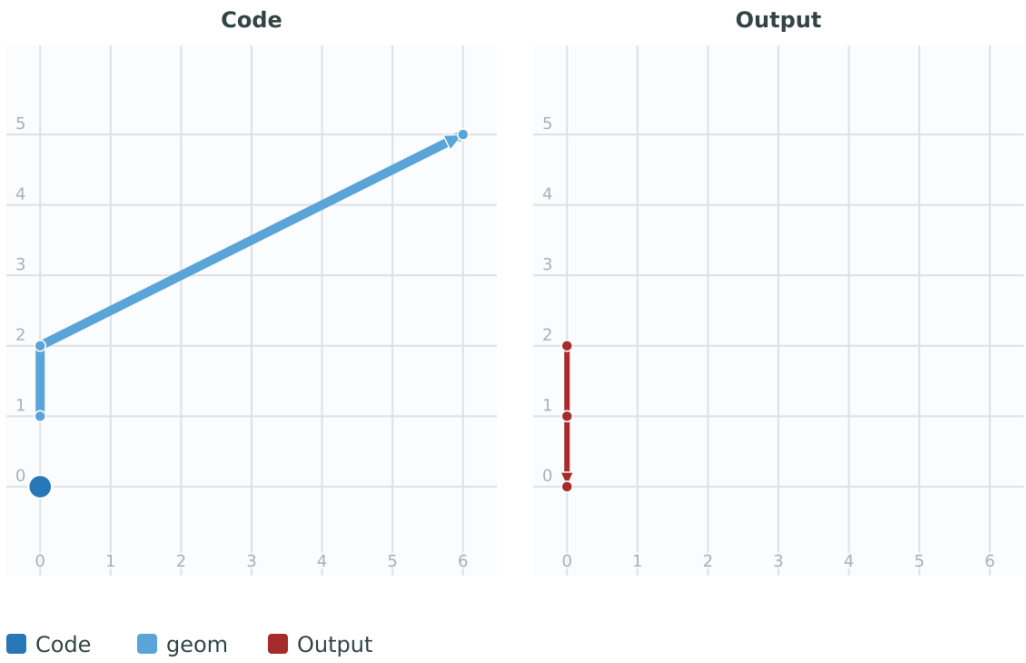
Code

```
SELECT ST_SetPoint(foo.geom, ST_NumPoints(foo.geom) - 1, 'POINT(-1 1 3)')
FROM (SELECT 'LINESTRING(-1 2 3,-1 3 4,5 6 7)'::geometry As geom) As foo;
```

Output

```
LINESTRING(-1 2 3,-1 3 4,-1 1 3)
```

Figure



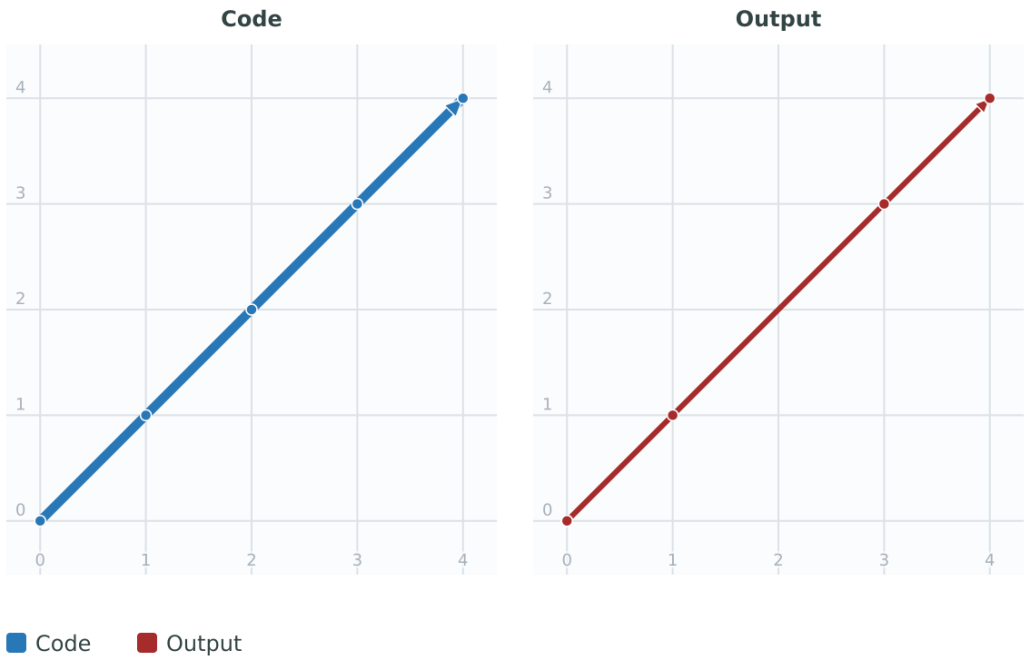
Code

```
SELECT ST_SetPoint(g, -3, p)
FROM ST_GeomFromText('LINESTRING(0 0,1 1,2 2,3 3,4 4)') AS g
    CROSS JOIN ST_PointN(g, 1) as p;
```

Output

```
LINESTRING(0 0,1 1,0 0,3 3,4 4)
```

Figure



See Also

ST_AddPoint, ST_NPoints, ST_NumPoints, ST_PointN, ST_RemovePoint

7.5.31 ST_ShiftLongitude

ST_ShiftLongitude — Shifts the longitude coordinates of a geometry between -180..180 and 0..360.

Synopsis

geometry **ST_ShiftLongitude**(geometry geom);

Description

Reads every point/vertex in a geometry, and shifts its longitude coordinate from -180..0 to 180..360 and vice versa if between these ranges. This function is symmetrical so the result is a 0..360 representation of a -180..180 data and a -180..180 representation of a 0..360 data.



Note
This is only useful for data with coordinates in longitude/latitude; e.g. SRID 4326 (WGS 84 geographic)



Warning
Pre-1.3.4 bug prevented this from working for MULTIPOINT. 1.3.4+ works with MULTIPOINT as well.

 This function supports 3d and will not drop the z-index.

Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced.

NOTE: this function was renamed from "ST_Shift_Longitude" in 2.2.0

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

This example performs a forward longitude shift on a single point.

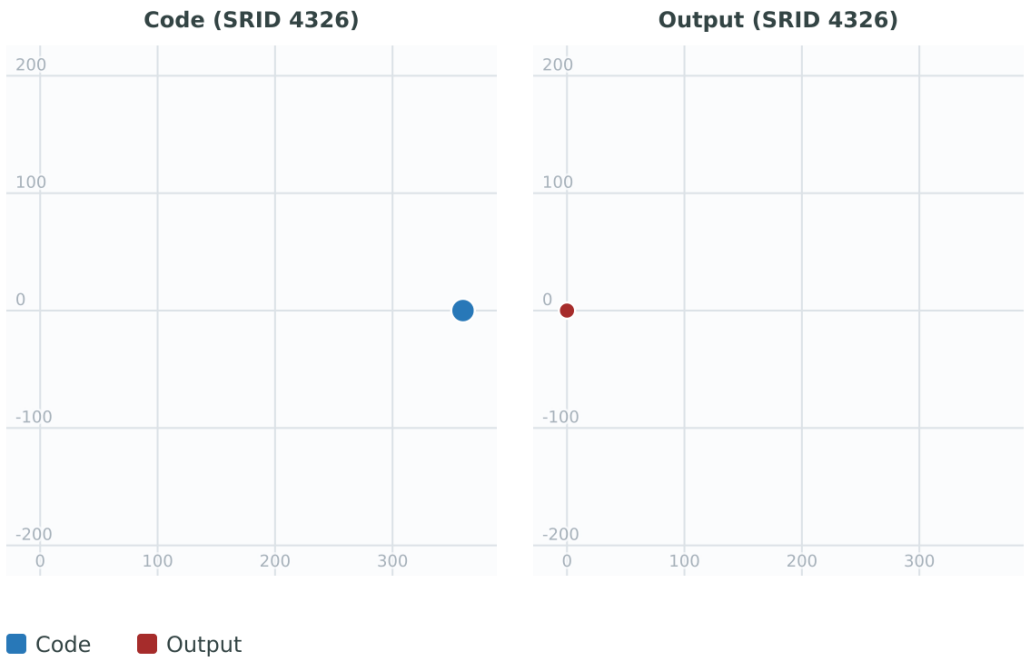
Code

```
SELECT ST_ShiftLongitude('SRID=4326;POINT(270 0)'::geometry)
```

Output

```
POINT(-90 0)
```

Figure



This example performs a reverse longitude shift on a single point.

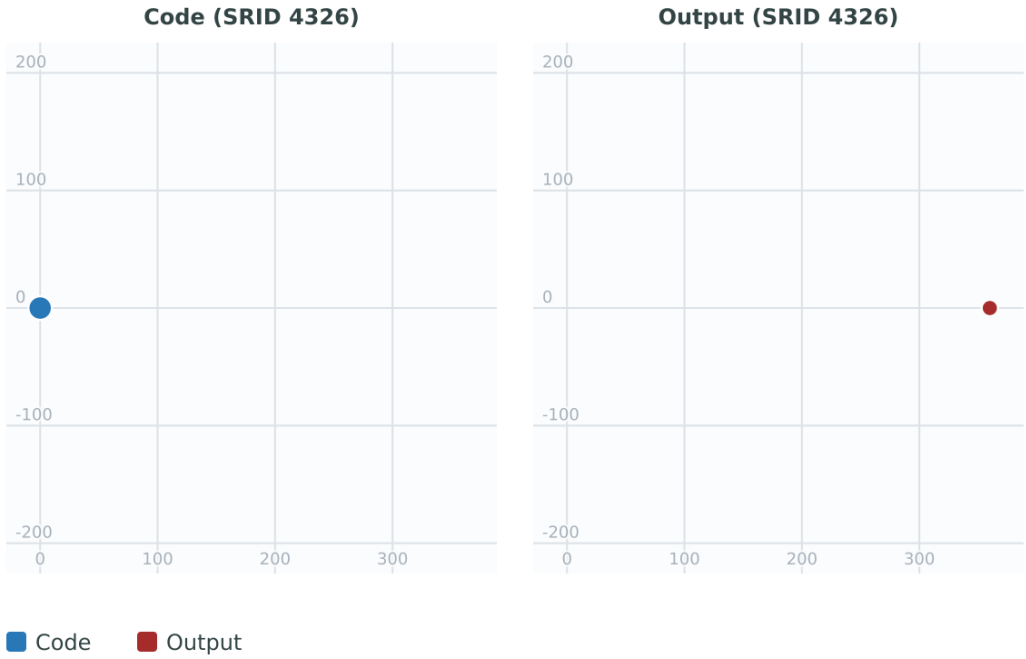
Code

```
SELECT ST_ShiftLongitude('SRID=4326;POINT(-90 0)'::geometry)
```

Output

```
POINT(270 0)
```

Figure



This example shows that LineStrings are shifted only where coordinates require it.

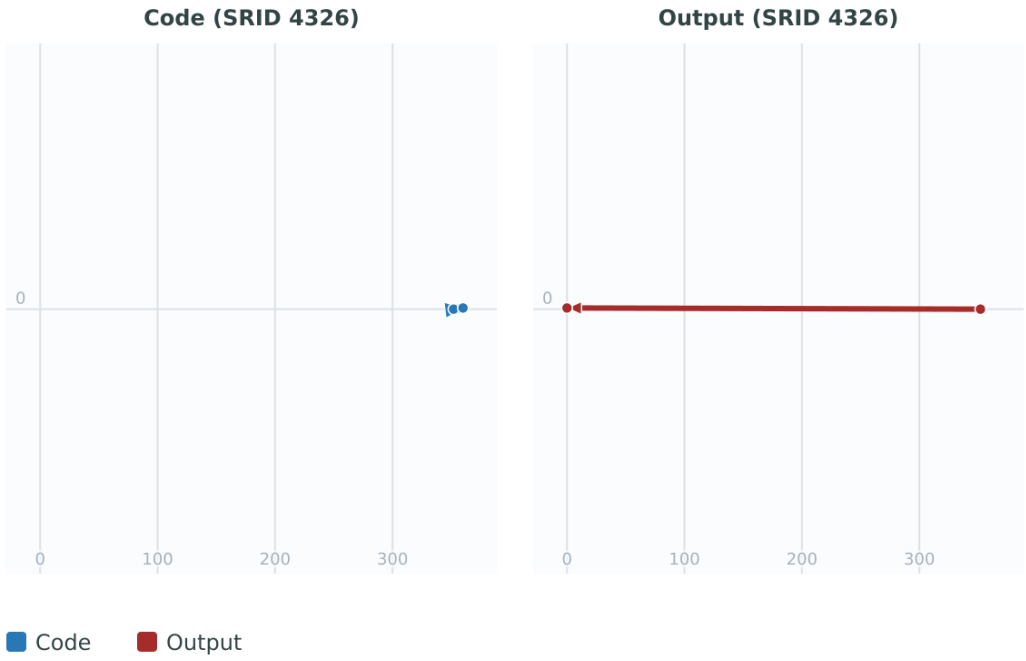
Code

```
SELECT ST_ShiftLongitude('SRID=4326;LINESTRING(174 12,182 13)'::geometry)
```

Output

```
LINESTRING(174 12,-178 13)
```

Figure



See Also

ST_WrapX

7.5.32 ST_WrapX

ST_WrapX — Wrap a geometry around an X value.

Synopsis

geometry **ST_WrapX**(geometry geom, float8 wrap, float8 move);

Description

This function splits the input geometries and then moves every resulting component falling on the right (for negative 'move') or on the left (for positive 'move') of given 'wrap' line in the direction specified by the 'move' parameter, finally re-unioning the pieces together.



Note
This is useful to "recenter" long-lat input to have features of interest not spawned from one side to the other.

Availability: 2.3.0 requires GEOS

 This function supports 3d and will not drop the z-index.

Examples

Move all components whose bounding boxes fall completely to the left of `x = 0` by +360.

Code

```
SELECT ST_WrapX(geom, 0, 360);
```

Move all components whose bounding boxes fall completely to the left of `x = -30` by +360.

Code

```
SELECT ST_WrapX(geom, -30, 360);
```

See Also

ST_ShiftLongitude

7.5.33 ST_SnapToGrid

ST_SnapToGrid — Snap all points of the input geometry to a regular grid.

Synopsis

geometry **ST_SnapToGrid**(geometry geomA, float originX, float originY, float sizeX, float sizeY);
geometry **ST_SnapToGrid**(geometry geomA, float sizeX, float sizeY);
geometry **ST_SnapToGrid**(geometry geomA, float size);
geometry **ST_SnapToGrid**(geometry geomA, geometry pointOrigin, float sizeX, float sizeY, float sizeZ, float sizeM);

Description

Variant 1,2,3: Snap all points of the input geometry to the grid defined by its origin and cell size. Remove consecutive points falling on the same cell, eventually returning NULL if output points are not enough to define a geometry of the given type. Collapsed geometries in a collection are stripped from it. Useful for reducing precision.

Variant 4: Introduced 1.1.0 - Snap all points of the input geometry to the grid defined by its origin (the second argument, must be a point) and cell sizes. Specify 0 as size for any dimension you don't want to snap to a grid.



Note
The returned geometry might lose its simplicity (see ST_IsSimple).



Note
Before release 1.1.0 this function always returned a 2d geometry. Starting at 1.1.0 the returned geometry will have same dimensionality as the input one with higher dimension values untouched. Use the version taking a second geometry argument to define all grid dimensions.

Availability: 1.0.0RC1

Availability: 1.1.0 - Z and M support

 This function supports 3d and will not drop the z-index.

Examples

Snap a geometry column to a precision grid of 10^-3.

Code

```
UPDATE mytable
   SET geom = ST_SnapToGrid(geom, 0.001);
```

Display how the same grid size rounds a LineString.

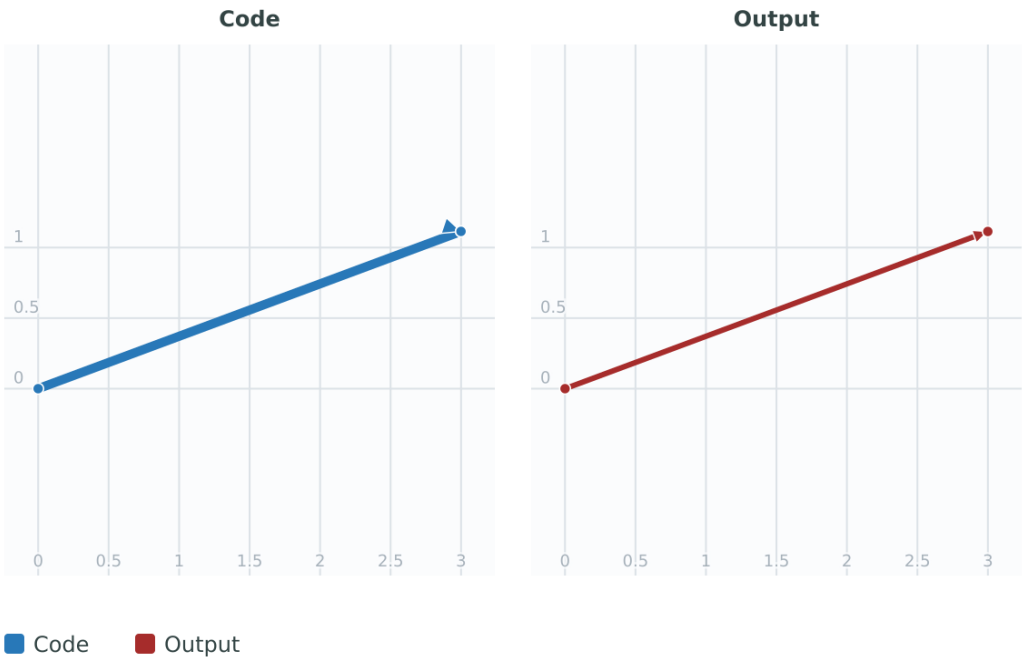
Code

```
SELECT ST_SnapToGrid('LINESTRING(1.1115678 2.123,4.111111 3.2374897,4.11112 3.23748667)':: ↩
    geometry,
        0.001);
```

Output

```
LINESTRING(1.112 2.123,4.111 3.237)
```

Figure



This example snaps a 4D geometry.

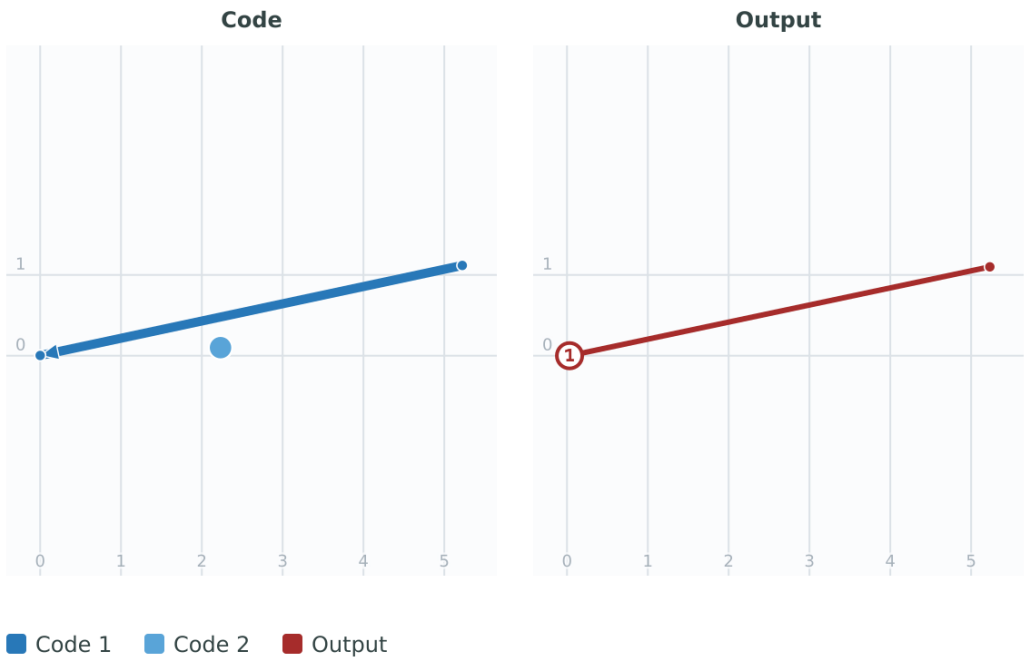
Code

```
SELECT ST_SnapToGrid('LINESTRING(-1.1115678 2.123 2.3456 1.11111,
        4.111111 3.2374897 3.1234 1.1111,-1.11111112 2.123 2.3456 1.1111112)'::geometry,
        'POINT(1.12 2.22 3.2 4.4444)',
        0.1, 0.1, 0.1, 0.01);
```

Output

```
LINESTRING(-1.08 2.12 2.3 1.1144,4.12 3.22 3.1 1.1144,-1.08 2.12 2.3 1.1144)
```

Figure



With a 4D geometry - the ST_SnapToGrid(geom,size) only touches x and y coords but keeps m and z the same.

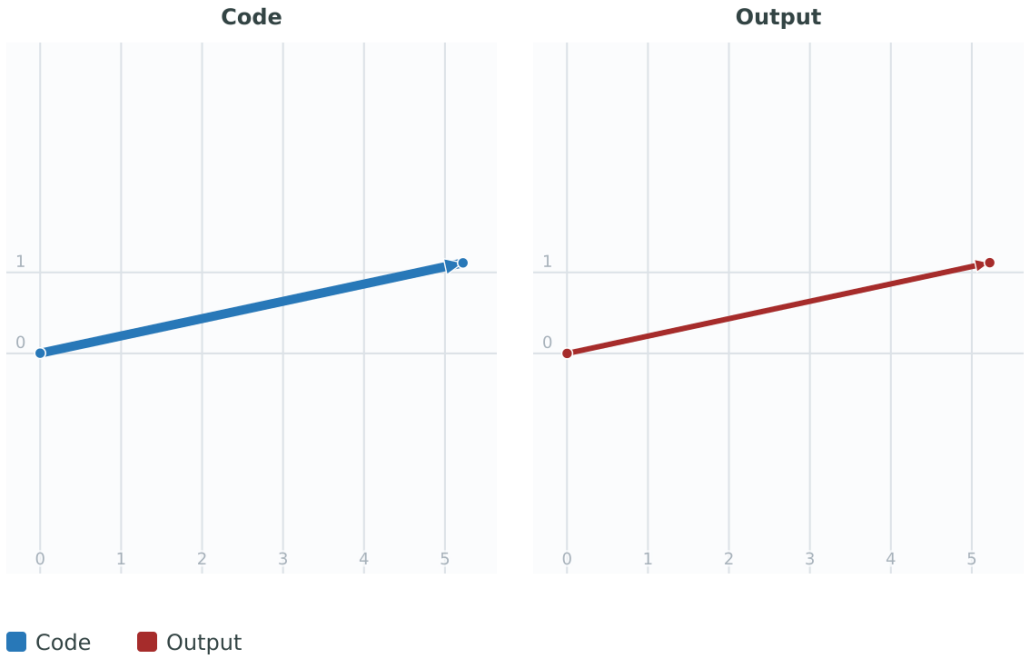
Code

```
SELECT ST_SnapToGrid('LINESTRING(-1.1115678 2.123 3 2.3456,
        4.111111 3.2374897 3.1234 1.1111)'::geometry,
       0.01);
```

Output

```
LINESTRING(-1.11 2.12 3 2.3456,4.11 3.24 3.1234 1.1111)
```

Figure



See Also

ST_Snap, ST_AsEWKT, ST_AsText, ST_GeomFromText, ST_GeomFromEWKT, ST_Simplify

7.5.34 ST_Snap

ST_Snap — Snap segments and vertices of input geometry to vertices of a reference geometry.

Synopsis

geometry **ST_Snap**(geometry input, geometry reference, float tolerance);

Description

Snaps the vertices and segments of a geometry to another Geometry's vertices. A snap distance tolerance is used to control where snapping is performed. The result geometry is the input geometry with the vertices snapped. If no snapping occurs then the input geometry is returned unchanged.

Snapping one geometry to another can improve robustness for overlay operations by eliminating nearly-coincident edges (which cause problems during noding and intersection calculation).

Too much snapping can result in invalid topology being created, so the number and location of snapped vertices is decided using heuristics to determine when it is safe to snap. This can result in some potential snaps being omitted, however.



Note
The returned geometry might lose its simplicity (see ST_IsSimple) and validity (see ST_IsValid).

Performed by the GEOS module.

Availability: 2.0.0

Examples

A MultiPolygon snapped to a LineString at 1.01 times their distance.

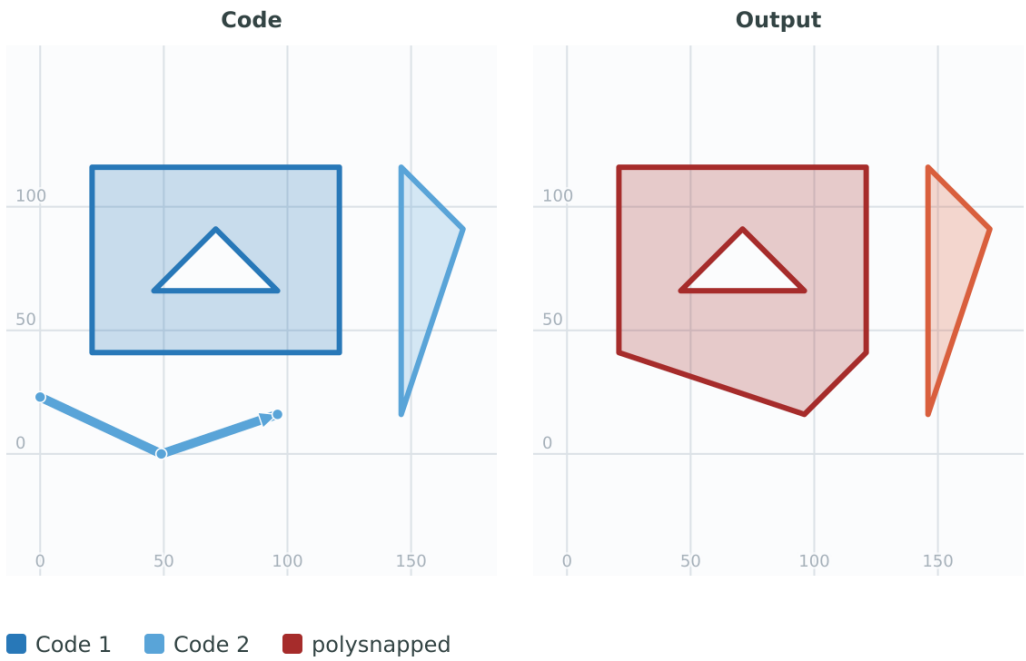
Code

```
SELECT ST_Snap(poly, line, ST_Distance(poly, line)*1.01) AS polysnapped
FROM (SELECT
   ST_GeomFromText('MULTIPOLYGON(
     ((26 125,26 200,126 200,126 125,26 125 ),
      ( 51 150,101 150,76 175,51 150 )),
      (( 151 100,151 200,176 175,151 100 )))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
 MULTIPOLYGON(((26 125,26 200,126 200,126 125,101 100,26 125),
(51 150,101 150,76 175,51 150)),((151 100,151 200,176 175,151 100)))
```

Figure



A MultiPolygon snapped to a LineString at 1.25 times their distance.

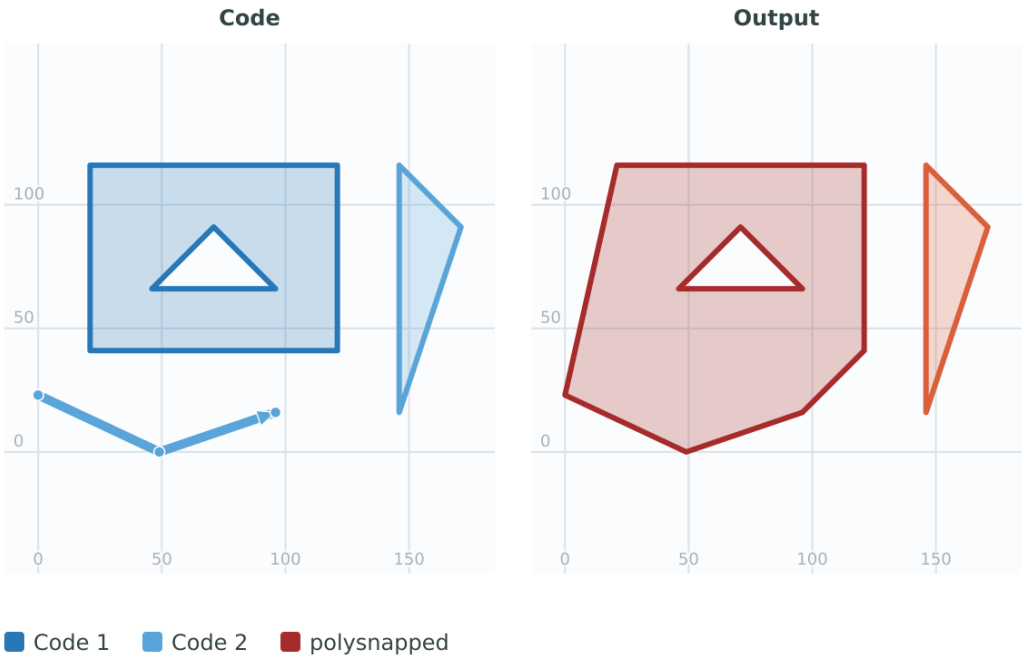
Code

```
SELECT ST_Snap(poly, line, ST_Distance(poly, line)*1.25) AS polysnapped
FROM (SELECT
  ST_GeomFromText('MULTIPOLYGON(
    (( 26 125,26 200,126 200,126 125,26 125 ),
      ( 51 150,101 150,76 175,51 150 )),
      (( 151 100,151 200,176 175,151 100 )))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
MULTIPOLYGON(((5 107,26 200,126 200,126 125,101 100,54 84,5 107),
(51 150,101 150,76 175,51 150)),((151 100,151 200,176 175,151 100)))
```

Figure



The LineString snapped to the original MultiPolygon at 1.01 times their distance.

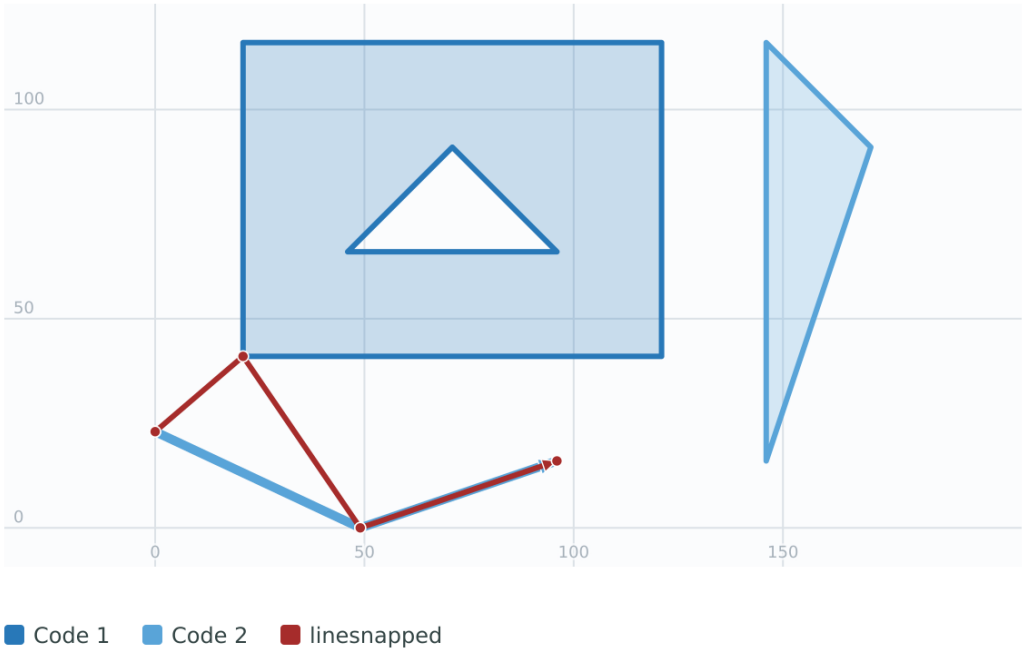
Code

```
SELECT ST_Snap(line, poly, ST_Distance(poly, line)*1.01) AS linesnapped
FROM (SELECT
  ST_GeomFromText('MULTIPOLYGON(
     ((26 125,26 200,126 200,126 125,26 125),
      (51 150,101 150,76 175,51 150 )),
      ((151 100,151 200,176 175,151 100)))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
LINESTRING(5 107,26 125,54 84,101 100)
```

Figure



The LineString snapped to the original MultiPolygon at 1.25 times their distance.

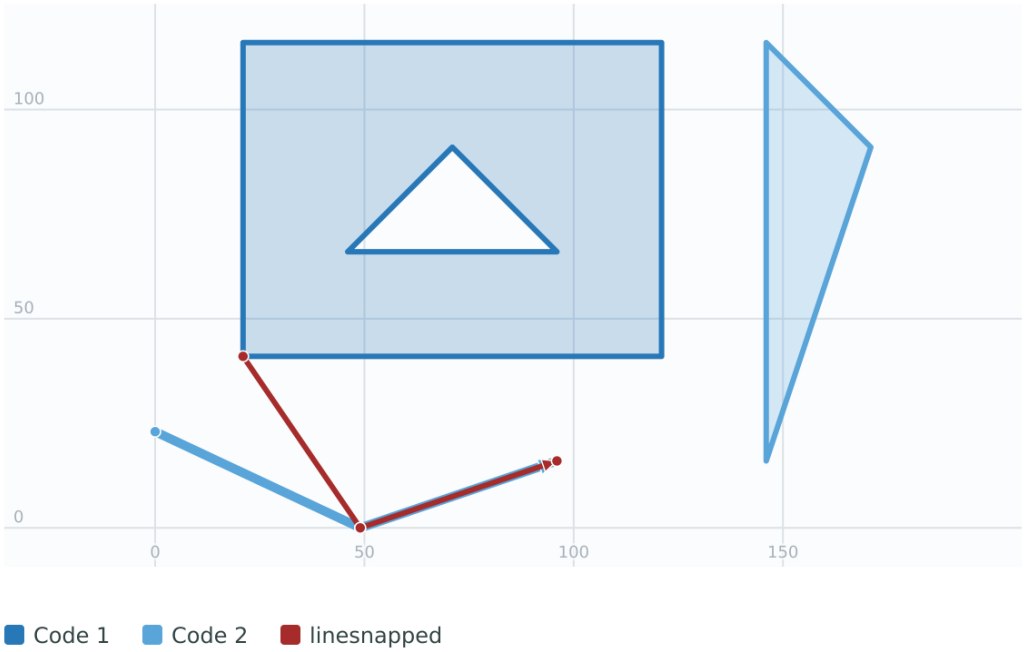
Code

```
SELECT ST_Snap(line, poly, ST_Distance(poly, line)*1.25) AS linesnapped
FROM (SELECT
  ST_GeomFromText('MULTIPOLYGON(
     (( 26 125,26 200,126 200,126 125,26 125 ),
      (51 150,101 150,76 175,51 150 )),
      ((151 100,151 200,176 175,151 100 )))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
LINESTRING(26 125,54 84,101 100)
```

Figure



See Also

ST_SnapToGrid

7.5.35 ST_SwapOrdinates

ST_SwapOrdinates — Returns a version of the given geometry with given ordinate values swapped.

Synopsis

geometry **ST_SwapOrdinates**(geometry geom, cstring ords);

Description

Returns a version of the given geometry with given ordinates swapped.

The `ords` parameter is a 2-characters string naming the ordinates to swap. Valid names are: x,y,z and m.

Availability: 2.2.0

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Example

This example scales the M value by 2.

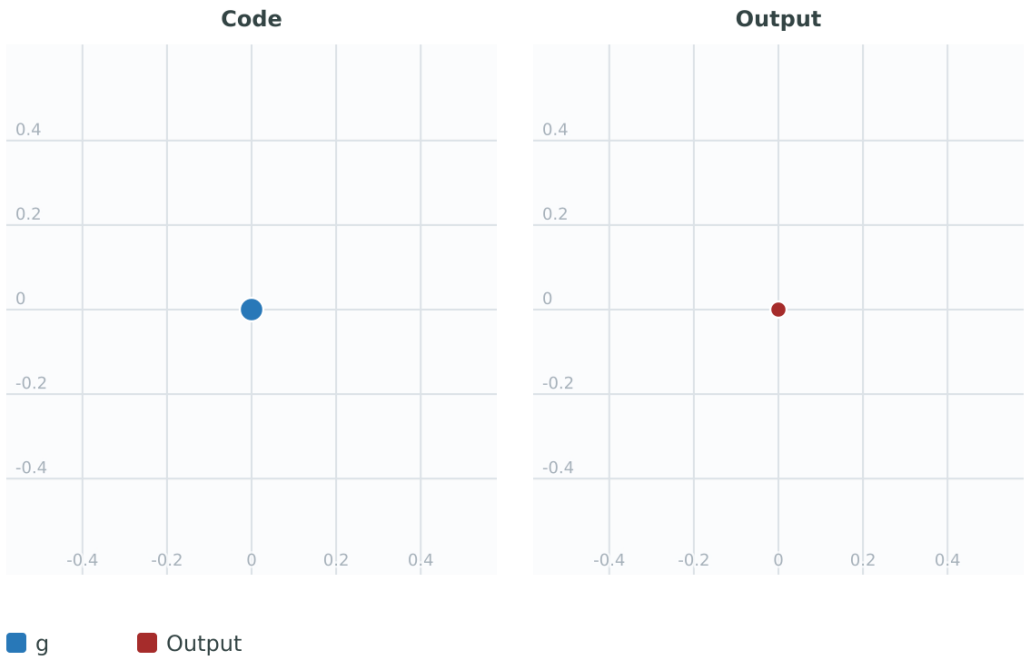
Code

```
SELECT ST_SwapOrdinates(ST_Scale(ST_SwapOrdinates(g, 'xm'),
      2, 1
    ),
  'xm') FROM ( SELECT 'POINT ZM (0 0 0 2)'::geometry g ) foo;
```

Output

```
POINT ZM (0 0 0 4)
```

Figure



See Also

ST_FlipCoordinates

7.6 Geometry Validation

7.6.1 ST_IsValid

ST_IsValid — Tests if a geometry is well-formed in 2D.

Synopsis

boolean **ST_IsValid**(geometry g);
boolean **ST_IsValid**(geometry g, integer flags);

Description

Tests if an ST_Geometry value is well-formed and valid in 2D according to the OGC rules. For geometries with 3 and 4 dimensions, the validity is still only tested in 2 dimensions. For geometries that are invalid, a PostgreSQL NOTICE is emitted providing details of why it is not valid.

For the version with the `flags` parameter, supported values are documented in ST_IsValidDetail This version does not print a NOTICE explaining invalidity.

For more information on the definition of geometry validity, refer to Section 4.4



Note
SQL-MM defines the result of ST_IsValid(NULL) to be 0, while PostGIS returns NULL.

Performed by the GEOS module.

The version accepting flags is available starting with 2.0.0.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1.



This method implements the SQL/MM specification. SQL-MM 3: 5.1.9



Note
Neither OGC-SFS nor SQL-MM specifications include a flag argument for ST_IsValid. The flag is a PostGIS extension.

Examples

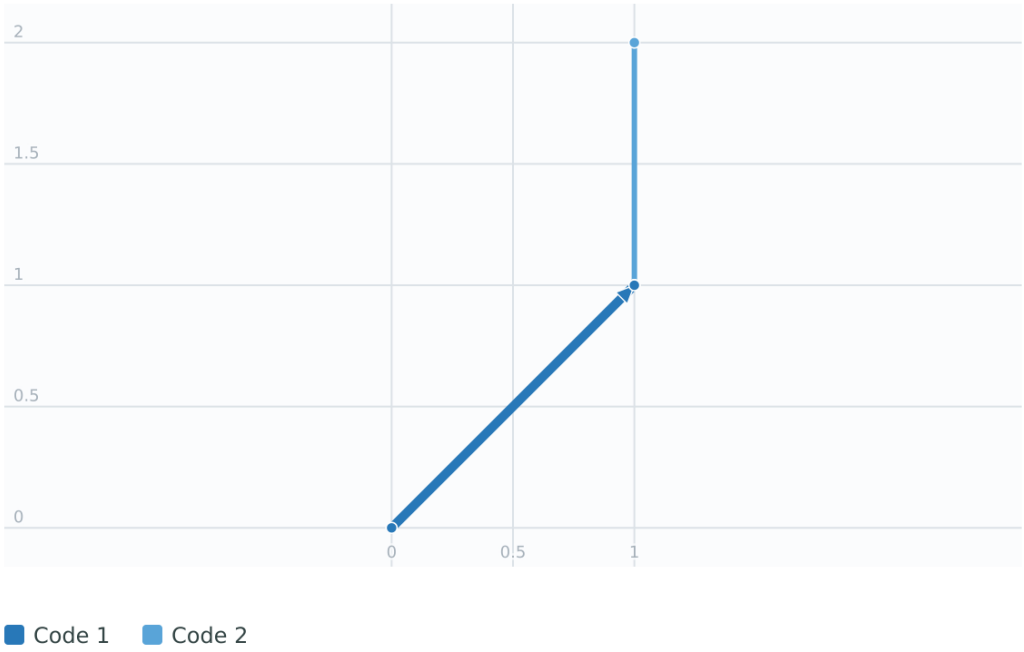
Code

```
SELECT ST_IsValid('LINESTRING(0 0,1 1)') As good_line,
ST_IsValid('POLYGON((0 0,1 1,1 2,1 1,0 0))') As bad_poly;
```

Output

```
NOTICE:  Self-intersection at or near point 0 0
 good_line | bad_poly
-----------+----------
 t         | f
```

Figure



Find tables with invalid geometries in registered geometry columns.

Code

```
CREATE TEMP TABLE invalid_geometry_columns (
    table_name text,
    column_name text,
    invalid_count bigint
);

DO $$
DECLARE
    rec record;
    invalid_count bigint;
BEGIN
    FOR rec IN
        SELECT f_table_schema, f_table_name, f_geometry_column
        FROM geometry_columns
        ORDER BY f_table_schema, f_table_name, f_geometry_column
    LOOP
        EXECUTE format(
            'SELECT count(*) FROM %I.%I WHERE NOT ST_IsValid(%I, 0)',
            rec.f_table_schema,
            rec.f_table_name,
            rec.f_geometry_column
        ) INTO invalid_count;

        IF invalid_count > 0 THEN
            INSERT INTO invalid_geometry_columns
            VALUES (
                format('%I.%I', rec.f_table_schema, rec.f_table_name),
                rec.f_geometry_column,
                invalid_count
            );
        END IF;
    END LOOP;
END;
```

```
$$;

SELECT * FROM invalid_geometry_columns;
```

See Also

ST_IsSimple, ST_IsValidReason, ST_IsValidDetail,

7.6.2 ST_IsValidDetail

ST_IsValidDetail — Returns a `valid_detail` row stating if a geometry is valid or if not a reason and a location.

Synopsis

valid_detail **ST_IsValidDetail**(geometry geom, integer flags);

Description

Returns a `valid_detail` row, containing a boolean (`valid`) stating if a geometry is valid, a varchar (`reason`) stating a reason why it is invalid and a geometry (`location`) pointing out where it is invalid.

Useful to improve on the combination of ST_IsValid and ST_IsValidReason to generate a detailed report of invalid geometries.

The optional `flags` parameter is a bitfield. It can have the following values:

- 0: Use usual OGC SFS validity semantics.
- 1: Consider certain kinds of self-touching rings (inverted shells and exverted holes) as valid. This is also known as "the ESRI flag", since this is the validity model used by those tools. Note that this is invalid under the OGC model.

Performed by the GEOS module.

Availability: 2.0.0

Examples

This example returns the first three rejects from a successful quintuplet experiment.

Code

```
SELECT gid, reason(ST_IsValidDetail(geom)), location(ST_IsValidDetail(geom)) as location
FROM
(SELECT ST_MakePolygon(ST_ExteriorRing(e.buff), array_agg(f.line)) As geom, gid
FROM (SELECT ST_Buffer(ST_Point(x1*10,y1), z1) As buff, x1*10 + y1*100 + z1*1000 As gid
    FROM generate_series(-4,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,8) z1
    WHERE x1 > y1*0.5 AND z1 < x1*y1) As e
    INNER JOIN (SELECT ST_Translate(ST_ExteriorRing(ST_Buffer(ST_Point(x1*10,y1), z1)),y1*1, ↩
     z1*2) As line
    FROM generate_series(-3,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,10) z1
    WHERE x1 > y1*0.75 AND z1 < x1*y1) As f
ON (ST_Area(e.buff) > 78 AND ST_Contains(e.buff, f.line))
GROUP BY gid, e.buff) As quintuplet_experiment
WHERE ST_IsValid(geom) = false
ORDER BY gid
LIMIT 3;
```

Output

```
 gid  |      reason       |  location
------+-------------------+-------------
 5330 | Self-intersection | POINT(32 5)
 5340 | Self-intersection | POINT(42 5)
 5350 | Self-intersection | POINT(52 5)
```

This example reports a valid LineString.

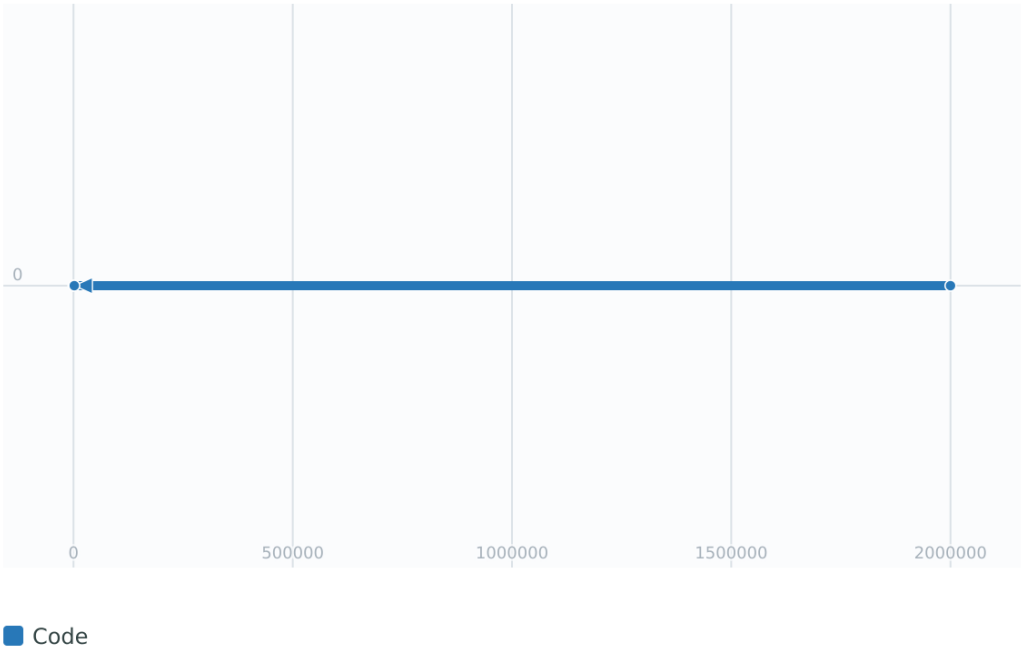
Code

```
SELECT * FROM ST_IsValidDetail('LINESTRING(220227 150406,2220227 150407,222020 150410)');
```

Output

```
 valid | reason | location
-------+--------+----------
 t     |        |
```

Figure



See Also

ST_IsValid, ST_IsValidReason

7.6.3 ST_IsValidReason

ST_IsValidReason — Returns text stating if a geometry is valid, or a reason for invalidity.

Synopsis

text **ST_IsValidReason**(geometry geomA);
text **ST_IsValidReason**(geometry geomA, integer flags);

Description

Returns text stating if a geometry is valid, or if invalid a reason why.

Useful in combination with ST_IsValid to generate a detailed report of invalid geometries and reasons.

Allowed `flags` are documented in ST_IsValidDetail.

Performed by the GEOS module.

Availability: 1.4

Availability: 2.0 version taking flags.

Examples

This example uses an invalid bow-tie polygon.

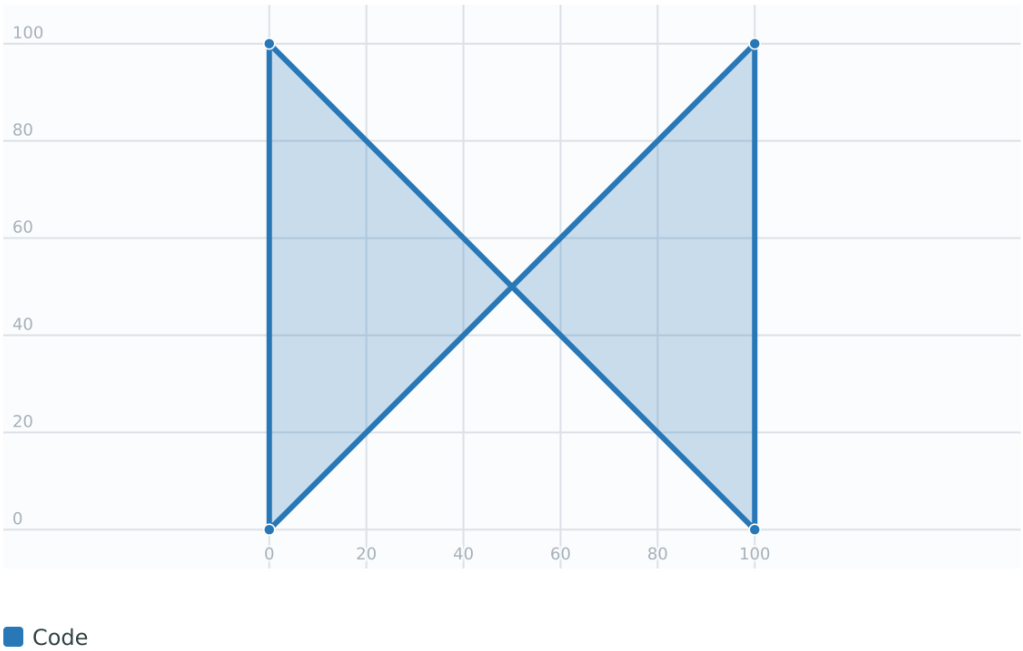
Code

```
SELECT ST_IsValidReason(
    'POLYGON ((100 200,100 100,200 200,
     200 100,100 200))'::geometry) as validity_info;
```

Output

```
Self-intersection[150 150]
```

Figure



This example returns the first three rejects from a successful quintuplet experiment.

Code

```
SELECT gid, ST_IsValidReason(geom) as validity_info
FROM
(SELECT ST_MakePolygon(ST_ExteriorRing(e.buff), array_agg(f.line)) As geom, gid
FROM (SELECT ST_Buffer(ST_Point(x1*10,y1), z1) As buff, x1*10 + y1*100 + z1*1000 As gid
    FROM generate_series(-4,6) x1
```

```
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,8) z1
    WHERE x1 > y1*0.5 AND z1 < x1*y1) As e
    INNER JOIN (SELECT ST_Translate(ST_ExteriorRing(ST_Buffer(ST_Point(x1*10,y1), z1)),y1*1, ↩
     z1*2) As line
    FROM generate_series(-3,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,10) z1
    WHERE x1 > y1*0.75 AND z1 < x1*y1) As f
ON (ST_Area(e.buff) > 78 AND ST_Contains(e.buff, f.line))
GROUP BY gid, e.buff) As quintuplet_experiment
WHERE ST_IsValid(geom) = false
ORDER BY gid
LIMIT 3;
```

Output

```
 gid  |      validity_info
------+--------------------------
 5330 | Self-intersection [32 5]
 5340 | Self-intersection [42 5]
 5350 | Self-intersection [52 5]
```

Simple example.

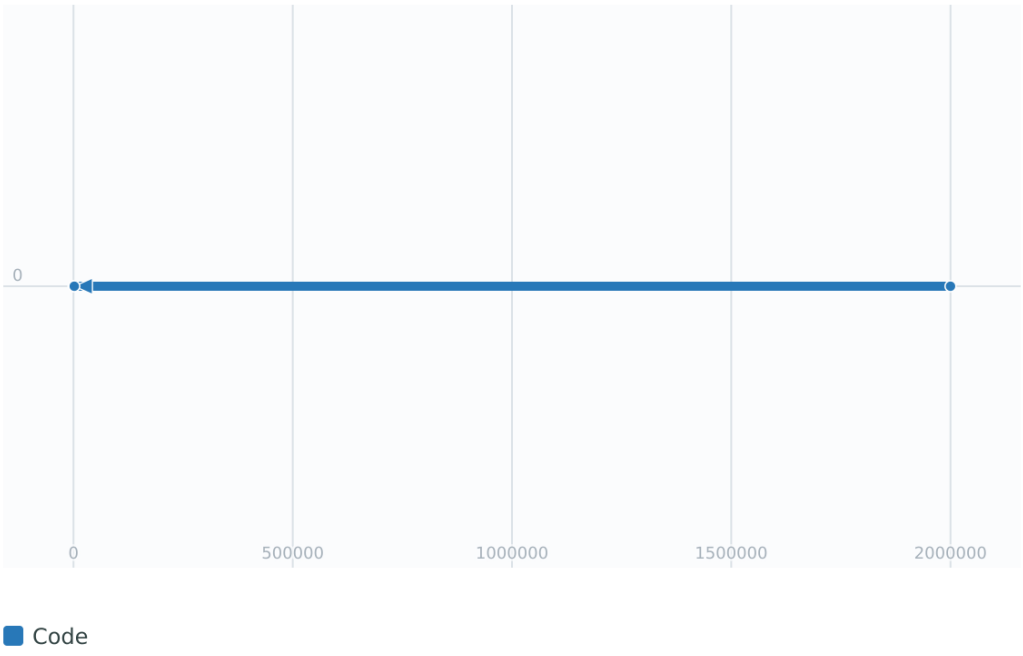
Code

```
SELECT ST_IsValidReason('LINESTRING(220227 150406,2220227 150407,222020 150410)');
```

Output

```
 st_isvalidreason
------------------
 Valid Geometry
```

Figure



See Also

ST_IsValid, ST_Summary

7.6.4 ST_MakeValid

ST_MakeValid — Attempts to make an invalid geometry valid without losing vertices.

Synopsis

geometry **ST_MakeValid**(geometry input);
geometry **ST_MakeValid**(geometry input, text params);

Description

The function attempts to create a valid representation of a given invalid geometry without losing any of the input vertices. Valid geometries are returned unchanged.

Supported inputs are: POINTS, MULTIPOINTS, LINESTRINGS, MULTILINESTRINGS, POLYGONS, MULTIPOLYGONS and GEOMETRYCOLLECTIONS containing any mix of them.

In case of full or partial dimensional collapses, the output geometry may be a collection of lower-to-equal dimension geometries, or a geometry of lower dimension.

Single polygons may become multi-geometries in case of self-intersections.

The `params` argument can be used to supply an options string to select the method to use for building valid geometry. The options string is in the format "method=linework|structure keepcollapsed=true|false". If no "params" argument is provided, the "linework" algorithm will be used as the default.

The "method" key has two values.

- "linework" is the default algorithm, and builds valid geometries by first extracting all lines, noding that linework together, then building a value output from the linework. The requirement that no vertices are lost can generate complex collections as outputs.
- "structure" is an algorithm that distinguishes between interior and exterior rings, building new geometry by unioning exterior rings, and then differencing all interior rings. The results tend to be more intuitive, where collapsed lines are discarded in the output.

The "keepcollapsed" key is only valid for the "structure" algorithm, and takes a value of "true" or "false".

- Keep collapsed of "false" is the default, and means that geometry components that collapse to a lower dimensionality, for example a one-point linestring will be dropped.
- Keep collapsed of "true" means geometry components that collapse to a lower dimensionality will be retained.

Performed by the GEOS module.

Availability: 2.0.0

Enhanced: 2.0.1, speed improvements

Enhanced: 2.1.0, added support for GEOMETRYCOLLECTION and MULTIPOINT.

Enhanced: 3.1.0, added removal of Coordinates with NaN values.

Enhanced: 3.2.0, added algorithm options, 'linework' and 'structure' which requires GEOS >= 3.10.0.

 This function supports 3d and will not drop the z-index.

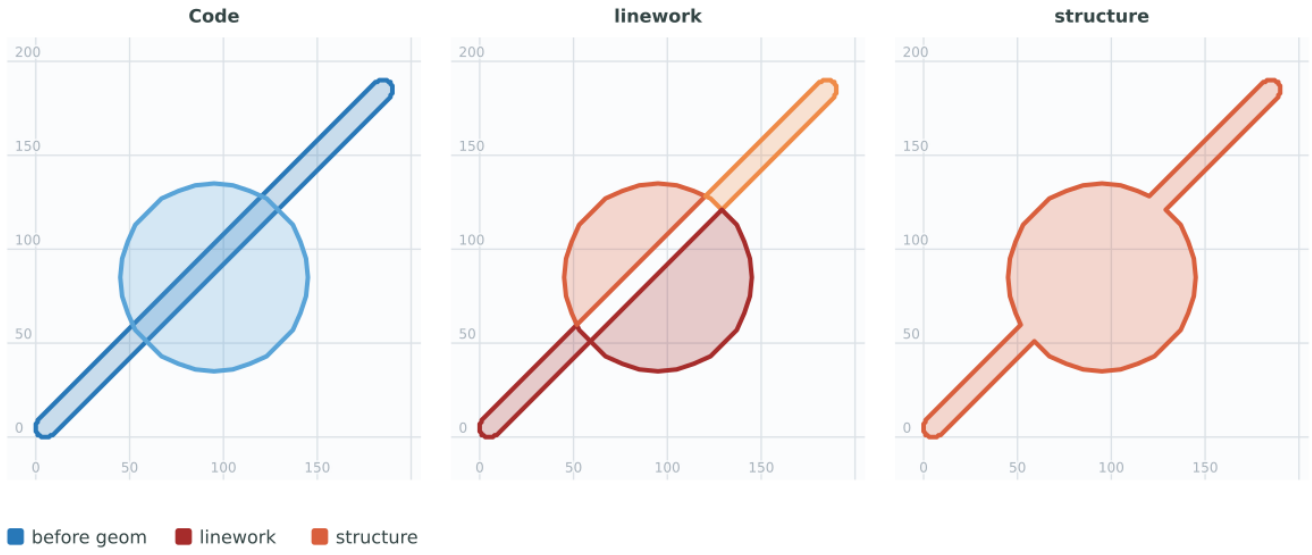
Examples

Make valid a MultiPolygon of two overlapping polygons using the linework and structure methods.

Code

```
SELECT ST_MakeValid(before_geom) AS linework,
  ST_MakeValid(before_geom, 'method=structure') AS structure
FROM (SELECT 'MULTIPOLYGON(((186 194,187 194,188 195,189 195,190 195,
191 195,192 195,193 194,194 194,194 193,195 192,195 191,
195 190,195 189,195 188,194 187,194 186,14 6,13 6,12 5,11 5,
10 5,9 5,8 5,7 6,6 6,6 7,5 8,5 9,5 10,5 11,5 12,6 13,6 14,186 194)),
((150 90,149 80,146 71,142 62,135 55,128 48,119 44,110 41,100 40,
90 41,81 44,72 48,65 55,58 62,54 71,51 80,50 90,51 100,
54 109,58 118,65 125,72 132,81 136,90 139,100 140,110 139,
119 136,128 132,135 125,142 118,146 109,149 100,150 90)))'::geometry AS before_geom) AS f;
```

Figure

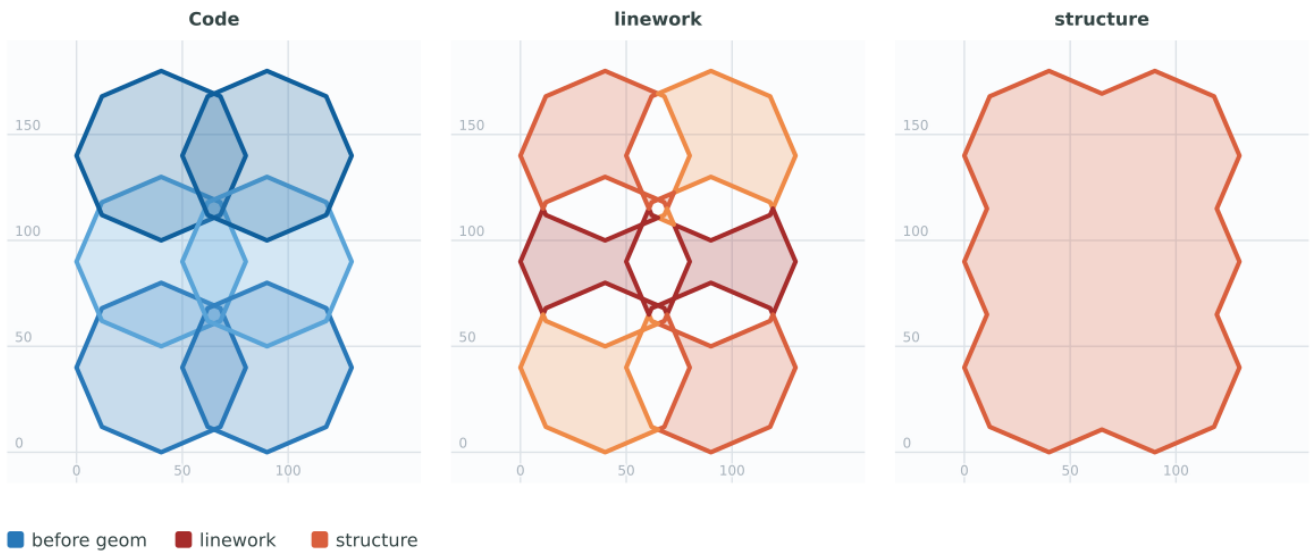


Make valid a MultiPolygon of six overlapping polygons using the linework and structure methods.

Code

```
SELECT ST_MakeValid(before_geom) AS linework,
  ST_MakeValid(before_geom, 'method=structure') AS structure
FROM (SELECT 'MULTIPOLYGON(((91 50,79 22,51 10,23 22,11 50,23 78,51 90,79 78,91 50)),
      ((91 100,79 72,51 60,23 72,11 100,23 128,51 140,79 128,91 100)),
      ((91 150,79 122,51 110,23 122,11 150,23 178,51 190,79 178,91 150)),
      ((141 50,129 22,101 10,73 22,61 50,73 78,101 90,129 78,141 50)),
      ((141 100,129 72,101 60,73 72,61 100,73 128,101 140,129 128,141 100)),
      ((141 150,129 122,101 110,73 122,61 150,73 178,101 190,129 178,141 150)))'::geometry ↩
    AS before_geom) AS c;
```

Figure



Collapsed Geometry Examples

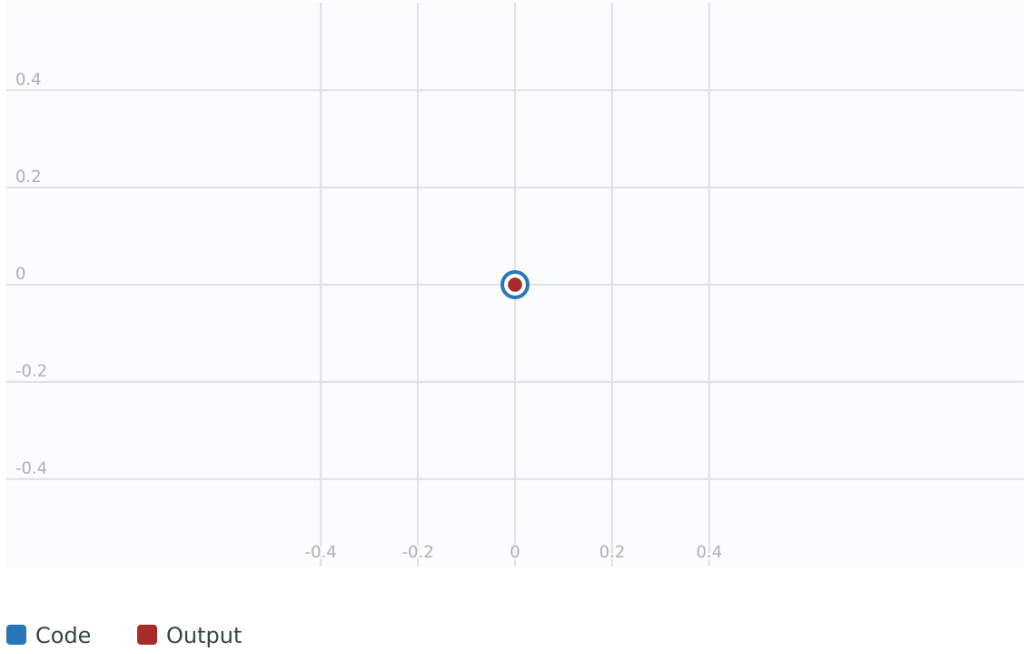
Code

```
SELECT ST_MakeValid(
'LINESTRING(0 0,0 0)',
'method=structure keepcollapsed=true'
);
```

Output

```
POINT(0 0)
```

Figure



Code

```
SELECT ST_MakeValid(
'LINESTRING(0 0,0 0)',
'method=structure keepcollapsed=false'
);
```

Output

```
LINESTRING EMPTY
```

See Also

ST_IsValid, ST_Collect, ST_CollectionExtract

7.7 Spatial Reference System Functions

7.7.1 ST_InverseTransformPipeline

ST_InverseTransformPipeline — Return a new geometry with coordinates transformed to a different spatial reference system using the inverse of a defined coordinate transformation pipeline.

Synopsis

geometry **ST_InverseTransformPipeline**(geometry geom, text pipeline, integer to_srid);

Description

Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline to go in the inverse direction.

Refer to ST_TransformPipeline for details on writing a transformation pipeline.

Availability: 3.4.0

The SRID of the input geometry is ignored, and the SRID of the output geometry will be set to zero unless a value is provided via the optional `to_srid` parameter. When using ST_TransformPipeline the pipeline is executed in a forward direction. Using `ST_InverseTransformPipeline()` the pipeline is executed in the inverse direction.

Transforms using pipelines are a specialised version of ST_Transform. In most cases `ST_Transform` will choose the correct operations to convert between coordinate systems, and should be preferred.

Examples

Change WGS 84 long lat to UTM 31N using the EPSG:16031 conversion

This example transforms in the inverse direction.

Code

```
SELECT ST_InverseTransformPipeline(
        'POINT(426857.9877165967 5427937.523342293)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::16031'
    ) AS wgs_geom;
```

Output

```
POINT(2 48.99999999999999)
```

Figure



GDA2020 example.

This example uses ST_Transform with automatic selection of a conversion pipeline.

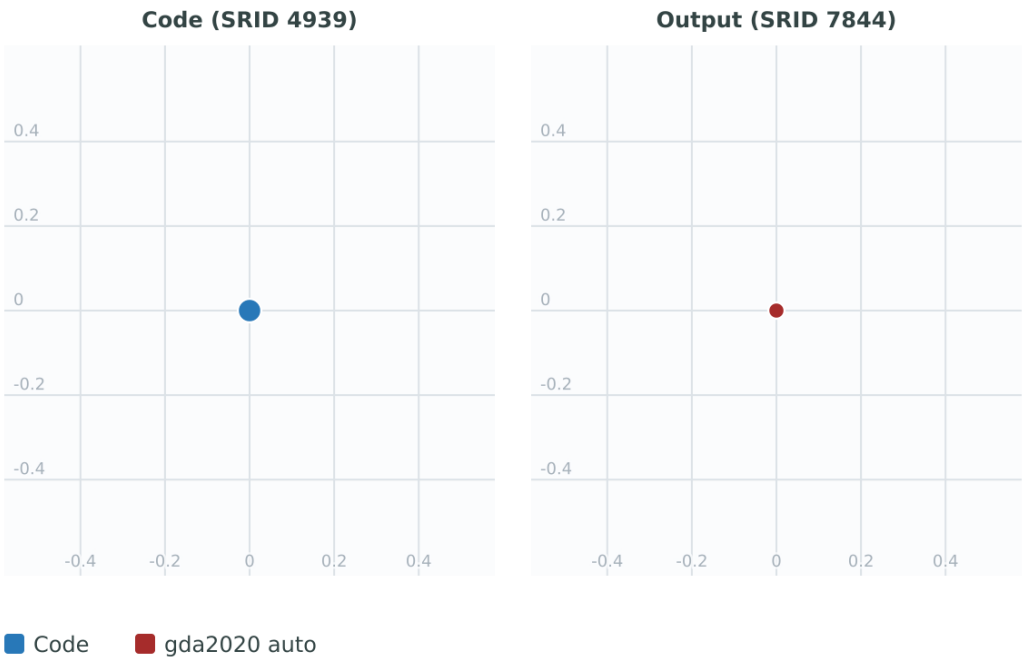
Code

```
SELECT ST_Transform('SRID=4939;POINT(143.0 -37.0)'::geometry, 7844) AS gda2020_auto;
```

Output

```
POINT(143.00000635638918 -36.999986706128176)
```

Figure



See Also

ST_Transform, ST_TransformPipeline

7.7.2 ST_SetSRID

ST_SetSRID — Set the SRID on a geometry.

Synopsis

geometry **ST_SetSRID**(geometry geom, integer srid);

Description

Sets the SRID on a geometry to a particular integer value. Useful in constructing bounding boxes for queries.



Note
This function does not transform the geometry coordinates in any way - it simply sets the meta data defining the spatial reference system the geometry is assumed to be in. Use ST_Transform if you want to transform the geometry into a new projection.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1.


This method supports Circular Strings and Curves.

Examples

Mark a point as WGS 84 longitude/latitude.

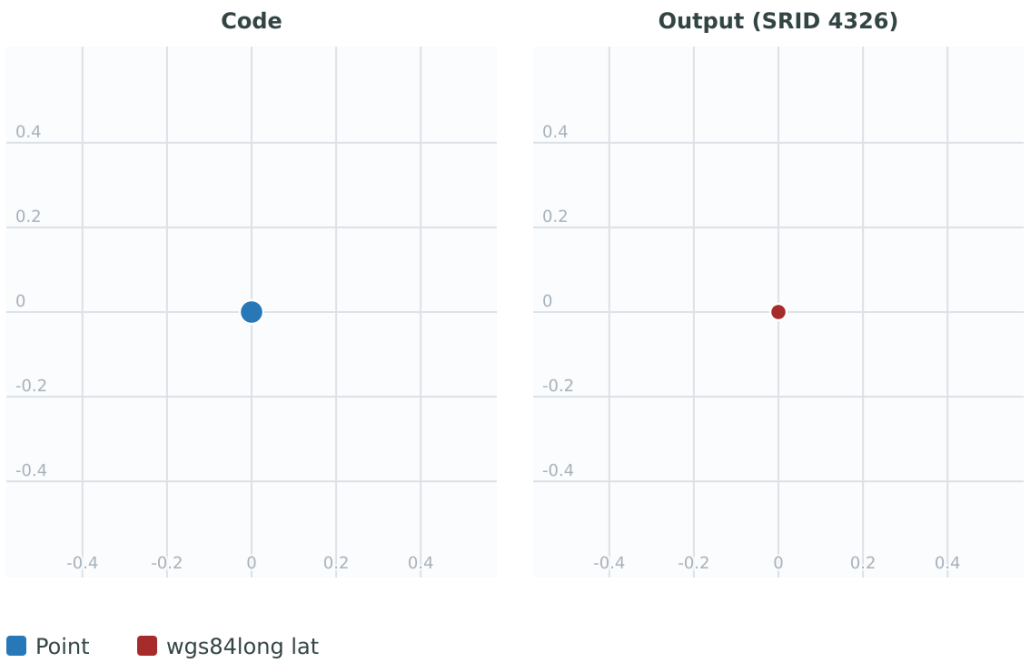
Code

```
SELECT ST_SetSRID(ST_Point(-123.365556, 48.428611), 4326) AS wgs84long_lat;
```

Output

```
SRID=4326;POINT(-123.365556 48.428611)
```

Figure



Mark a point as WGS 84 longitude/latitude and then transform it to Web Mercator.

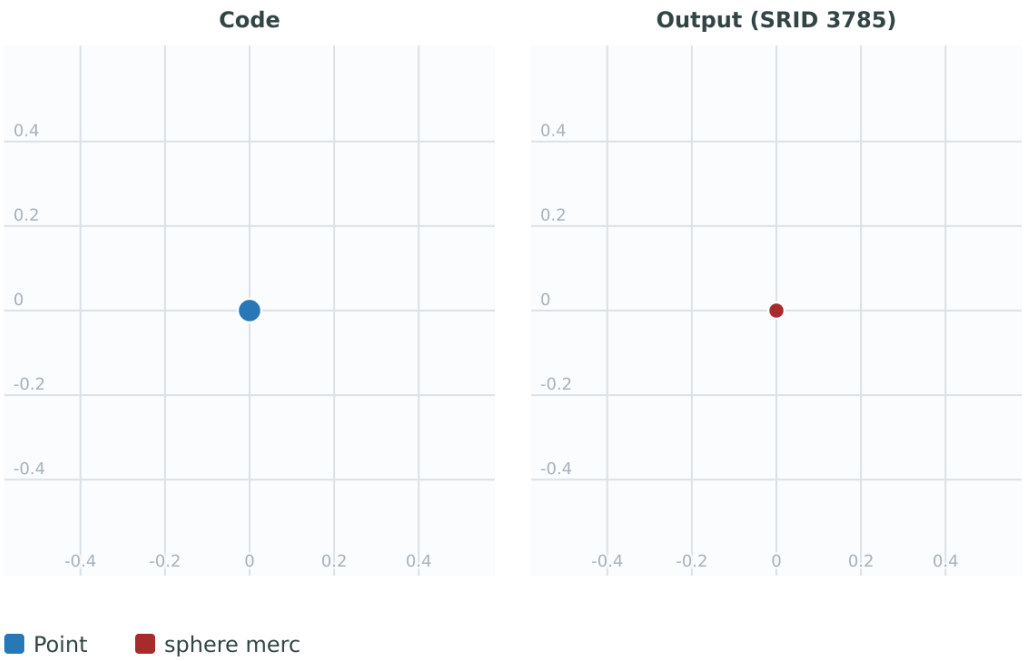
Code

```
SELECT ST_Transform(ST_SetSRID(ST_Point(-123.365556, 48.428611), 4326), 3785) AS sphere_merc ↩
    ;
```

Output

```
SRID=3785;POINT(-13732990.8753491 6178458.96425423)
```

Figure



See Also

Section 4.5, ST_SRID, ST_Transform, UpdateGeometrySRID

7.7.3 ST_SRID

ST_SRID — Returns the spatial reference identifier for a geometry.

Synopsis

integer **ST_SRID**(geometry g1);

Description

Returns the spatial reference identifier for the ST_Geometry as defined in spatial_ref_sys table. Section 4.5



Note
spatial_ref_sys table is a table that catalogs all spatial reference systems known to PostGIS and is used for transformations from one spatial reference system to another. So verifying you have the right spatial reference system identifier is important if you plan to ever transform your geometries.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.5

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_SRID(ST_GeomFromText('POINT(-71.1043 42.315)', 4326));
```

Output

```
 st_srid
---------
    4326
(1 row)
```

See Also

Section 4.5, ST_SetSRID, ST_Transform, ST_SRID, ST_SRID

7.7.4 ST_Transform

ST_Transform — Return a new geometry with coordinates transformed to a different spatial reference system.

Synopsis

geometry **ST_Transform**(geometry g1, integer srid);
geometry **ST_Transform**(geometry geom, text to_proj);
geometry **ST_Transform**(geometry geom, text from_proj, text to_proj);
geometry **ST_Transform**(geometry geom, text from_proj, integer to_srid);

Description

Returns a new geometry with its coordinates transformed to a different spatial reference system. The destination spatial reference `to_srid` may be identified by a valid SRID integer parameter (i.e. it must exist in the `spatial_ref_sys` table). Alternatively, a spatial reference defined as a PROJ.4 string can be used for `to_proj` and/or `from_proj`, however these methods are not optimized. If the destination spatial reference system is expressed with a PROJ.4 string instead of an SRID, the SRID of the output geometry will be set to zero. With the exception of functions with `from_proj`, input geometries must have a defined SRID.

ST_Transform is often confused with ST_SetSRID. ST_Transform actually changes the coordinates of a geometry from one spatial reference system to another, while ST_SetSRID() simply changes the SRID identifier of the geometry.

ST_Transform automatically selects a suitable conversion pipeline given the source and target spatial reference systems. To use a specific conversion method, use ST_TransformPipeline.



Note
Requires PostGIS be compiled with PROJ support. Use PostGIS_Full_Version to confirm you have PROJ support compiled in.



Note
If using more than one transformation, it is useful to have a functional index on the commonly used transformations to take advantage of index usage.



Note

Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

Enhanced: 2.3.0 support for direct PROJ.4 text was introduced.



This method implements the SQL/MM specification. SQL-MM 3: 5.1.6



This method supports Circular Strings and Curves.



This function supports Polyhedral surfaces.

Examples

Change Massachusetts state plane US feet geometry to WGS 84 long lat

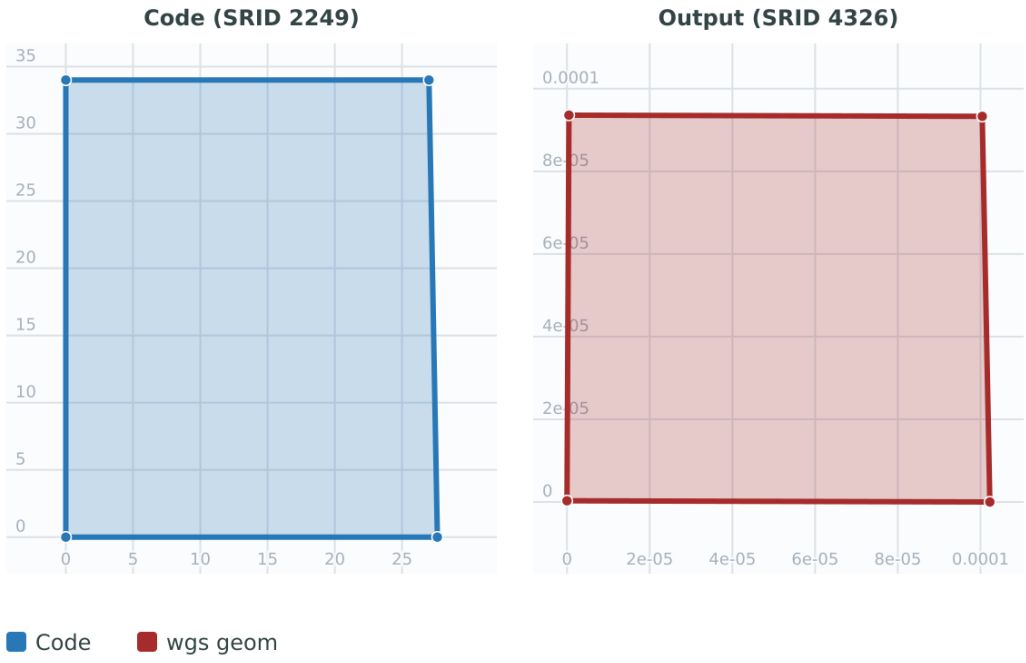
Code

```
SELECT ST_Transform(
  'SRID=2249;POLYGON((743238 2967416,743238 2967450,
    743265 2967450,743265.625 2967416,743238 2967416))'::geometry,
  4326) AS wgs_geom;
```

Output

```
SRID=4326;POLYGON((-71.1776848522251 42.3902896512903,-71.17768437663261 42.390382947801015,-71.17758443054647 ↩
    42.390382667791854,-71.17758259272304 42.390289364798825,-71.1776848522251 42.3902896512903))
```

Figure



3D Circular String example.

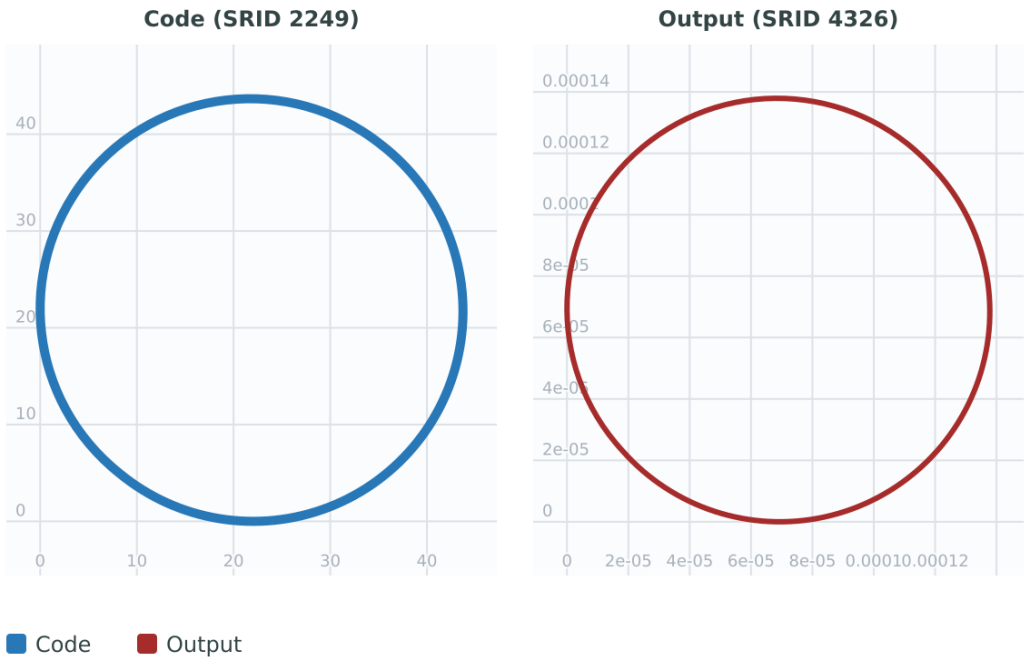
Code

```
SELECT ST_Transform('SRID=2249;CIRCULARSTRING(743238 2967416 1,743238 2967450 2,743265 ↩
    2967450 3,743265.625 2967416 3,743238 2967416 4)'::geometry,
    4326);
```

Output

```
 SRID=4326;CIRCULARSTRING(-71.1776848522251 42.3902896512903 1,-71.17768437663261 42.390382947801015 2,
-71.17758443054647 42.390382667791854 3,
-71.17758259272304 42.390289364798825 3,-71.1776848522251 42.3902896512903 4)
```

Figure



Example of creating a partial functional index. For tables where you are not sure all the geometries will be filled in, its best to use a partial index that leaves out null geometries which will both conserve space and make your index smaller and more efficient.

Code

```
CREATE INDEX idx_geom_26986_parcels
  ON parcels
  USING gist
  (ST_Transform(geom, 26986))
  WHERE geom IS NOT NULL;
```

Examples of using PROJ.4 text to transform with custom spatial references.

Find intersection of two polygons near the North pole, using a custom Gnomic projection. See http://boundlessgeo.com/2012/02/flattenin the-peel/.

Code

```
WITH data AS (
  SELECT
    ST_GeomFromText('POLYGON((170 50,170 72,-130 72,-130 50,170 50))', 4326) AS p1,
    ST_GeomFromText('POLYGON((-170 68,-170 90,-141 90,-141 68,-170 68))', 4326) AS p2,
    '+proj=gnom +ellps=WGS84 +lat_0=70 +lon_0=-160 +no_defs'::text AS gnom
```

```
)
SELECT ST_Transform(ST_Intersection(ST_Transform(p1, gnom), ST_Transform(p2, gnom)),
  gnom, 4326)
FROM data;
```

Output

```
POLYGON((-140.9999840290591 73.42768865981984,-141 68,-170 68,-170.0000026915361 74.05492179637055,-140.9999840290591 ↩
    73.42768865981984))
```

Figure



Configuring transformation behavior

Sometimes coordinate transformation involving a grid-shift can fail, for example if PROJ.4 has not been built with grid-shift files or the coordinate does not lie within the range for which the grid shift is defined. By default, PostGIS will throw an error if a grid shift file is not present, but this behavior can be configured on a per-SRID basis either by testing different `to_proj` values of PROJ.4 text, or altering the `proj4text` value within the `spatial_ref_sys` table.

For example, the proj4text parameter +datum=NAD87 is a shorthand form for the following +nadgrids parameter:

Code

```
+nadgrids=@conus, @alaska, @ntv2_0.gsb, @ntv1_can.dat
```

The @ prefix means no error is reported if the files are not present, but if the end of the list is reached with no file having been appropriate (ie. found and overlapping) then an error is issued.

If, conversely, you wanted to ensure that at least the standard files were present, but that if all files were scanned without a hit a null transformation is applied you could use:

Code

```
+nadgrids=@conus, @alaska, @ntv2_0.gsb, @ntv1_can.dat, null
```

The null grid shift file is a valid grid shift file covering the whole world and applying no shift. So for a complete example, if you wanted to alter PostGIS so that transformations to SRID 4267 that didn't lie within the correct range did not throw an ERROR, you would use the following:

Code

```
UPDATE spatial_ref_sys SET proj4text = '+proj=longlat +ellps=clrk66 +nadgrids=@conus,@alaska ↩
    ,@ntv2_0.gsb,@ntv1_can.dat,null +no_defs' WHERE srid = 4267;
```

See Also

Section 4.5, ST_SetSRID, ST_SRID, UpdateGeometrySRID, ST_TransformPipeline

7.7.5 ST_TransformPipeline

ST_TransformPipeline — Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline.

Synopsis

geometry **ST_TransformPipeline**(geometry g1, text pipeline, integer to_srid);

Description

Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline.

Transformation pipelines are defined using any of the following string formats:

- `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`. Note that a simple `EPSG:CODE` string does not uniquely identify a coordinate operation: the same EPSG code can be used for a CRS definition.
- A PROJ pipeline string of the form: `+proj=pipeline ...`. Automatic axis normalisation will not be applied, and if necessary the caller will need to add an additional pipeline step, or remove `axisswap` steps.
- Concatenated operations of the form: `urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,c`

Availability: 3.4.0

The SRID of the input geometry is ignored, and the SRID of the output geometry will be set to zero unless a value is provided via the optional `to_srid` parameter. When using `ST_TransformPipeline()` the pipeline is executed in a forward direction. Using ST_InverseTransformPipeline the pipeline is executed in the inverse direction.

Transforms using pipelines are a specialised version of ST_Transform. In most cases `ST_Transform` will choose the correct operations to convert between coordinate systems, and should be preferred.

Examples

Change WGS 84 long lat to UTM 31N using the EPSG:16031 conversion

This example transforms in the forward direction.

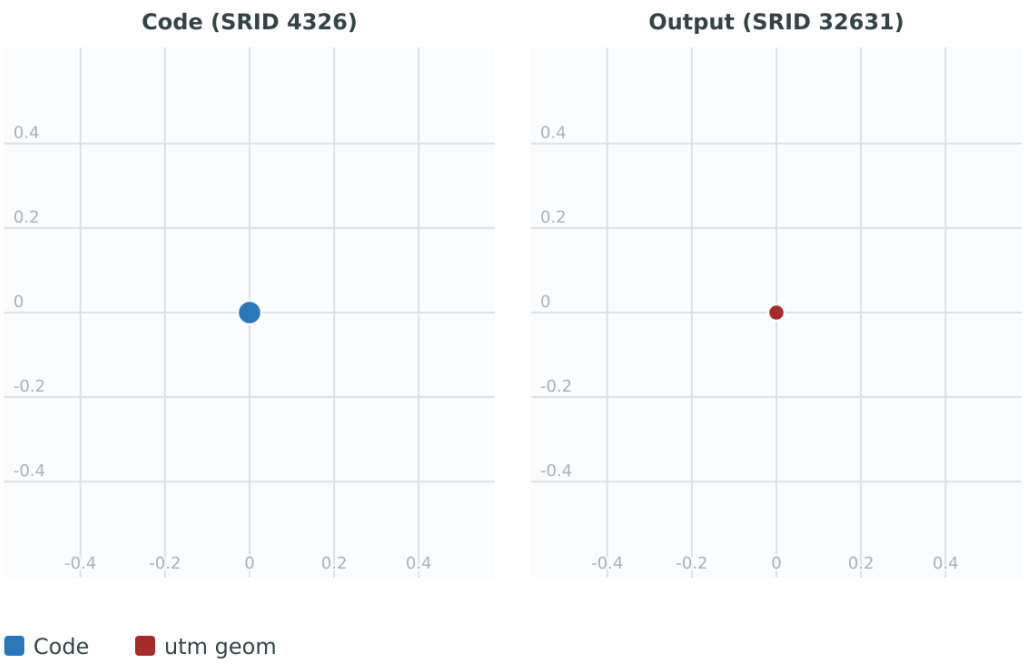
Code

```
SELECT
    ST_SetSRID(ST_TransformPipeline(
        'SRID=4326;POINT(2 49)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::16031'
    ), 32631) AS utm_geom;
```

Output

```
SRID=32631;POINT(426857.9877165967 5427937.523342293)
```

Figure



This example transforms in the inverse direction.

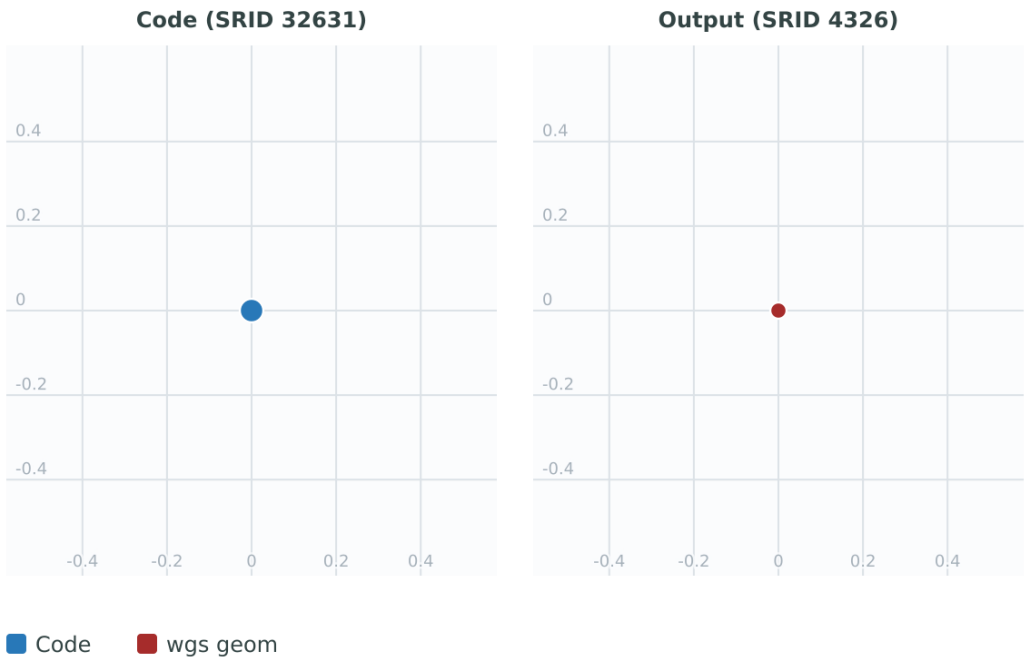
Code

```
SELECT
    ST_SetSRID(ST_InverseTransformPipeline(
        'SRID=32631;POINT(426857.9877165967 5427937.523342293)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::16031'
    ), 4326) AS wgs_geom;
```

Output

```
SRID=4326;POINT(2 48.99999999999999)
```

Figure



GDA2020 example.

This example uses ST_Transform with automatic selection of a conversion pipeline.

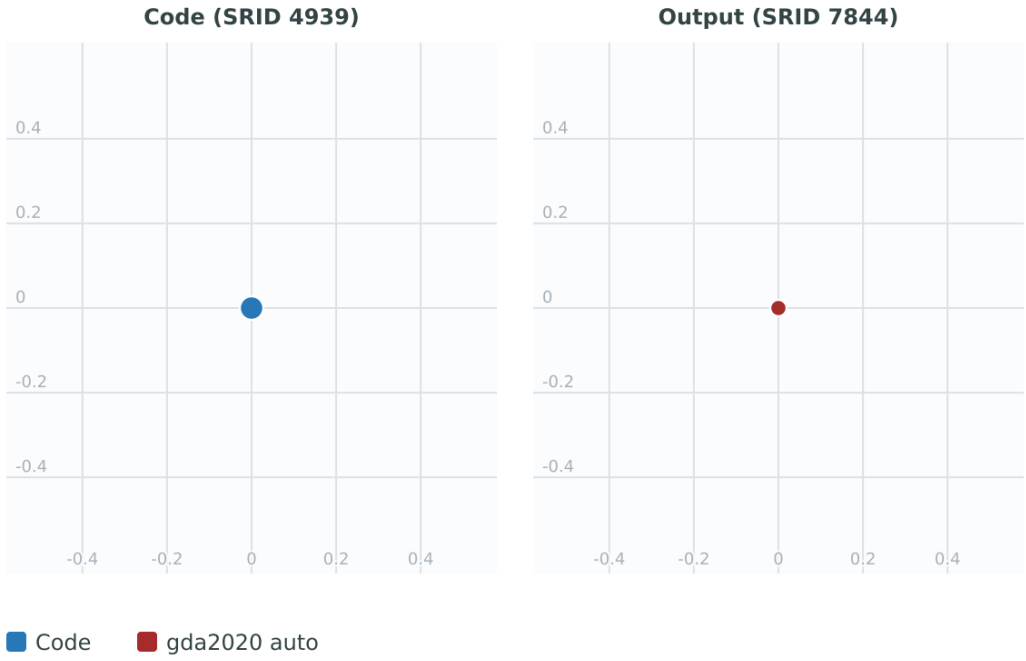
Code

```
SELECT ST_Transform('SRID=4939;POINT(143.0 -37.0)'::geometry, 7844) AS gda2020_auto;
```

Output

```
POINT(143.00000635638918 -36.999986706128176)
```

Figure



Using a defined conversion (EPSG:8447)

Code

```
SELECT
    ST_TransformPipeline(
        'SRID=4939;POINT(143.0 -37.0)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::8447'
    ) AS gda2020_code;
```

Output

```
POINT(143.0000063280214 -36.999986718287545)
```

This example uses a PROJ pipeline definition matching EPSG:8447, as returned from projinfo. Any axisswap steps must be removed.

Code

```
SELECT
    ST_TransformPipeline(
        'SRID=4939;POINT(143.0 -37.0)'::geometry,
        '+proj=pipeline
   +step +proj=unitconvert +xy_in=deg +xy_out=rad
   +step +proj=hgridshift +grids=au_icsm_GDA94_GDA2020_conformal_and_distortion.tif
   +step +proj=unitconvert +xy_in=rad +xy_out=deg'
    ) AS gda2020_pipeline;
```

Output

```
POINT(143.0000063280214 -36.999986718287545)
```

See Also

ST_Transform, ST_InverseTransformPipeline

7.7.6 postgis_srs_codes

postgis_srs_codes — Return the list of SRS codes associated with the given authority.

Synopsis

setof text **postgis_srs_codes**(text auth_name);

Description

Returns a set of all `auth_srid` for the given `auth_name`.

Availability: 3.4.0

Proj version 6+

Examples

List the first ten codes associated with the EPSG authority.

Code

```
SELECT * FROM postgis_srs_codes('EPSG') ORDER BY 1 LIMIT 10;
```

Output

```
10156
10157
10158
10160
10162
10163
10164
10165
10166
10167
```

See Also

postgis_srs, postgis_srs_all, postgis_srs_search

7.7.7 postgis_srs

postgis_srs — Return a metadata record for the requested authority and srid.

Synopsis

setof record **postgis_srs**(text auth_name, text auth_srid);

Description

Returns a metadata record for the requested `auth_srid` for the given `auth_name`. The record will have the `auth_name`, `auth_srid`, `srname`, `srtext`, `proj4text`, and the corners of the area of usage, `point_sw` and `point_ne`.

Availability: 3.4.0

Proj version 6+

Examples

Get the metadata for EPSG:3005.

Code

```
SELECT auth_name, auth_srid, srname,point_sw AS point_sw,point_ne AS point_ne
FROM postgis_srs('EPSG', '3005');
```

Output

```
-[ RECORD 1 ]-------------------
auth_name | EPSG
auth_srid | 3005
srname    | NAD83 / BC Albers
point_sw  | POINT(-139.04 48.25)
point_ne  | POINT(-114.08 60.01)
```

See Also

postgis_srs_codes, postgis_srs_all, postgis_srs_search

7.7.8 postgis_srs_all

postgis_srs_all — Return metadata records for every spatial reference system in the underlying Proj database.

Synopsis

setof record **postgis_srs_all**(void);

Description

Returns a set of all metadata records in the underlying Proj database. The records will have the `auth_name`, `auth_srid`, `srname`, `srtext`, `proj4text`, and the corners of the area of usage, `point_sw` and `point_ne`.

Availability: 3.4.0

Proj version 6+

Examples

Get the first 10 metadata records from the Proj database.

Code

```
SELECT auth_name, auth_srid, srname FROM postgis_srs_all() ORDER BY auth_name, auth_srid ↩
    LIMIT 10;
```

Output

```
 auth_name | auth_srid |                                srname
-----------+-----------+-----------------------------------------------------------------------
 EPSG      | 10156     | ETRS89 + MSL UK & Ireland VORF08 depth
 EPSG      | 10157     | ETRS89 + CD UK & Ireland VORF08 depth
 EPSG      | 10158     | S34J-IRF
 EPSG      | 10160     | S34J reconstruction east-orientated
 EPSG      | 10162     | JGD2011 / Japan Plane Rectangular CS I + JGD2011 (vertical) height
 EPSG      | 10163     | JGD2011 / Japan Plane Rectangular CS II + JGD2011 (vertical) height
 EPSG      | 10164     | JGD2011 / Japan Plane Rectangular CS III + JGD2011 (vertical) height
 EPSG      | 10165     | JGD2011 / Japan Plane Rectangular CS IV + JGD2011 (vertical) height
 EPSG      | 10166     | JGD2011 / Japan Plane Rectangular CS V + JGD2011 (vertical) height
 EPSG      | 10167     | JGD2011 / Japan Plane Rectangular CS VI + JGD2011 (vertical) height
```

See Also

postgis_srs_codes, postgis_srs, postgis_srs_search

7.7.9 postgis_srs_search

postgis_srs_search — Return metadata records for projected coordinate systems that have areas of usage that fully contain the bounds parameter.

Synopsis

setof record **postgis_srs_search**(geometry bounds, text auth_name=EPSG);

Description

Return a set of metadata records for projected coordinate systems that have areas of usage that fully contain the bounds parameter. Each record will have the `auth_name`, `auth_srid`, `srname`, `srtext`, `proj4text`, and the corners of the area of usage, `point_sw` and `point_ne`.

The search only looks for projected coordinate systems, and is intended for users to explore the possible systems that work for the extent of their data.

Availability: 3.4.0

Proj version 6+

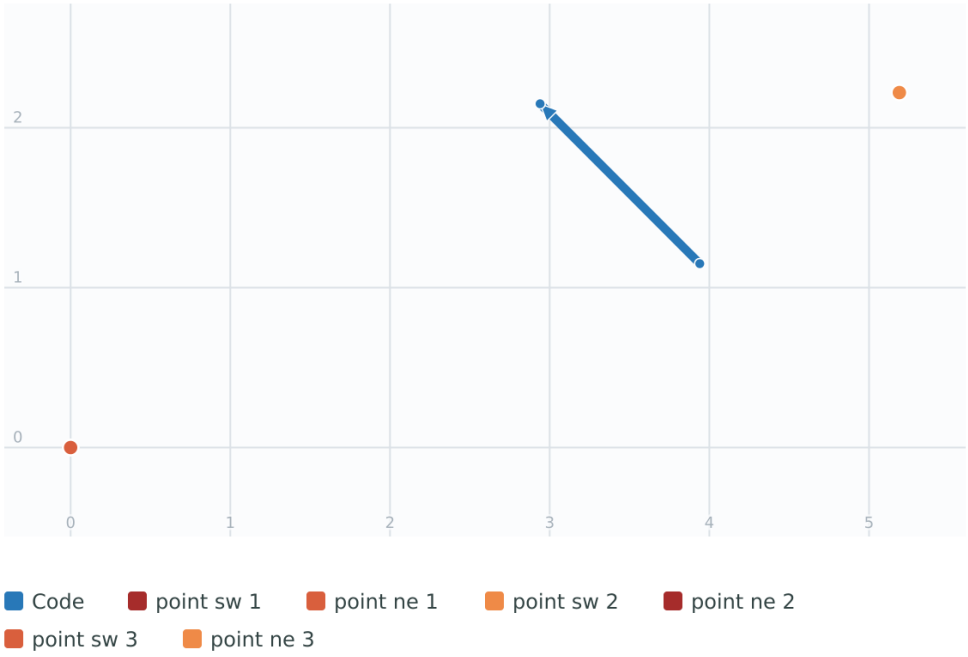
Examples

Search for projected coordinate systems in Louisiana.

Code

```
SELECT auth_name, auth_srid, srname,point_sw AS point_sw,point_ne AS point_ne
FROM postgis_srs_search('SRID=4326;LINESTRING(-90 30,-91 31)')
WHERE auth_srid IN ('2801', '3452', '3457')
ORDER BY auth_name, auth_srid::integer
```

Figure



Scan a table for max extent and find projected coordinate systems that might suit.

Code

```
WITH ext AS (
  SELECT ST_Extent(geom) AS geom, Max(ST_SRID(geom)) AS srid
  FROM foo
)
SELECT auth_name, auth_srid, srname,
  point_sw AS point_sw,
  point_ne AS point_ne
FROM ext
```

```
CROSS JOIN postgis_srs_search(ST_SetSRID(ext.geom, ext.srid))
LIMIT 3;
```

See Also

postgis_srs_codes, postgis_srs_all, postgis_srs

7.8 Geometry Input

7.8.1 Well-Known Text (WKT)

7.8.1.1 ST_BdPolyFromText

ST_BdPolyFromText — Construct a Polygon given an arbitrary collection of closed linestrings as a MultiLineString Well-Known text representation.

Synopsis

geometry **ST_BdPolyFromText**(text WKT, integer srid);

Description

Construct a Polygon given an arbitrary collection of closed linestrings as a MultiLineString Well-Known text representation.



Note
Throws an error if WKT is not a MULTILINESTRING. Throws an error if output is a MULTIPOLYGON; use ST_BdMPolyFromText in that case, or see ST_BuildArea() for a postgis-specific approach.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

Performed by the GEOS module.

Availability: 1.1.0

See Also

ST_BuildArea, ST_BdMPolyFromText

7.8.1.2 ST_BdMPolyFromText

ST_BdMPolyFromText — Construct a MultiPolygon given an arbitrary collection of closed linestrings as a MultiLineString text representation Well-Known text representation.

Synopsis

geometry **ST_BdMPolyFromText**(text WKT, integer srid);

Description

Construct a Polygon given an arbitrary collection of closed linestrings, polygons, MultiLineStrings as Well-Known text representation.



Note
Throws an error if WKT is not a MULTILINESTRING. Forces MULTIPOLYGON output even when result is really only composed by a single POLYGON; use ST_BdPolyFromText if you're sure a single POLYGON will result from operation, or see ST_BuildArea() for a postgis-specific approach.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

Performed by the GEOS module.

Availability: 1.1.0

See Also

ST_BuildArea, ST_BdPolyFromText

7.8.1.3 ST_GeogFromText

ST_GeogFromText — Return a specified geography value from Well-Known Text representation or extended (WKT).

Synopsis

geography **ST_GeogFromText**(text EWKT);

Description

Returns a geography object from the well-known text or extended well-known representation. SRID 4326 is assumed if unspecified. This is an alias for ST_GeographyFromText. Points are always expressed in long lat form.

Examples

Add a WGS 84 geography point column to a table.

Code

```
ALTER TABLE sometable ADD COLUMN geog geography(POINT, 4326);
```

Populate the geography column from longitude and latitude columns.

Code

```
UPDATE sometable
SET geog = ST_GeogFromText('SRID=4326;POINT(' || lon || ' ' || lat || ')');
```

Specify a geography point using EPSG:4267, NAD27.

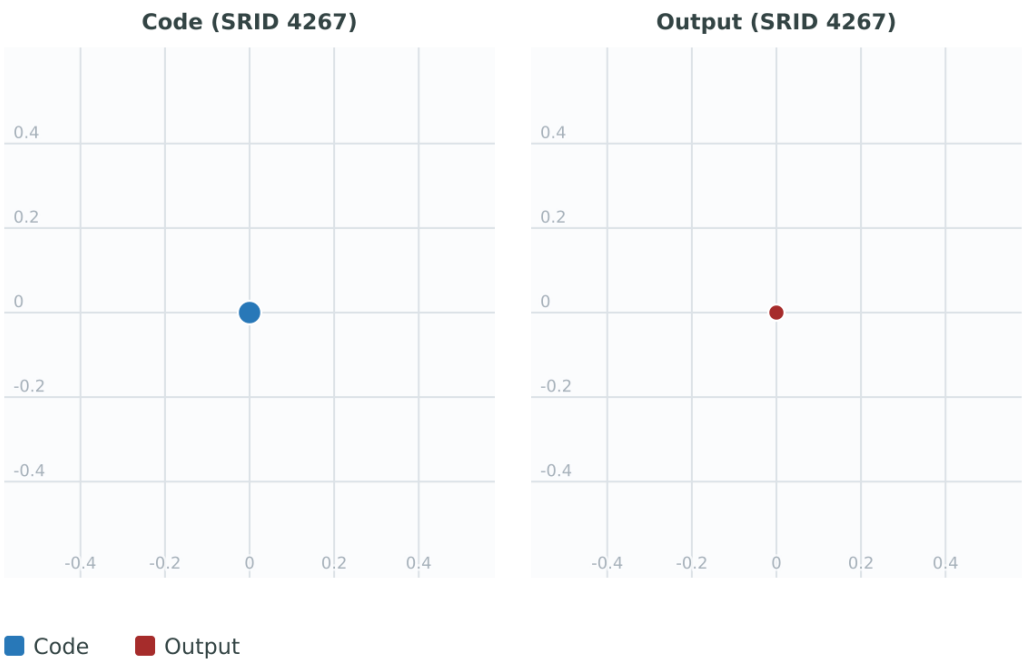
Code

```
SELECT ST_GeogFromText('SRID=4267;POINT(-77.0092 38.889588)');
```

Output

```
SRID=4267;POINT(-77.0092 38.889588)
```

Figure



See Also

ST_AsText, ST_GeographyFromText

7.8.1.4 ST_GeographyFromText

ST_GeographyFromText — Return a specified geography value from Well-Known Text representation or extended (WKT).

Synopsis

geography **ST_GeographyFromText**(text EWKT);

Description

Returns a geography object from the well-known text representation. SRID 4326 is assumed if unspecified.

See Also

ST_GeogFromText, ST_AsText

7.8.1.5 ST_GeomCollFromText

ST_GeomCollFromText — Makes a collection Geometry from collection WKT with the given SRID. If SRID is not given, it defaults to 0.

Synopsis

geometry **ST_GeomCollFromText**(text WKT, integer srid);
geometry **ST_GeomCollFromText**(text WKT);

Description

Makes a collection Geometry from the Well-Known-Text (WKT) representation with the given SRID. If SRID is not given, it defaults to 0.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

Returns null if the WKT is not a GEOMETRYCOLLECTION



Note
If you are absolutely sure all your WKT geometries are collections, don't use this function. It is slower than ST_GeomFromText since it adds an additional validation step.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2


This method implements the SQL/MM specification.

Examples

Code

```
SELECT ST_GeomCollFromText('GEOMETRYCOLLECTION(POINT(1 2), LINESTRING(1 2,3 4))');
```

See Also

ST_GeomFromText, ST_SRID

7.8.1.6 ST_GeomFromEWKT

ST_GeomFromEWKT — Return a specified ST_Geometry value from Extended Well-Known Text representation (EWKT).

Synopsis

geometry **ST_GeomFromEWKT**(text EWKT);

Description

Constructs a PostGIS ST_Geometry object from the OGC Extended Well-Known text (EWKT) representation.



Note
The EWKT format is not an OGC standard, but an PostGIS specific format that includes the spatial reference system (SRID) identifier

Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Create a LineString with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;LINESTRING(-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932)') AS geom;
```

Output

```
 geometry_type | srid
---------------+------
 ST_LineString | 4269
(1 row)
```

Create a MultiLineString with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;MULTILINESTRING((-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932))') AS geom;
```

Output

```
    geometry_type    | srid
---------------------+------
 ST_MultiLineString  | 4269
(1 row)
```

Create a Point with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;POINT(-71.064544 42.28787)') AS geom;
```

Output

```
 geometry_type | srid
---------------+------
 ST_Point      | 4269
(1 row)
```

Create a Polygon with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;POLYGON((-71.1776585052917 ↩
    42.3902909739571,-71.1776820268866 42.3903701743239,
-71.1776063012595 42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 ↩
    42.3902909739571))') AS geom;
```

Output

```
 geometry_type | srid
---------------+------
 ST_Polygon    | 4269
(1 row)
```

Create a MultiPolygon with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;MULTIPOLYGON(((-71.1031880899493 42.3152774590236,
-71.1031627617667 42.3152960829043,-71.102923838298 42.3149156848307,
-71.1023097974109 42.3151969047397,-71.1019285062273 42.3147384934248,
-71.102505233663 42.3144722937587,-71.10277487471 42.3141658254797,
-71.103113945163 42.3142739188902,-71.10324876416 42.31402489987,
-71.1033002961013 42.3140393340215,-71.1033488797549 42.3139495090772,
-71.103396240451 42.3138632439557,-71.1041521907712 42.3141153348029,
-71.1041411411543 42.3141545014533,-71.1041287795912 42.3142114839058,
-71.1041188134329 42.3142693656241,-71.1041112482575 42.3143272556118,
-71.1041072845732 42.3143851580048,-71.1041057218871 42.3144430686681,
-71.1041065602059 42.3145009876017,-71.1041097995362 42.3145589148055,
-71.1041166403905 42.3146168544148,-71.1041258822717 42.3146748022936,
-71.1041375307579 42.3147318674446,-71.1041492906949 42.3147711126569,
-71.1041598612795 42.314808571739,-71.1042515013869 42.3151287620809,
-71.1041173835118 42.3150739481917,-71.1040809891419 42.3151344119048,
-71.1040438678912 42.3151191367447,-71.1040194562988 42.3151832057859,
-71.1038734225584 42.3151140942995,-71.1038446938243 42.3151006300338,
-71.1038315271889 42.315094347535,-71.1037393329282 42.315054824985,
-71.1035447555574 42.3152608696313,-71.1033436658644 42.3151648370544,
-71.1032580383161 42.3152269126061,-71.103223066939 42.3152517403219,
-71.1031880899493 42.3152774590236)),
((-71.1043632495873 42.315113108546,-71.1043583974082 42.3151211109857,
-71.1043443253471 42.3150676015829,-71.1043850704575 42.3150793250568,-71.1043632495873 ↩
    42.315113108546)))') AS geom;
```

Output

```
   geometry_type   | srid
-------------------+------
 ST_MultiPolygon   | 4269
(1 row)
```

Create a 3D CircularString without an SRID.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)') AS ↩
    geom;
```

Output

```
   geometry_type    | srid
--------------------+------
 ST_CircularString  | 0
(1 row)
```

Create a PolyhedralSurface without an SRID.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('POLYHEDRALSURFACE(
    ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
    ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
    ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)') AS geom;
```

Output

```
      geometry_type       | srid
--------------------------+------
 ST_PolyhedralSurface     | 0
(1 row)
```

See Also

ST_AsEWKT, ST_GeomFromText

7.8.1.7 ST_GeomFromMARC21

ST_GeomFromMARC21 — Takes MARC21/XML geographic data as input and returns a PostGIS geometry object.

Synopsis

geometry **ST_GeomFromMARC21** (text marcxml);

Description

This function creates a PostGIS geometry from a MARC21/XML record, which can contain a `POINT` or a `POLYGON`. In case of multiple geographic data entries in the same MARC21/XML record, a `MULTIPOINT` or `MULTIPOLYGON` will be returned. If the record contains mixed geometry types, a `GEOMETRYCOLLECTION` will be returned. It returns NULL if the MARC21/XML record does not contain any geographic data (datafield:034).

LOC MARC21/XML versions supported:

- MARC21/XML 1.1

Availability: 3.3.0, requires libxml2 2.6+



Note
The MARC21/XML Coded Cartographic Mathematical Data currently does not provide any means to describe the Spatial Reference System of the encoded coordinates, so this function will always return a geometry with `SRID 0`.



Note

Returned `POLYGON` geometries will always be clockwise oriented.

Examples

Converting MARC21/XML geographic data containing a single `POINT` encoded as `hddd.dddddd`

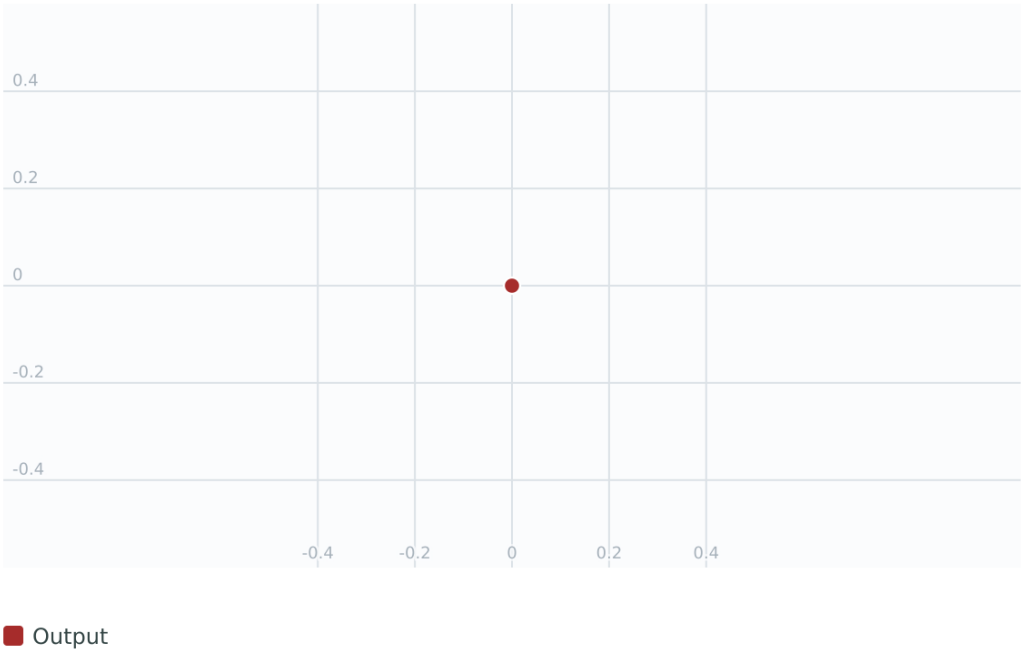
Code

```
SELECT ST_GeomFromMARC21(
'<record xmlns="http://www.loc.gov/MARC21/slim">
    <leader>00000nz a2200000nc 4500</leader>
    <controlfield tag="001">040277569</controlfield>
    <datafield tag="034" ind1=" " ind2=" ">
        <subfield code="d">W004.500000</subfield>
        <subfield code="e">W004.500000</subfield>
        <subfield code="f">N054.250000</subfield>
        <subfield code="g">N054.250000</subfield>
    </datafield>
</record>'
);
```

Output

```
POINT(-4.5 54.25)
```

Figure



Converting MARC21/XML geographic data containing a single `POLYGON` encoded as `hdddmmss`

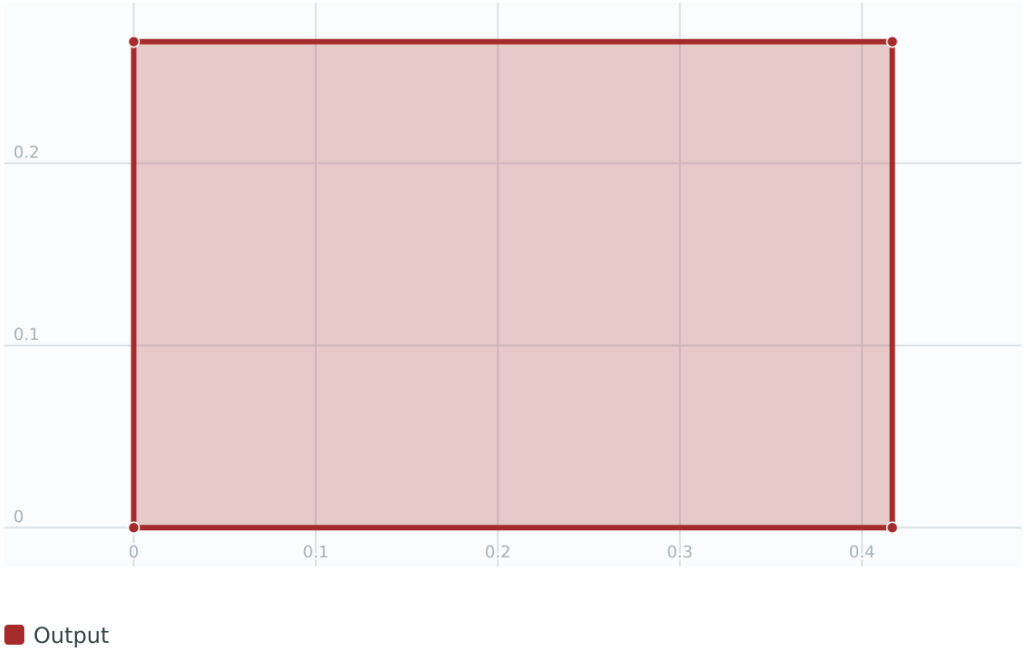
Code

```
SELECT ST_GeomFromMARC21(
'<record xmlns="http://www.loc.gov/MARC21/slim">
    <leader>01062cem a2200241 a 4500</leader>
    <controlfield tag="001">   84696781 </controlfield>
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a">a</subfield>
        <subfield code="b">50000</subfield>
        <subfield code="d">E0130600</subfield>
        <subfield code="e">E0133100</subfield>
        <subfield code="f">N0523900</subfield>
        <subfield code="g">N0522300</subfield>
    </datafield>
</record>'
);
```

Output

```
POLYGON((13.1 52.65,13.516666666666667 52.65,13.516666666666667 52.38333333333333,13.1 52.38333333333333,13.1 52.65))
```

Figure



Converting MARC21/XML geographic data containing a `POLYGON` and a `POINT`:

Code

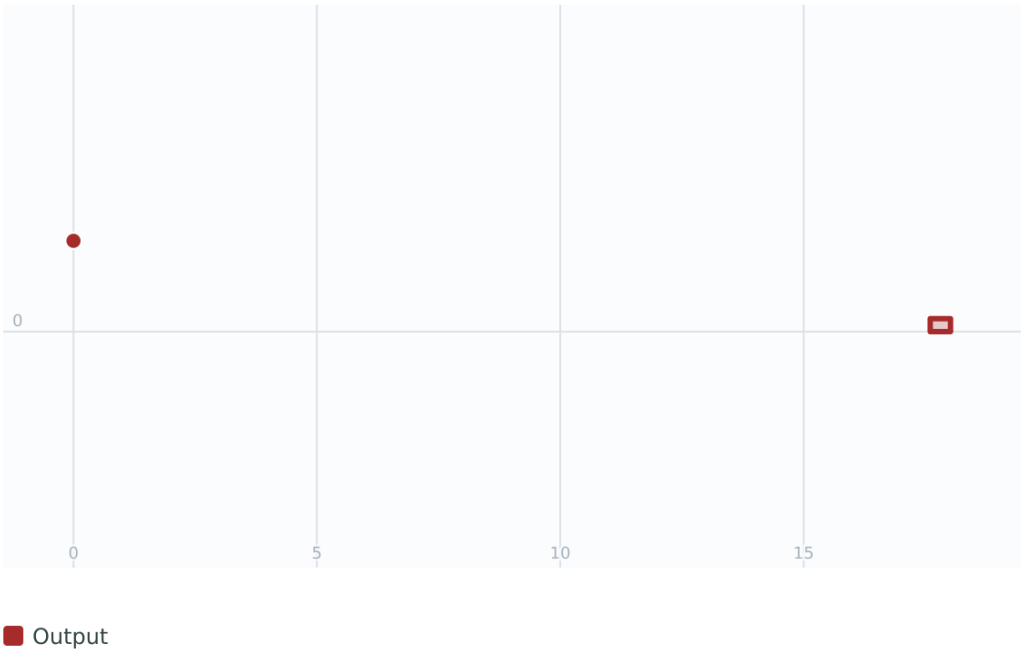
```
SELECT ST_GeomFromMARC21(
'<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a">a</subfield>
        <subfield code="b">50000</subfield>
        <subfield code="d">E0130600</subfield>
        <subfield code="e">E0133100</subfield>
        <subfield code="f">N0523900</subfield>
        <subfield code="g">N0522300</subfield>
    </datafield>
    <datafield tag="034" ind1=" " ind2=" ">
```

```
        <subfield code="d">W004.500000</subfield>
        <subfield code="e">W004.500000</subfield>
        <subfield code="f">N054.250000</subfield>
        <subfield code="g">N054.250000</subfield>
    </datafield>
</record>'
);
```

Output

```
GEOMETRYCOLLECTION(POLYGON((13.1 52.65,13.516666666666667 52.65,13.516666666666667 52.38333333333333,13.1 ↩
    52.38333333333333,13.1 52.65)), POINT(-4.5 54.25))
```

Figure



See Also

ST_AsMARC21

7.8.1.8 ST_GeometryFromText

ST_GeometryFromText — Returns a geometry from Well-Known Text (WKT). Alias for ST_GeomFromText.

Synopsis

geometry **ST_GeometryFromText**(text WKT);
geometry **ST_GeometryFromText**(text WKT, integer srid);

Description

This function is an alias for ST_GeomFromText and provides the same signatures.

Availability: 1.2.2

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.40

See Also

ST_GeomFromText

7.8.1.9 ST_GeomFromText

ST_GeomFromText — Return a specified ST_Geometry value from Well-Known Text representation (WKT).

Synopsis

geometry **ST_GeomFromText**(text WKT);
geometry **ST_GeomFromText**(text WKT, integer srid);

Description

Constructs a PostGIS ST_Geometry object from the OGC Well-Known text representation.



Note
There are two variants of ST_GeomFromText function. The first takes no SRID and returns a geometry with no defined spatial reference system (SRID=0). The second takes a SRID as the second argument and returns a geometry that includes this SRID as part of its metadata.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2 - option SRID is from the conformance suite.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.40

This method supports Circular Strings and Curves.



Note
While not OGC-compliant, ST_MakePoint is faster than ST_GeomFromText and ST_PointFromText. It is also easier to use for numeric coordinate values. ST_Point is another option similar in speed to ST_MakePoint and is OGC-compliant, but doesn't support anything but 2D points.

Changed: 2.0.0 In prior versions of PostGIS ST_GeomFromText('GEOMETRYCOLLECTION(EMPTY)') was allowed. This is now illegal in PostGIS 2.0.0 to better conform with SQL/MM standards. This should now be written as ST_GeomFromText('GEOMETR EMPTY')

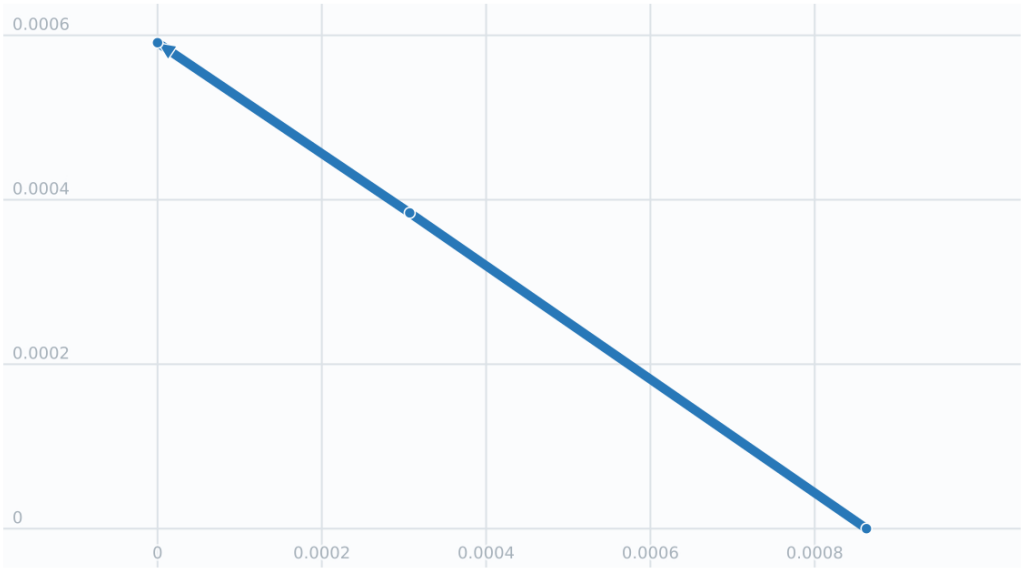
Examples

Create a LineString with SRID 0.

Code

```
SELECT ST_SRID(ST_GeomFromText('LINESTRING(-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932)'));
```

Figure

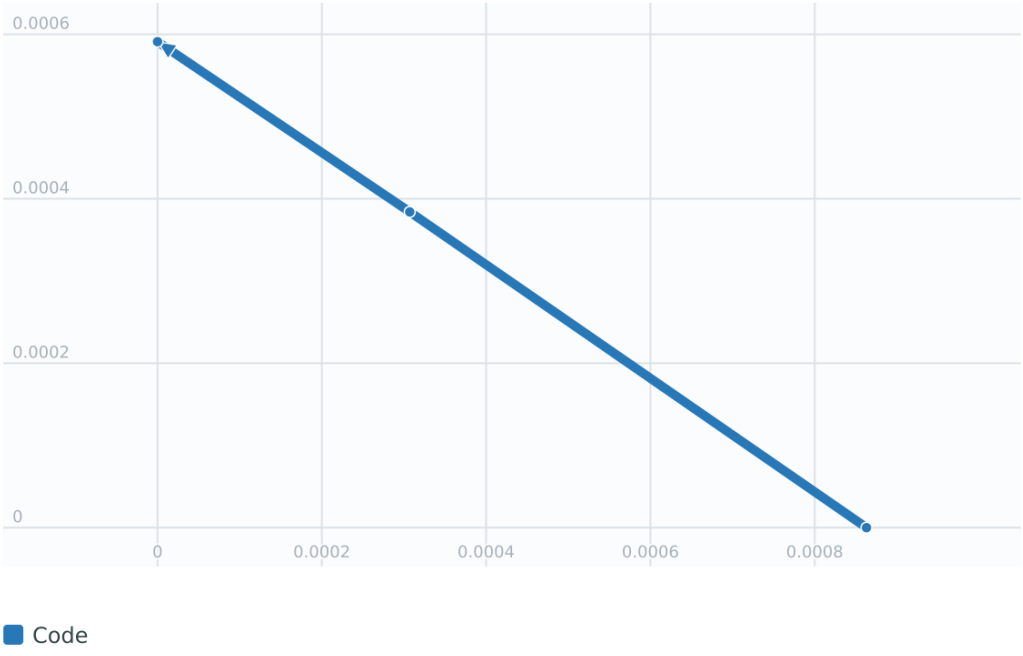


Create the same LineString with SRID 4269.

Code

```
SELECT ST_SRID(ST_GeomFromText('LINESTRING(-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932)', 4269));
```

Figure

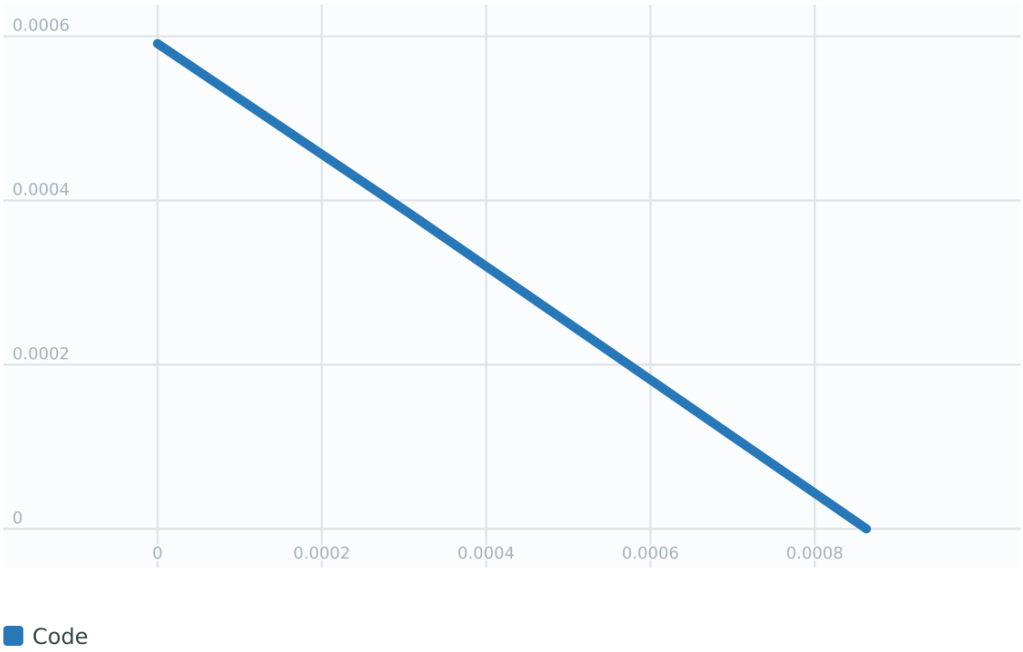


Create a MultiLineString.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('MULTILINESTRING((-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932))'));
```

Figure



Create a Point.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('POINT(-71.064544 42.28787)'));
```

Output

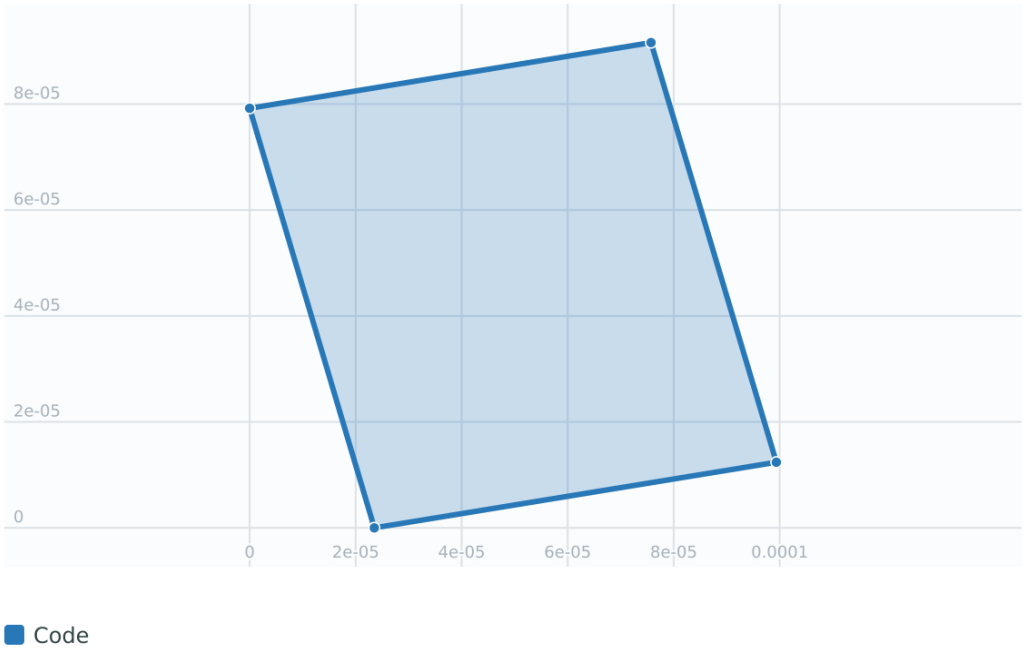
```
ST_Point
```

Create a Polygon.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('POLYGON((-71.1776585052917 ↩
    42.3902909739571,-71.1776820268866 42.3903701743239,
-71.1776063012595 42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 ↩
    42.3902909739571))'));
```

Figure

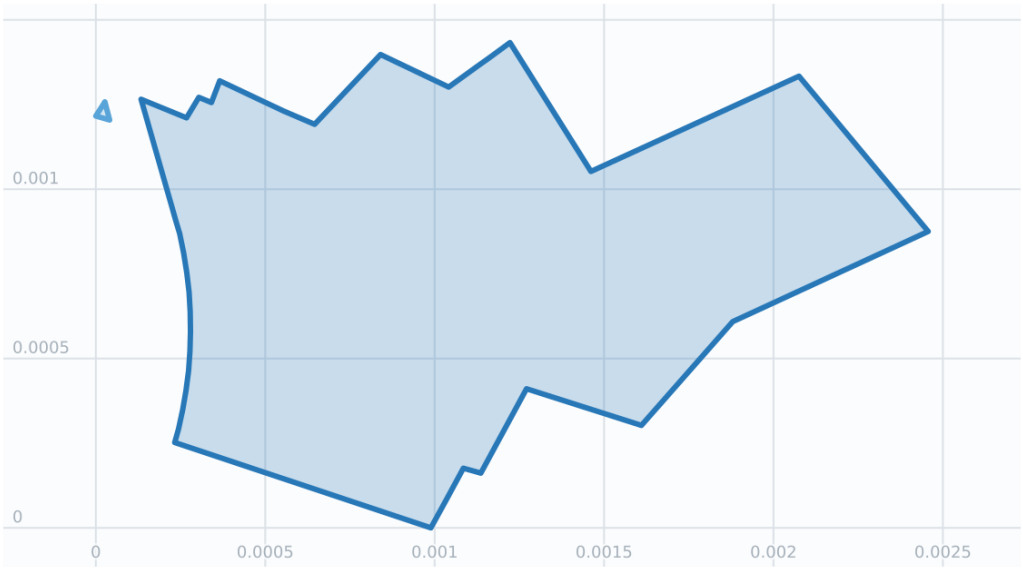


Create a MultiPolygon with SRID 4326.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('MULTIPOLYGON(((-71.1031880899493 42.3152774590236,
-71.1031627617667 42.3152960829043,-71.102923838298 42.3149156848307,
-71.1023097974109 42.3151969047397,-71.1019285062273 42.3147384934248,
-71.102505233663 42.3144722937587,-71.10277487471 42.3141658254797,
-71.103113945163 42.3142739188902,-71.10324876416 42.31402489987,
-71.1033002961013 42.3140393340215,-71.1033488797549 42.3139495090772,
-71.103396240451 42.3138632439557,-71.1041521907712 42.3141153348029,
-71.1041411411543 42.3141545014533,-71.1041287795912 42.3142114839058,
-71.1041188134329 42.3142693656241,-71.1041112482575 42.3143272556118,
-71.1041072845732 42.3143851580048,-71.1041057218871 42.3144430686681,
-71.1041065602059 42.3145009876017,-71.1041097995362 42.3145589148055,
-71.1041166403905 42.3146168544148,-71.1041258822717 42.3146748022936,
-71.1041375307579 42.3147318674446,-71.1041492906949 42.3147711126569,
-71.1041598612795 42.314808571739,-71.1042515013869 42.3151287620809,
-71.1041173835118 42.3150739481917,-71.1040809891419 42.3151344119048,
-71.1040438678912 42.3151191367447,-71.1040194562988 42.3151832057859,
-71.1038734225584 42.3151140942995,-71.1038446938243 42.3151006300338,
-71.1038315271889 42.315094347535,-71.1037393329282 42.315054824985,
-71.1035447555574 42.3152608696313,-71.1033436658644 42.3151648370544,
-71.1032580383161 42.3152269126061,-71.103223066939 42.3152517403219,
-71.1031880899493 42.3152774590236)),
((-71.1043632495873 42.315113108546,-71.1043583974082 42.3151211109857,
-71.1043443253471 42.3150676015829,-71.1043850704575 42.3150793250568,-71.1043632495873 ↩
    42.315113108546)))', 4326));
```

Figure

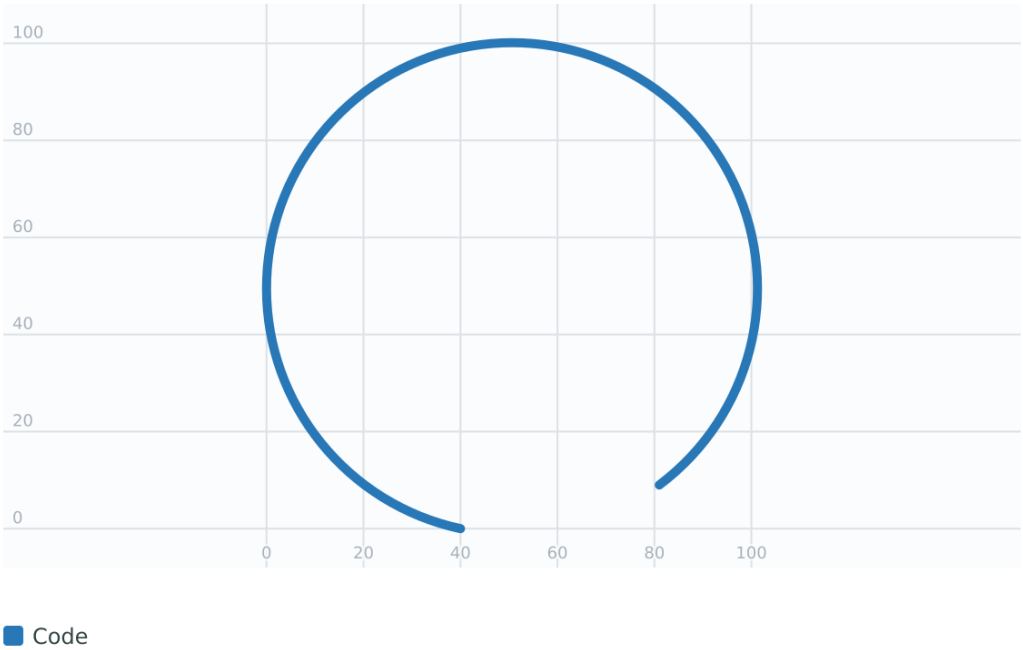


Create a CircularString.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('CIRCULARSTRING(220268 150415,220227 150505,220227 ↩
    150406)'));
```

Figure



See Also

ST_GeomFromEWKT, ST_GeomFromWKB, ST_SRID

7.8.1.10 ST_LineFromText

ST_LineFromText — Makes a Geometry from WKT representation with the given SRID. If SRID is not given, it defaults to 0.

Synopsis

geometry **ST_LineFromText**(text WKT);
geometry **ST_LineFromText**(text WKT, integer srid);

Description

Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0. If WKT passed in is not a LINESTRING, then null is returned.



Note
OGC SPEC 3.2.6.2 - optional SRID is from the conformance suite.



Note
If you know all your geometries are LINESTRINGS, its more efficient to just use ST_GeomFromText. This just calls ST_GeomFromText and adds additional validation that it returns a linestring.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2



This method implements the SQL/MM specification. SQL-MM 3: 7.2.8

Examples

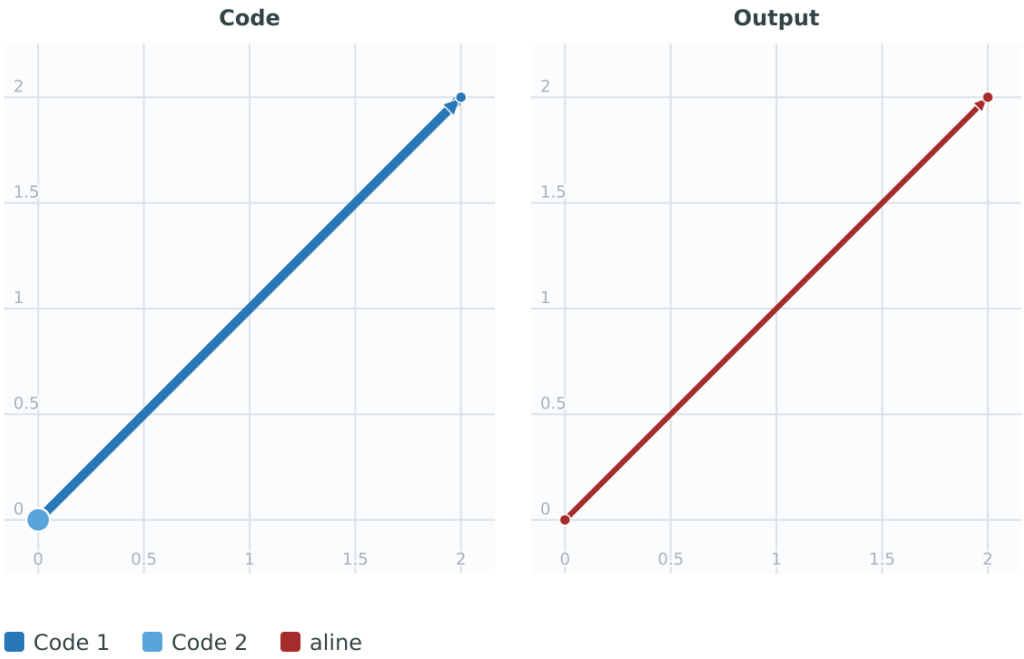
Code

```
SELECT ST_LineFromText('LINESTRING(1 2,3 4)') AS aline,
ST_LineFromText('POINT(1 2)') IS NULL AS null_return;
```

Output

```
aline                | null_return
-------------------------------------
LINESTRING(1 2,3 4)  | t
```

Figure



See Also

ST_GeomFromText

7.8.1.11 ST_MLineFromText

ST_MLineFromText — Return a specified ST_MultiLineString value from WKT representation.

Synopsis

geometry **ST_MLineFromText**(text WKT, integer srid);
geometry **ST_MLineFromText**(text WKT);

Description

Makes a Geometry from Well-Known-Text (WKT) with the given SRID. If SRID is not given, it defaults to 0.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

Returns null if the WKT is not a MULTILINESTRING



Note
If you are absolutely sure all your WKT geometries are points, don't use this function. It is slower than ST_GeomFromText since it adds an additional validation step.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2


This method implements the SQL/MM specification. SQL-MM 3: 9.4.4

Examples

Code

```
SELECT ST_MLineFromText('MULTILINESTRING((1 2,3 4),(4 5,6 7))');
```

See Also

ST_GeomFromText

7.8.1.12 ST_MPointFromText

ST_MPointFromText — Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

Synopsis

geometry **ST_MPointFromText**(text WKT, integer srid);
geometry **ST_MPointFromText**(text WKT);

Description

Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

Returns null if the WKT is not a MULTIPOINT



Note
If you are absolutely sure all your WKT geometries are points, don't use this function. It is slower than ST_GeomFromText since it adds an additional validation step.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. 3.2.6.2


This method implements the SQL/MM specification. SQL-MM 3: 9.2.4

Examples

Code

```
SELECT ST_MPointFromText('MULTIPOINT((1 2),(3 4))');
SELECT ST_MPointFromText('MULTIPOINT((-70.9590 42.1180),(-70.9611 42.1223))', 4326);
```

See Also

ST_GeomFromText

7.8.1.13 ST_MPolyFromText

ST_MPolyFromText — Makes a MultiPolygon Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

Synopsis

geometry **ST_MPolyFromText**(text WKT, integer srid);
geometry **ST_MPolyFromText**(text WKT);

Description

Makes a MultiPolygon from WKT with the given SRID. If SRID is not given, it defaults to 0.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite

Throws an error if the WKT is not a MULTIPOLYGON



Note
If you are absolutely sure all your WKT geometries are multipolygons, don't use this function. It is slower than ST_GeomFromText since it adds an additional validation step.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 9.6.4

Examples

Create a 3D MultiPolygon with an interior ring and SRID 0.

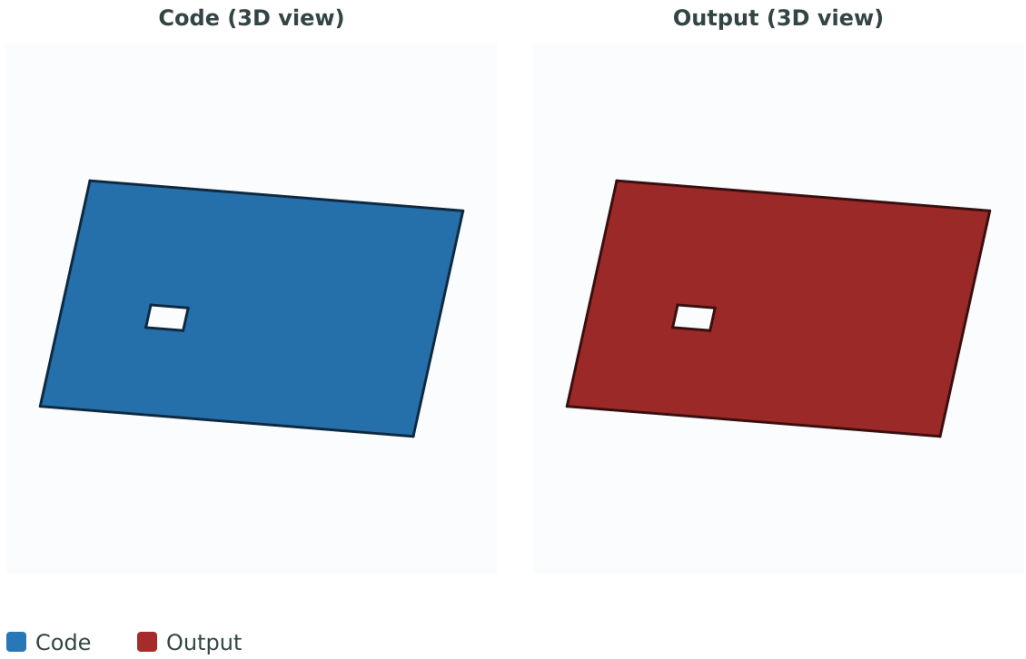
Code

```
SELECT ST_MPolyFromText(
    'MULTIPOLYGON(((0 0 1,20 0 1,20 20 1,0 20 1,0 0 1),
                   (5 5 3,5 7 3,7 7 3,7 5 3,5 5 3)))'
);
```

Output

```
MULTIPOLYGON(((0 0 1,20 0 1,20 20 1,0 20 1,0 0 1),(5 5 3,5 7 3,7 7 3,7 5 3,5 5 3)))
```

Figure

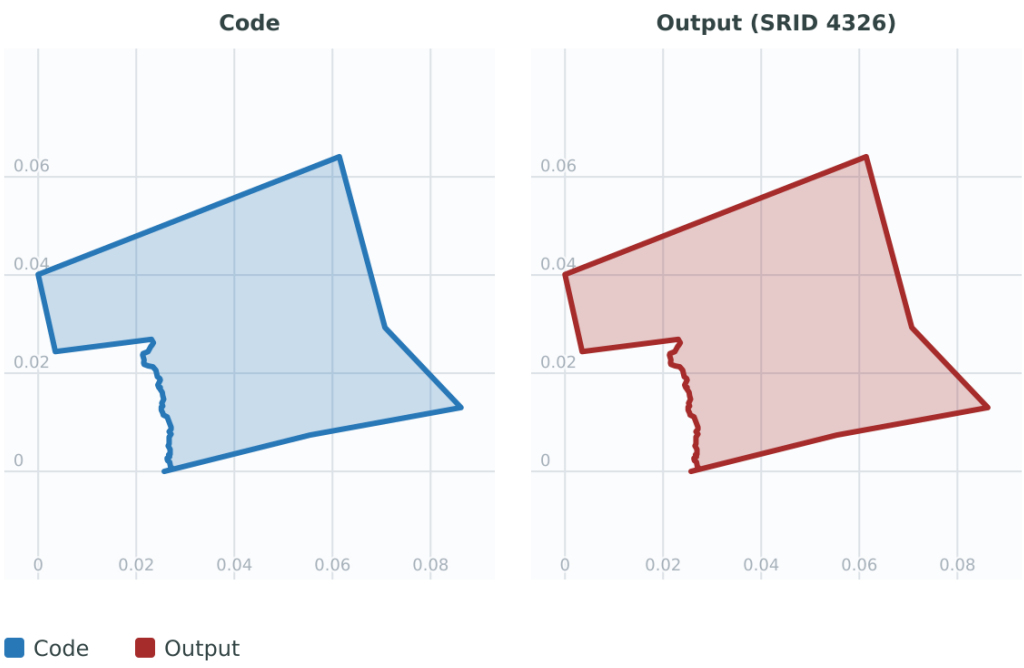


Create a MultiPolygon with SRID 4326.

Code

```
SELECT ST_MPolyFromText(
    'MULTIPOLYGON(((-70.916 42.1002,-70.9468 42.0946,-70.9765 42.0872,
      -70.9754 42.0875,-70.9749 42.0879,-70.9752 42.0881,-70.9754 42.0891,
      -70.9758 42.0894,-70.9759 42.0897,-70.9759 42.0899,-70.9754 42.0902,
      -70.9756 42.0906,-70.9753 42.0907,-70.9753 42.0917,-70.9757 42.0924,
      -70.9755 42.0928,-70.9755 42.0942,-70.9751 42.0948,-70.9755 42.0953,
      -70.9751 42.0958,-70.9751 42.0962,-70.9759 42.0983,-70.9767 42.0987,
      -70.9768 42.0991,-70.9771 42.0997,-70.9771 42.1003,-70.9768 42.1005,
      -70.977 42.1011,-70.9766 42.1019,-70.9768 42.1026,-70.9769 42.1033,
      -70.9775 42.1042,-70.9773 42.1043,-70.9776 42.1043,-70.9778 42.1048,
      -70.9773 42.1058,-70.9774 42.1061,-70.9779 42.1065,-70.9782 42.1078,
      -70.9788 42.1085,-70.9798 42.1087,-70.9806 42.109,-70.9807 42.1093,
      -70.9806 42.1099,-70.9809 42.1109,-70.9808 42.1112,-70.9798 42.1116,
      -70.9792 42.1127,-70.979 42.1129,-70.9787 42.1134,-70.979 42.1139,
      -70.9791 42.1141,-70.9987 42.1116,-71.0022 42.1273,-70.9408 42.1513,
      -70.9315 42.1165,-70.916 42.1002)))',
    4326
);
```

Figure



See Also

ST_GeomFromText, ST_SRID

7.8.1.14 ST_PointFromText

ST_PointFromText — Makes a point Geometry from WKT with the given SRID. If SRID is not given, it defaults to unknown.

Synopsis

geometry **ST_PointFromText**(text WKT);
geometry **ST_PointFromText**(text WKT, integer srid);

Description

Constructs a PostGIS ST_Geometry point object from the OGC Well-Known text representation. If SRID is not given, it defaults to unknown (currently 0). If geometry is not a WKT point representation, returns null. If completely invalid WKT, then throws an error.



Note
There are 2 variants of ST_PointFromText function, the first takes no SRID and returns a geometry with no defined spatial reference system. The second takes a spatial reference id as the second argument and returns an ST_Geometry that includes this srid as part of its meta-data. The srid must be defined in the spatial_ref_sys table.



Note
If you are absolutely sure all your WKT geometries are points, don't use this function. It is slower than ST_GeomFromText since it adds an additional validation step. If you are building points from long lat coordinates and care more about performance and accuracy than OGC compliance, use ST_MakePoint or OGC compliant alias ST_Point.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2 - option SRID is from the conformance suite.

This method implements the SQL/MM specification. SQL-MM 3: 6.1.8

Examples

Create a point with the default SRID 0.

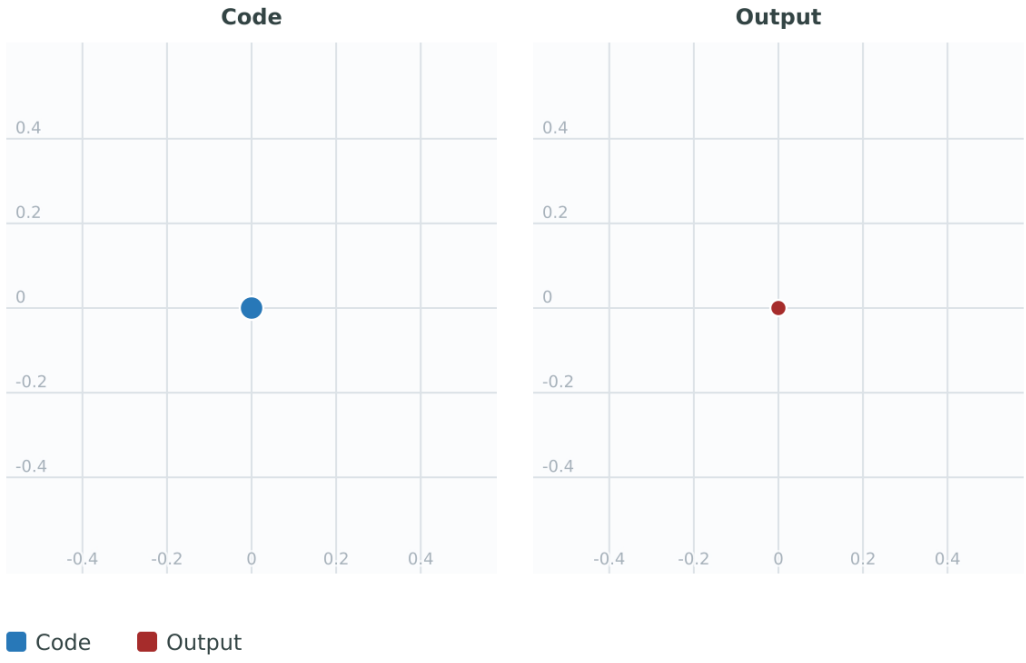
Code

```
SELECT ST_PointFromText('POINT(-71.064544 42.28787)');
```

Output

```
POINT(-71.064544 42.28787)
```

Figure



Create a point with SRID 4326.

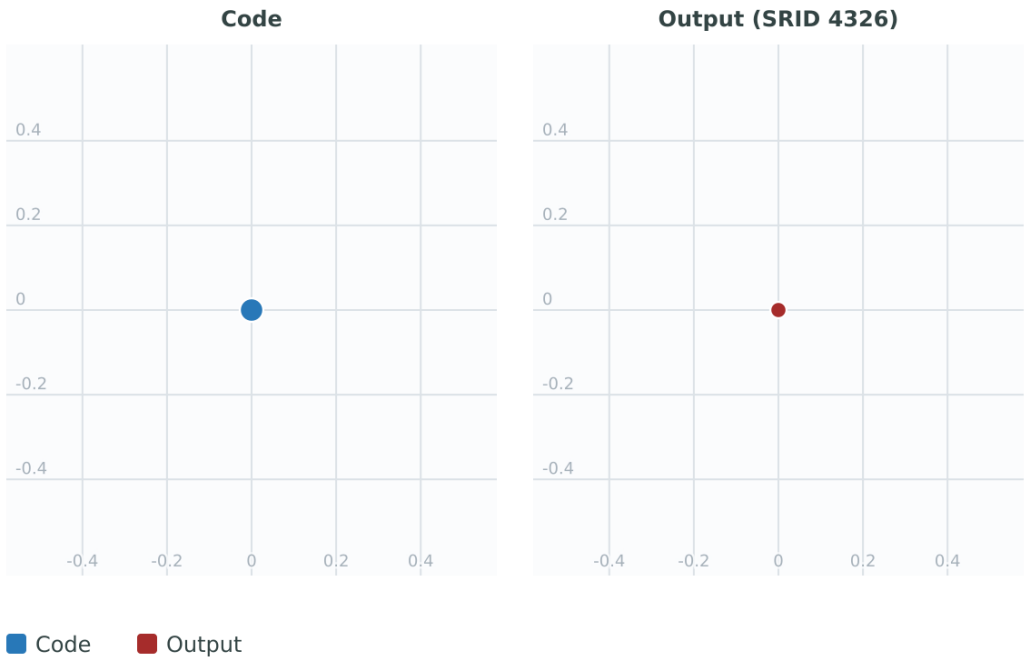
Code

```
SELECT ST_PointFromText('POINT(-71.064544 42.28787)', 4326);
```

Output

```
SRID=4326;POINT(-71.064544 42.28787)
```

Figure



See Also

ST_GeomFromText, ST_MakePoint, ST_Point, ST_SRID

7.8.1.15 ST_PolygonFromText

ST_PolygonFromText — Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

Synopsis

geometry **ST_PolygonFromText**(text WKT);
geometry **ST_PolygonFromText**(text WKT, integer srid);

Description

Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0. Returns null if WKT is not a polygon.

OGC SPEC 3.2.6.2 - option SRID is from the conformance suite



Note
If you are absolutely sure all your WKT geometries are polygons, don't use this function. It is slower than ST_GeomFromText since it adds an additional validation step.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 8.3.6

Examples

The result is returned as a native geometry. The documentation test runner compares that value directly with the readable EWKT shown below and uses the same geometry for the generated figure.

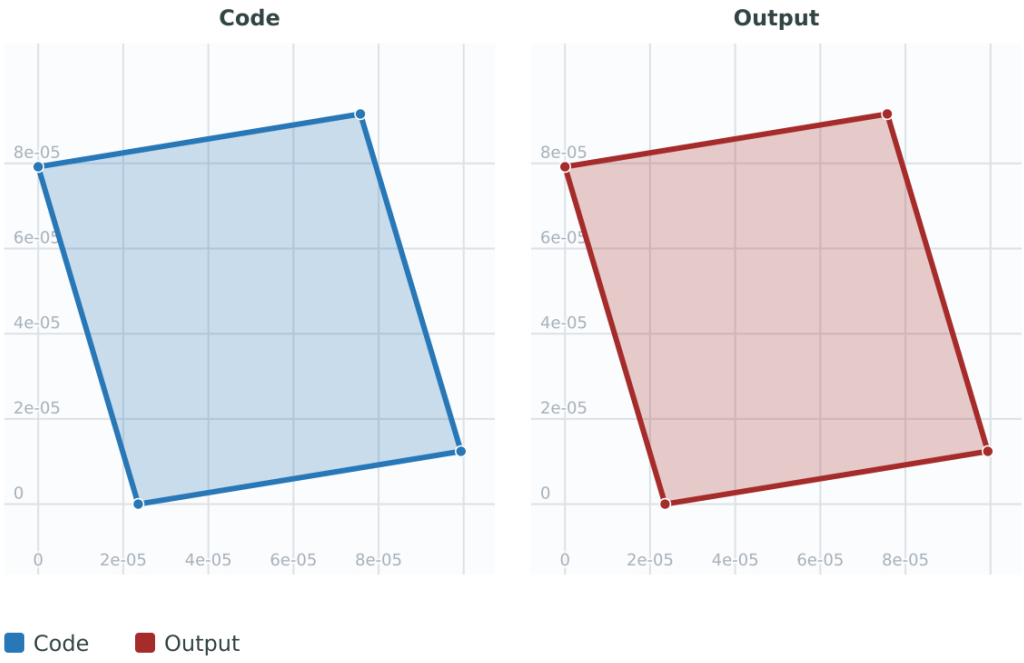
Code

```
SELECT ST_PolygonFromText('POLYGON((-71.1776585052917 42.3902909739571,-71.1776820268866 ↩
    42.3903701743239,
-71.1776063012595 42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 ↩
    42.3902909739571))');
```

Output

```
POLYGON((-71.1776585052917 42.3902909739571,-71.1776820268866 42.3903701743239,-71.1776063012595 ↩
    42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 42.3902909739571))
```

Figure



Code

```
SELECT ST_PolygonFromText('POINT(1 2)') IS NULL AS point_is_notpoly;
```

Output

```
t
```

See Also

ST_GeomFromText

7.8.1.16 ST_WKTToSQL

ST_WKTToSQL — Returns a geometry from Well-Known Text (WKT). SQL/MM alias for ST_GeomFromText.

Synopsis

geometry **ST_WKTToSQL**(text WKT);

Description

This function is the SQL/MM alias for ST_GeomFromText with no SRID argument.

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.34

See Also

ST_GeomFromText

7.8.2 Well-Known Binary (WKB)

7.8.2.1 ST_GeogFromWKB

ST_GeogFromWKB — Creates a geography instance from a Well-Known Binary geometry representation (WKB) or extended Well Known Binary (EWKB).

Synopsis

geography **ST_GeogFromWKB**(bytea wkb);

Description

The `ST_GeogFromWKB` function, takes a well-known binary representation (WKB) of a geometry or PostGIS Extended WKB and creates an instance of the appropriate geography type. This function plays the role of the Geometry Factory in SQL.

If SRID is not specified, it defaults to 4326 (WGS 84 long lat).

 This method supports Circular Strings and Curves.

Examples

Although bytea rep contains single \, these need to be escaped when inserting into a table.

Code

```
SELECT ST_GeogFromWKB(E'\\001\\002\\000\\000\\000\\002\\000\\000\\000\\037\\205\\353Q ↩
    \\270~\\\\\\300\\323Mb\\020X\\231C@\\020X9\\264\\310~\\\\\\300)\\\\\\217\\302\\365\\230 ↩
    C@');
```

Output

```
LINESTRING(-113.98 39.198,-113.981 39.195)
```

Figure



See Also

ST_GeogFromText, ST_AsBinary

7.8.2.2 ST_GeomFromEWKB

ST_GeomFromEWKB — Return a specified ST_Geometry value from Extended Well-Known Binary representation (EWKB).

Synopsis

geometry **ST_GeomFromEWKB**(bytea EWKB);

Description

Constructs a PostGIS geometry from an Extended Well-Known Binary (EWKB) representation.



Note
The EWKB format is not an OGC standard, but a PostGIS specific format that includes the spatial reference system (SRID) identifier

Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

This example constructs `LINESTRING(-71.160281 42.258729,-71.160837 42.259113,-71.161144 42.25932)` in NAD83 longitude/latitude (SRID 4269) from its EWKB representation.

PostgreSQL escape string syntax (`E'...'`) always interprets backslash escapes. Each backslash therefore has to be doubled so that the bytea input parser receives the octal escapes.

Code

```
SELECT ST_GeomFromEWKB((
    E'\\001\\002\\000\\000\\040\\255\\020\\000\\000\\003\\000\\000\\000\\344\\112\\075' ||
    E'\\013\\102\\312\\121\\300\\156\\303\\050\\010\\036\\041\\105\\100\\047\\277\\105' ||
    E'\\047\\113\\312\\121\\300\\366\\173\\142\\235\\052\\041\\105\\100\\225\\174\\354' ||
    E'\\056\\120\\312\\121\\300\\160\\231\\323\\145\\061\\041\\105\\100'
)::bytea);
```

Output

```
0102000020 ↩
    AD10000003000000E44A3D0B42CA51C06EC328081E21454027BF45274BCA51C0F67B629D2A214540957CEC2E50CA51C07099D36531214540
```

Figure



With `standard_conforming_strings` enabled (the default), ordinary string literals preserve backslashes. The same EWKB can therefore be written with a single backslash before each octal byte value.

Code

```
SELECT ST_GeomFromEWKB((
    '\001\002\000\000\040\255\020\000\000\003\000\000\000\344\112\075' ||
    '\013\102\312\121\300\156\303\050\010\036\041\105\100\047\277\105' ||
    '\047\113\312\121\300\366\173\142\235\052\041\105\100\225\174\354' ||
    '\056\120\312\121\300\160\231\323\145\061\041\105\100'
)::bytea);
```

Output

```
0102000020 ↩
    AD10000003000000E44A3D0B42CA51C06EC328081E21454027BF45274BCA51C0F67B629D2A214540957CEC2E50CA51C0
```

See Also

ST_AsBinary, ST_AsEWKB, ST_GeomFromWKB

7.8.2.3 ST_GeomFromWKB

ST_GeomFromWKB — Creates a geometry instance from a Well-Known Binary geometry representation (WKB) and optional SRID.

Synopsis

geometry **ST_GeomFromWKB**(bytea geom);
geometry **ST_GeomFromWKB**(bytea geom, integer srid);

Description

The `ST_GeomFromWKB` function, takes a well-known binary representation of a geometry and a Spatial Reference System ID (`SRID`) and creates an instance of the appropriate geometry type. This function plays the role of the Geometry Factory in SQL. This is an alternate name for ST_WKBToSQL.

If SRID is not specified, it defaults to 0 (Unknown).

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.7.2 - the optional SRID is from the conformance suite

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.41

 This method supports Circular Strings and Curves.

Examples

Although bytea rep contains single \, these need to be escaped when inserting into a table. Unless standard_conforming_strings is set to on.

Code

```
SELECT ST_GeomFromWKB(E'\\001\\002\\000\\000\\000\\002\\000\\000\\000\\037\\205\\353Q ↩
    \\270~\\\\\\300\\323Mb\\020X\\231C@\\020X9\\264\\310~\\\\\\300)\\\\\\217\\302\\365\\230 ↩
    C@', 4326);
```

Output

```
SRID=4326;LINESTRING(-113.98 39.198,-113.981 39.195)
```

Figure



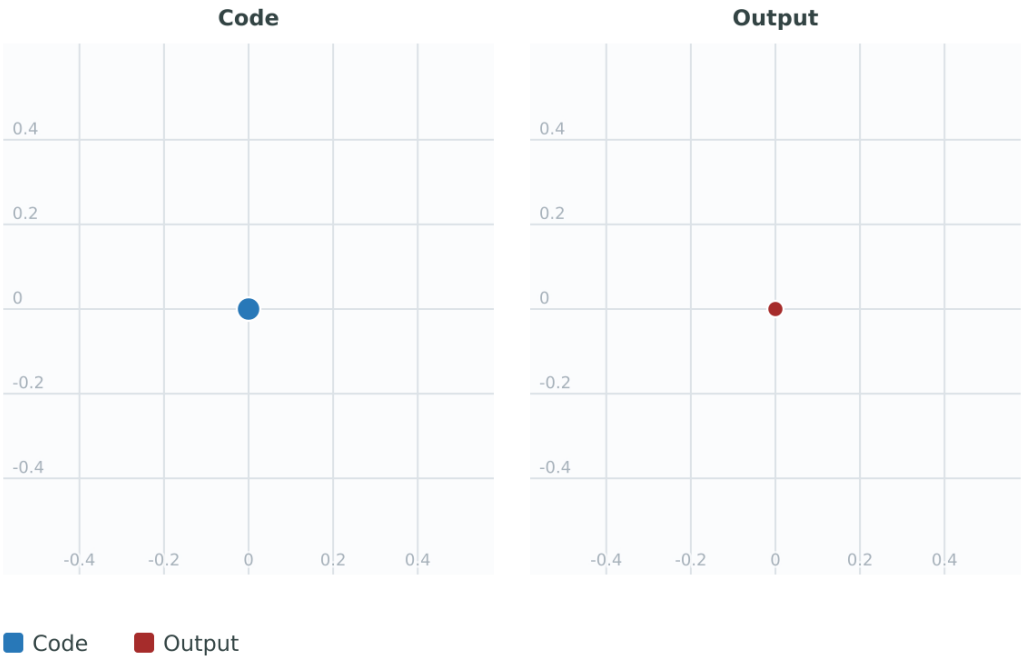
Code

```
SELECT ST_GeomFromWKB(ST_AsEWKB('POINT(2 5)'::geometry)
  );
```

Output

```
POINT(2 5)
```

Figure



The PostgreSQL decode function converts a hexadecimal WKB string to the bytea value accepted by ST_GeomFromWKB.

Code

```
SELECT ST_AsEWKT(ST_GeomFromWKB(
    decode('0101000000e5d022dbf93e2e40dbf97e6abc743540', 'hex'),
    4326
)) AS geom;
```

Output

```
SRID=4326;POINT(15.123 21.456)
```

See Also

ST_WKBToSQL, ST_AsBinary, ST_AsEWKT, ST_GeomFromEWKB

7.8.2.4 ST_LineFromWKB

ST_LineFromWKB — Creates a LINESTRING from a WKB representation, optionally using a given SRID.

Synopsis

geometry **ST_LineFromWKB**(bytea WKB);
geometry **ST_LineFromWKB**(bytea WKB, integer srid);

Description

The ST_LineFromWKB function takes a Well-Known Binary (WKB) representation of a geometry and an optional Spatial Reference System ID (SRID), and creates a LINESTRING geometry. It is a type-specific geometry factory in SQL.

If an SRID is not specified, it defaults to 0. NULL is returned if the input bytea does not represent a LINESTRING geometry.



Note
OGC SPEC 3.2.6.2 - optional SRID is from the conformance suite.



Note
If you know all your geometries are LINESTRINGs, it is more efficient to use ST_GeomFromWKB directly. This function calls ST_GeomFromWKB and adds validation that the result is a LINESTRING.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2



This method implements the SQL/MM specification. SQL-MM 3: 7.2.9

Examples

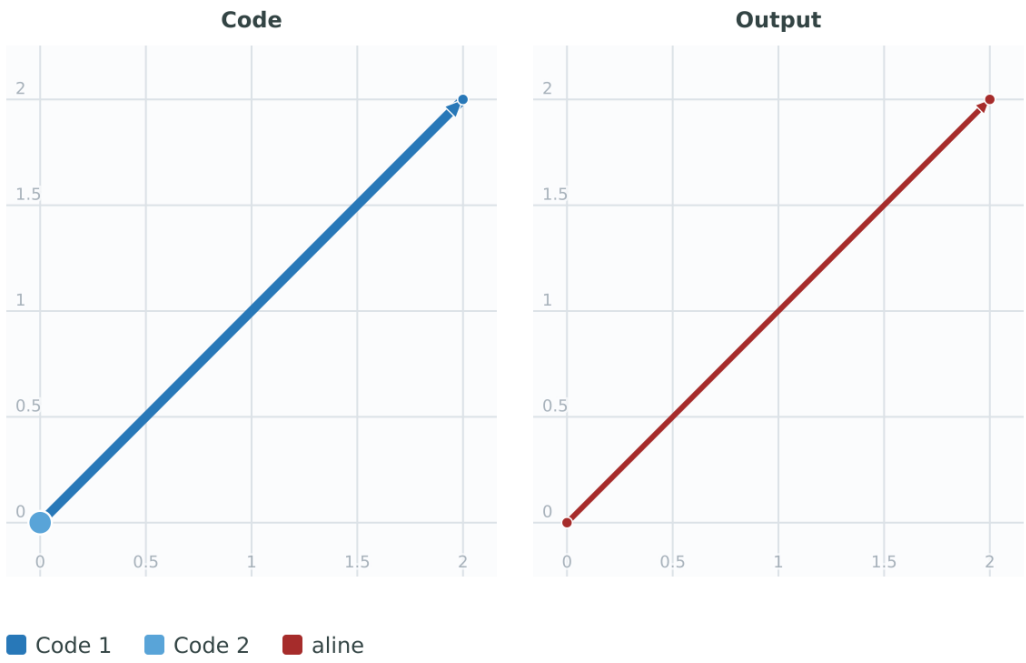
Code

```
SELECT ST_LineFromWKB(
    ST_AsBinary(ST_GeomFromText('LINESTRING(1 2,3 4)'))
) AS aline,
ST_LineFromWKB(
    ST_AsBinary(ST_GeomFromText('POINT(1 2)'))
) IS NULL AS null_return;
```

Output

```
aline                 | null_return
-----------------------------------
LINESTRING(1 2,3 4)   | t
```

Figure



See Also

ST_GeomFromWKB, ST_LinestringFromWKB

7.8.2.5 ST_LinestringFromWKB

ST_LinestringFromWKB — Creates a `LINESTRING` from a WKB representation, optionally using a given SRID.

Synopsis

geometry **ST_LinestringFromWKB**(bytea WKB);
geometry **ST_LinestringFromWKB**(bytea WKB, integer srid);

Description

The `ST_LinestringFromWKB` function takes a Well-Known Binary (WKB) representation of a geometry and an optional Spatial Reference System ID (`SRID`), and creates a `LINESTRING` geometry. It is an alias for ST_LineFromWKB.

If an SRID is not specified, it defaults to 0. `NULL` is returned if the input `bytea` does not represent a `LINESTRING` geometry.



Note
OGC SPEC 3.2.6.2 - optional SRID is from the conformance suite.

Note
If you know all your geometries are LINESTRINGs, it is more efficient to use ST_GeomFromWKB directly. This function calls ST_GeomFromWKB and adds validation that the result is a LINESTRING.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 7.2.9

Examples

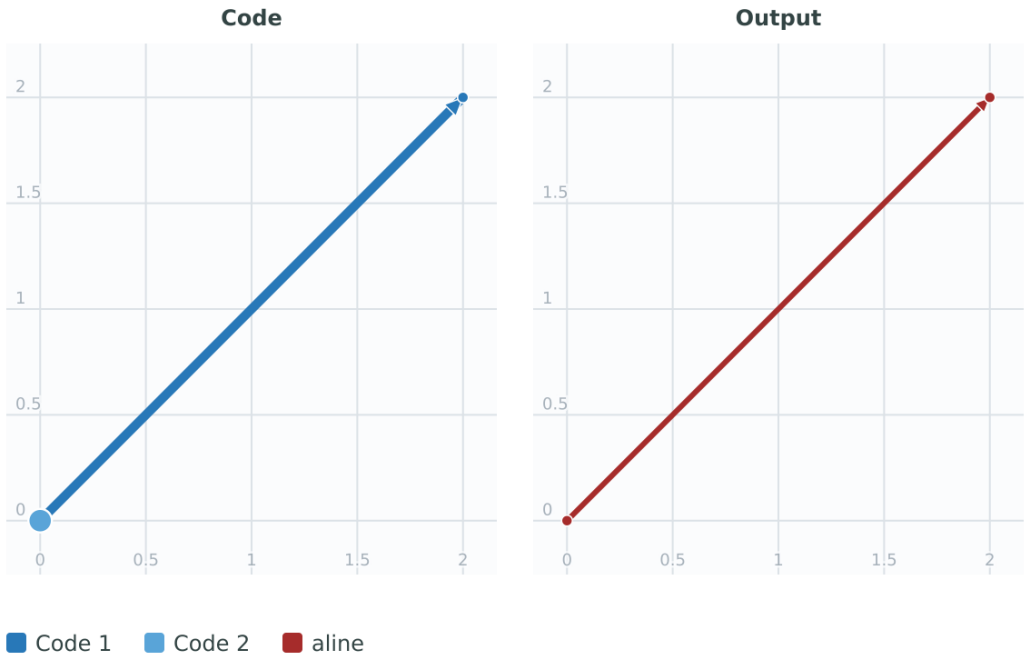
Code

```
SELECT ST_LinestringFromWKB(
    ST_AsBinary(ST_GeomFromText('LINESTRING(1 2,3 4)'))
) AS aline,
ST_LinestringFromWKB(
    ST_AsBinary(ST_GeomFromText('POINT(1 2)'))
) IS NULL AS null_return;
```

Output

```
aline                | null_return
-------------------------------------
LINESTRING(1 2,3 4)  | t
```

Figure



See Also

ST_GeomFromWKB, ST_LineFromWKB

7.8.2.6 ST_PointFromWKB

ST_PointFromWKB — Makes a geometry from WKB with the given SRID

Synopsis

geometry **ST_GeomFromWKB**(bytea geom);
geometry **ST_GeomFromWKB**(bytea geom, integer srid);

Description

The `ST_PointFromWKB` function, takes a well-known binary representation of geometry and a Spatial Reference System ID (`SRID`) and creates an instance of the appropriate geometry type - in this case, a `POINT` geometry. This function plays the role of the Geometry Factory in SQL.

If an SRID is not specified, it defaults to 0. `NULL` is returned if the input `bytea` does not represent a `POINT` geometry.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.7.2

This method implements the SQL/MM specification. SQL-MM 3: 6.1.9

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

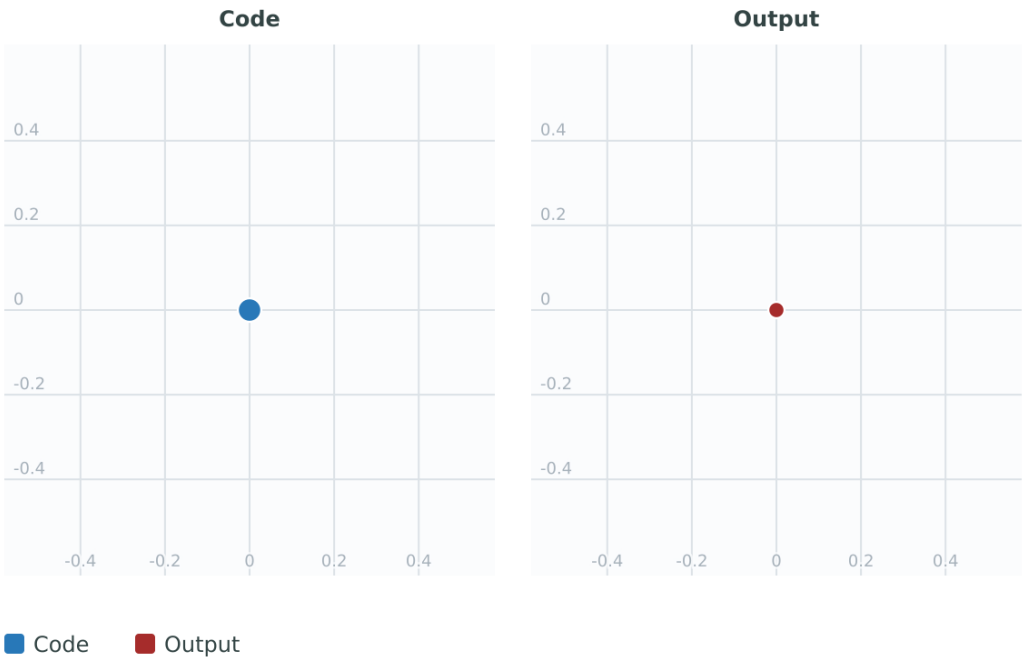
Code

```
SELECT ST_PointFromWKB(ST_AsEWKB('POINT(2 5)'::geometry));
```

Output

```
POINT(2 5)
```

Figure



Code

```
SELECT ST_PointFromWKB(ST_AsEWKB('LINESTRING(2 5,2 6)'::geometry));
```

Output

```
null
```

See Also

ST_GeomFromWKB, ST_LineFromWKB

7.8.2.7 ST_WKBToSQL

ST_WKBToSQL — Returns a geometry from Well-Known Binary (WKB). SQL/MM alias for ST_GeomFromWKB without an SRID argument.

Synopsis

geometry **ST_WKBToSQL**(bytea WKB);

Description

This function is the SQL/MM alias for ST_GeomFromWKB with no SRID argument.

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.36

See Also

ST_GeomFromWKB

7.8.3 Other Formats

7.8.3.1 ST_Box2dFromGeoHash

ST_Box2dFromGeoHash — Return a BOX2D from a GeoHash string.

Synopsis

box2d **ST_Box2dFromGeoHash**(text geohash, integer precision=full_precision_of_geohash);

Description

Return a BOX2D from a GeoHash string.

If no `precision` is specified ST_Box2dFromGeoHash returns a BOX2D based on full precision of the input GeoHash string.

If `precision` is specified ST_Box2dFromGeoHash will use that many characters from the GeoHash to create the BOX2D. Lower precision values results in larger BOX2Ds and larger values increase the precision.

Availability: 2.1.0

Examples

Code

```
SELECT ST_Box2dFromGeoHash('9qqj7nmxncgyy4d0dbxqz0');
```

Output

```
BOX(-115.172816 36.114646,-115.172816 36.114646)
```

Code

```
SELECT ST_Box2dFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 0);
```

Output

```
BOX(-180 -90,180 90)
```

Code

```
SELECT ST_Box2dFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 10);
```

Output

```
BOX(-115.17282128334 36.1146408319473,-115.172810554504 36.1146461963654)
```

See Also

ST_GeoHash, ST_GeomFromGeoHash, ST_PointFromGeoHash

7.8.3.2 ST_GeomFromGeoHash

ST_GeomFromGeoHash — Return a geometry from a GeoHash string.

Synopsis

geometry **ST_GeomFromGeoHash**(text geohash, integer precision=full_precision_of_geohash);

Description

Return a geometry from a GeoHash string. The geometry will be a polygon representing the GeoHash bounds.

If no `precision` is specified ST_GeomFromGeoHash returns a polygon based on full precision of the input GeoHash string.

If `precision` is specified ST_GeomFromGeoHash will use that many characters from the GeoHash to create the polygon.

Availability: 2.1.0

Examples

The cells are returned together so their relative sizes remain visible in a shared coordinate frame.

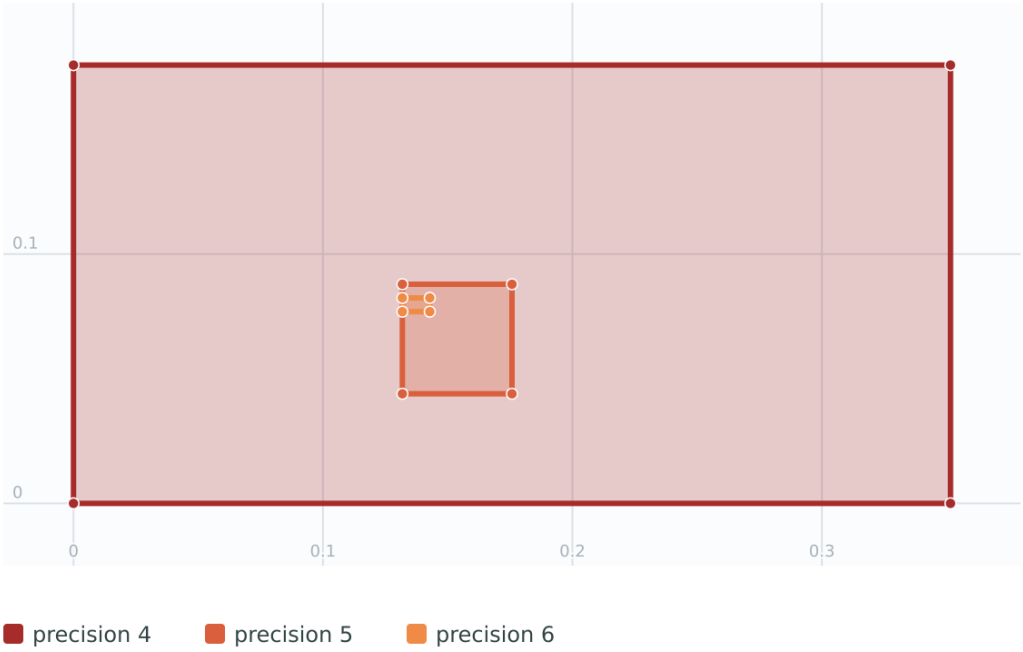
Code

```
SELECT ST_GeomFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 4) AS "precision 4", ST_GeomFromGeoHash( ↩
    '9qqj7nmxncgyy4d0dbxqz0', 5) AS "precision 5", ST_GeomFromGeoHash('9 ↩
    qqj7nmxncgyy4d0dbxqz0', 6) AS "precision 6";
```

Output

```
-[ RECORD 1 ]----------
precision 4 | POLYGON((-115.3125 36.03515625,-115.3125 36.2109375,-114.9609375 36.2109375,-114.9609375 ↩
     36.03515625,-115.3125 36.03515625))
precision 5 | POLYGON((-115.1806640625 36.0791015625,-115.1806640625 36.123046875,-115.13671875 ↩
     36.123046875,-115.13671875 36.0791015625,-115.1806640625 36.0791015625))
precision 6 | POLYGON((-115.1806640625 36.112060546875,-115.1806640625 36.1175537109375,-115.169677734375 ↩
     36.1175537109375,-115.169677734375 36.112060546875,-115.1806640625 36.112060546875))
```

Figure



See Also

ST_GeoHash,ST_Box2dFromGeoHash, ST_PointFromGeoHash

7.8.3.3 ST_GeomFromGML

ST_GeomFromGML — Takes as input GML representation of geometry and outputs a PostGIS geometry object

Synopsis

geometry **ST_GeomFromGML**(text geomgml);
geometry **ST_GeomFromGML**(text geomgml, integer srid);

Description

Constructs a PostGIS ST_Geometry object from the OGC GML representation.

ST_GeomFromGML works only for GML Geometry fragments. It throws an error if you try to use it on a whole GML document.

OGC GML versions supported:

- GML 3.2.1 Namespace
- GML 3.1.1 Simple Features profile SF-2 (with GML 3.1.0 and 3.0.0 backward compatibility)

- GML 2.1.2

OGC GML standards, cf: http://www.opengeospatial.org/standards/gml:

Availability: 1.5, requires libxml2 1.6+

Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced.

Enhanced: 2.0.0 default srid optional parameter added.

Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

GML allow mixed dimensions (2D and 3D inside the same MultiGeometry for instance). As PostGIS geometries don't, ST_GeomFromGML convert the whole geometry to 2D if a missing Z dimension is found once.

GML support mixed SRS inside the same MultiGeometry. As PostGIS geometries don't, ST_GeomFromGML, in this case, reproject all subgeometries to the SRS root node. If no srsName attribute available for the GML root node, the function throw an error.

ST_GeomFromGML function is not pedantic about an explicit GML namespace. You could avoid to mention it explicitly for common usages. But you need it if you want to use XLink feature inside GML.

Examples

A single geometry with srsName.

Code

```
SELECT ST_AsEWKT(ST_GeomFromGML($$
    <gml:LineString xmlns:gml="http://www.opengis.net/gml"
            srsName="EPSG:4269">
        <gml:coordinates>
          -71.16028,42.258729 -71.160837,42.259112 -71.161143,42.25932
        </gml:coordinates>
    </gml:LineString>
$$));
```

Output

```
SRID=4269;LINESTRING(-71.16028 42.258729,-71.160837 42.259112,-71.161143 42.25932)
```

Figure



XLink usage.

Code

```
SELECT ST_AsEWKT(ST_GeomFromGML($$
    <gml:LineString xmlns:gml="http://www.opengis.net/gml"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            srsName="EPSG:4269">
      <gml:pointProperty>
          <gml:Point gml:id="p1"><gml:pos>-71.16028 42.258729</gml:pos></gml:Point>
      </gml:pointProperty>
      <gml:pos>-71.160837 42.259112</gml:pos>
      <gml:pos>-71.161143 42.258729</gml:pos>
        <gml:pointProperty>
            <gml:Point xlink:type="simple" xlink:href="#p1"/>
        </gml:pointProperty>
    </gml:LineString>
$$));
```

Output

```
SRID=4269;LINESTRING(-71.16028 42.258729,-71.160837 42.259112,-71.161143 42.258729,-71.16028 42.258729)
```

Figure



Polyhedral Surface.

Code

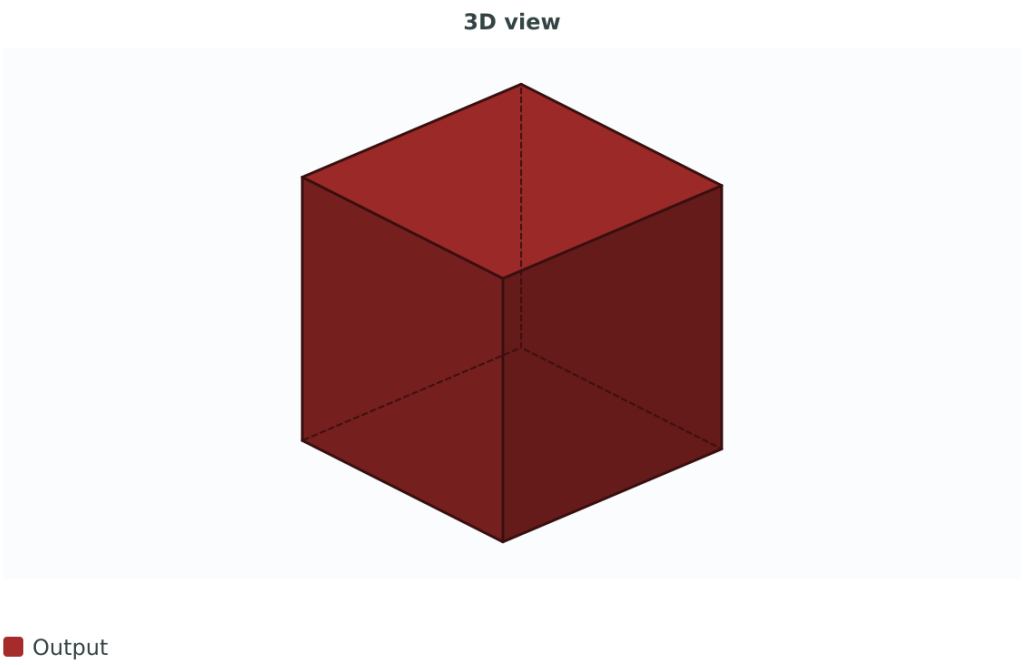
```
SELECT ST_AsEWKT(ST_GeomFromGML('
<gml:PolyhedralSurface xmlns:gml="http://www.opengis.net/gml">
<gml:polygonPatches>
  <gml:PolygonPatch>
    <gml:exterior>
      <gml:LinearRing><gml:posList srsDimension="3">0 0 0 0 0 1 0 1 1 0 1 0 0 0 0</gml: ↩
    posList></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing><gml:posList srsDimension="3">0 0 0 0 1 0 1 1 0 1 0 0 0 0 0</gml:posList ↩
    ></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing><gml:posList srsDimension="3">0 0 0 1 0 0 1 0 1 0 0 1 0 0 0</gml:posList ↩
    ></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing><gml:posList srsDimension="3">1 1 0 1 1 1 1 0 1 1 0 0 1 1 0</gml:posList ↩
    ></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing><gml:posList srsDimension="3">0 1 0 0 1 1 1 1 1 1 0 0 1 0</gml:posList ↩
    ></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
```

```
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing><gml:posList srsDimension="3">0 0 1 1 0 1 1 1 1 0 1 1 0 0 1</gml:posList ↩
   ></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
</gml:polygonPatches>
</gml:PolyhedralSurface>'));
```

Output

```
 POLYHEDRALSURFACE(((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)))
```

Figure



See Also

Section 2.2.3, ST_AsGML, ST_GMLToSQL

7.8.3.4 ST_GeomFromGeoJSON

ST_GeomFromGeoJSON — Takes as input a geojson representation of a geometry and outputs a PostGIS geometry object

Synopsis

geometry **ST_GeomFromGeoJSON**(text geomjson);
geometry **ST_GeomFromGeoJSON**(json geomjson);
geometry **ST_GeomFromGeoJSON**(jsonb geomjson);

Description

Constructs a PostGIS geometry object from the GeoJSON representation.

ST_GeomFromGeoJSON works only for JSON Geometry fragments. It throws an error if you try to use it on a whole JSON document.

Enhanced: 3.0.0 parsed geometry defaults to SRID=4326 if not specified otherwise.

Enhanced: 2.5.0 can now accept json and jsonb as inputs.

Availability: 2.0.0 requires - JSON-C >= 0.9



Note
If you do not have JSON-C enabled, support you will get an error notice instead of seeing an output. To enable JSON-C, run configure --with-jsondir=/path/to/json-c. See Section 2.2.3 for details.

 This function supports 3d and will not drop the z-index.

Examples

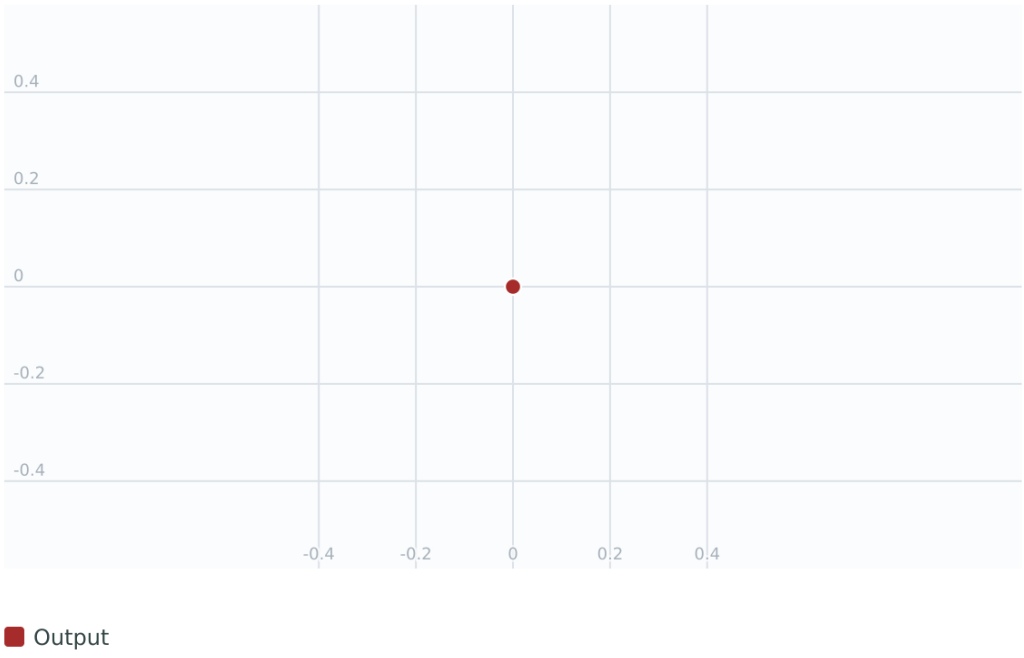
Code

```
SELECT ST_GeomFromGeoJSON('{"type":"Point","coordinates":[-48.23456,20.12345]}') AS wkt;
```

Output

```
POINT(-48.23456 20.12345)
```

Figure



This example creates a 3D LineString.

Code

```
SELECT ST_GeomFromGeoJSON('{"type":"LineString","coordinates":[[1,2,3],[4,5,6],[7,8,9]]}')  ↩
    AS wkt;
```

Output

```
LINESTRING Z (1 2 3,4 5 6,7 8 9)
```

Figure



See Also

ST_AsText, ST_AsGeoJSON, Section 2.2.3

7.8.3.5 ST_GeomFromKML

ST_GeomFromKML — Takes as input KML representation of geometry and outputs a PostGIS geometry object

Synopsis

geometry **ST_GeomFromKML**(text geomkml);

Description

Constructs a PostGIS geometry from an OGC KML representation.

ST_GeomFromKML works only for KML Geometry fragments. It throws an error if you try to use it on a whole KML document.

OGC KML versions supported:

- KML 2.2.0 Namespace

OGC KML standards, cf: http://www.opengeospatial.org/standards/kml:

Availability: 1.5, requires libxml2 2.6+

This function supports 3d and will not drop the z-index.

Note
ST_GeomFromKML does not support SQL/MM curve geometries.

Examples

A single LineString geometry.

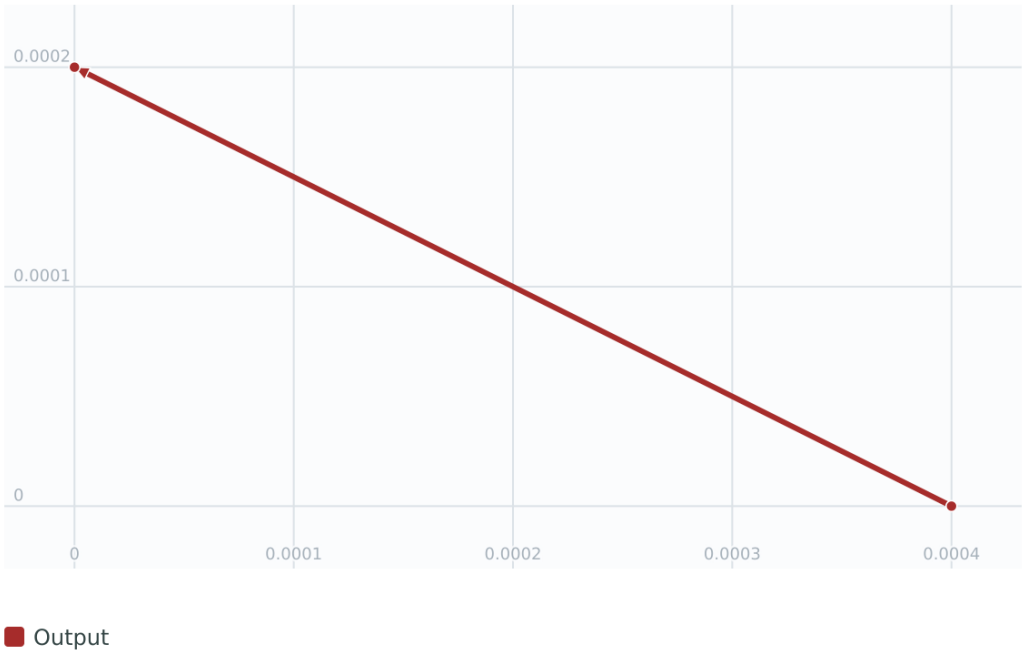
Code

```
SELECT ST_AsEWKT(ST_GeomFromKML($$
    <LineString>
        <coordinates>-71.1663, 42.2614
            -71.1667, 42.2616</coordinates>
    </LineString>
$$));
```

Output

```
SRID=4326;LINESTRING(-71.1663 42.2614,-71.1667 42.2616)
```

Figure



See Also

Section 2.2.3, ST_AsKML

7.8.3.6 ST_GeomFromTWKB

ST_GeomFromTWKB — Creates a geometry instance from a TWKB ("Tiny Well-Known Binary") geometry representation.

Synopsis

geometry **ST_GeomFromTWKB**(bytea twkb);

Description

The `ST_GeomFromTWKB` function, takes a a TWKB ("Tiny Well-Known Binary") geometry representation (WKB) and creates an instance of the appropriate geometry type.

Examples

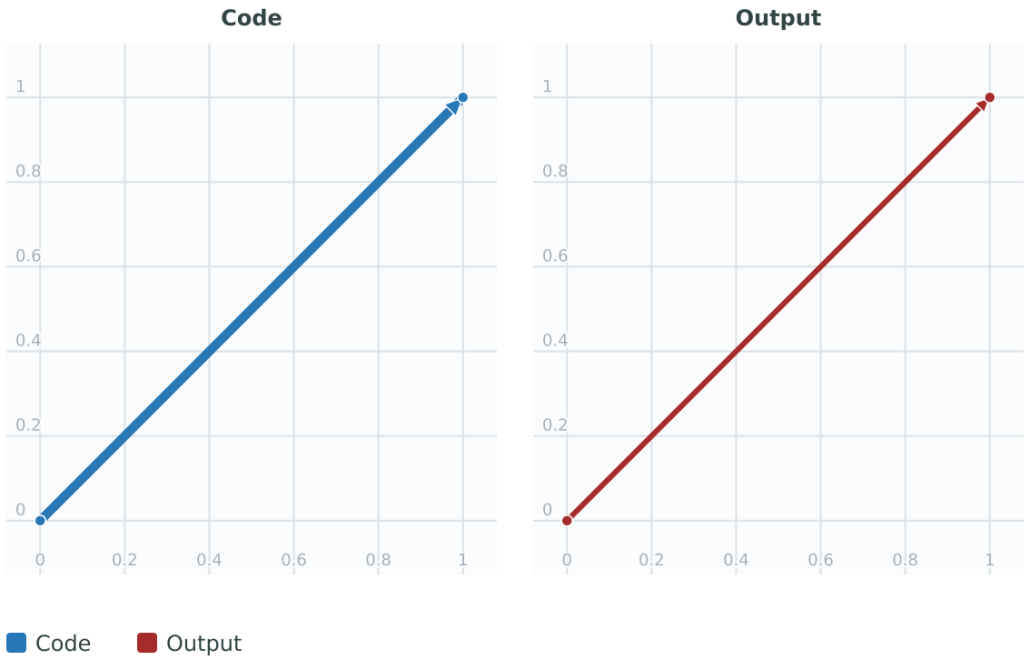
Code

```
SELECT ST_GeomFromTWKB(ST_AsTWKB('LINESTRING(126 34,127 35)'::geometry));
```

Output

```
LINESTRING(126 34, 127 35)
```

Figure



Code

```
SELECT ST_GeomFromTWKB(E'\\x620002f7f40dbce4040105');
```

Output

```
LINESTRING(-113.98 39.198,-113.981 39.195)
```

Figure



See Also

ST_AsTWKB

7.8.3.7 ST_GMLToSQL

ST_GMLToSQL — Returns a geometry from GML. Alias for ST_GeomFromGML.

Synopsis

geometry **ST_GMLToSQL**(text geomgml);
geometry **ST_GMLToSQL**(text geomgml, integer srid);

Description

This function is an alias for ST_GeomFromGML and provides the same signatures.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.50.

Availability: 1.5, requires libxml2 1.6+

Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced.

Enhanced: 2.0.0 default srid optional parameter added.

Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced.

See Also

Section 2.2.3, ST_GeomFromGML, ST_AsGML

7.8.3.8 ST_LineFromEncodedPolyline

ST_LineFromEncodedPolyline — Creates a LineString from an Encoded Polyline.

Synopsis

geometry **ST_LineFromEncodedPolyline**(text polyline, integer precision=5);

Description

Creates a LineString from an Encoded Polyline string.

Optional `precision` specifies how many decimal places will be preserved in Encoded Polyline. Value should be the same on encoding and decoding, or coordinates will be incorrect.

See http://developers.google.com/maps/documentation/utilities/polylinealgorithm

Availability: 2.2.0

Examples

This example creates a LineString from a polyline.

Code

```
SELECT ST_LineFromEncodedPolyline('_p~iF~ps|U_ulLnnqC_mqNvxq`@');
```

Output

```
SRID=4326;LINESTRING(-120.2 38.5,-120.95 40.7,-126.453 43.252)
```

Figure



This example selects a different precision than the one used for polyline encoding.

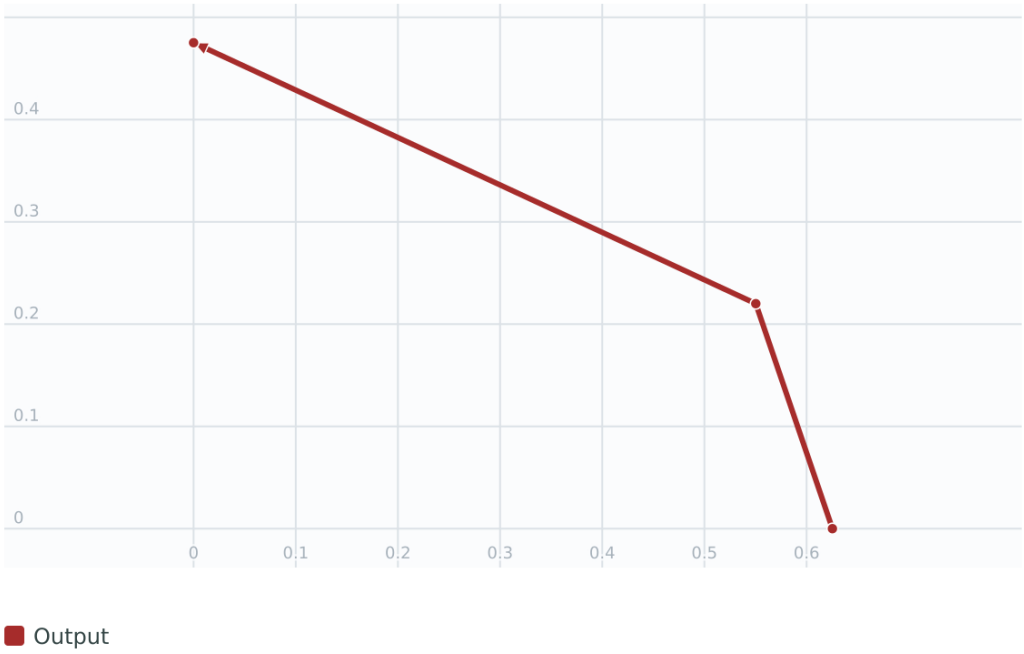
Code

```
SELECT ST_LineFromEncodedPolyline('_p~iF~ps|U_ulLnnqC_mqNvxq`@', 6);
```

Output

```
SRID=4326;LINESTRING(-12.02 3.85,-12.095 4.07,-12.6453 4.3252)
```

Figure



See Also

ST_AsEncodedPolyline

7.8.3.9 ST_PointFromGeoHash

ST_PointFromGeoHash — Return a point from a GeoHash string.

Synopsis

point **ST_PointFromGeoHash**(text geohash, integer precision=full_precision_of_geohash);

Description

Return a point from a GeoHash string. The point represents the center point of the GeoHash.

If no `precision` is specified ST_PointFromGeoHash returns a point based on full precision of the input GeoHash string.

If `precision` is specified ST_PointFromGeoHash will use that many characters from the GeoHash to create the point.

Availability: 2.1.0

Examples

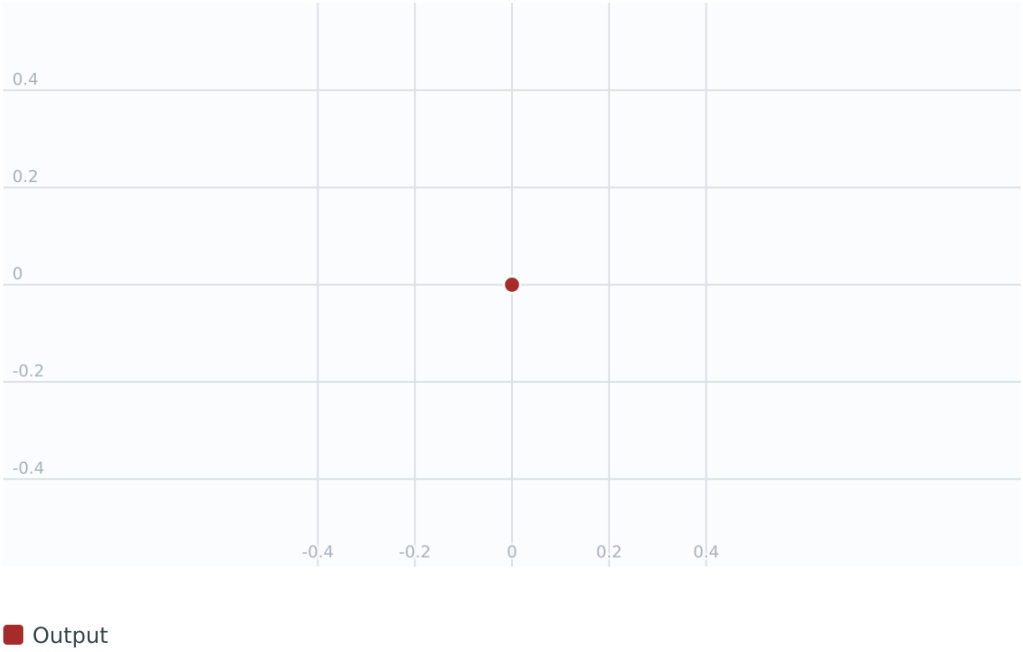
Code

```
SELECT ST_PointFromGeoHash('9qqj7nmxncgyy4d0dbxqz0');
```

Output

```
POINT(-115.172816 36.114646)
```

Figure



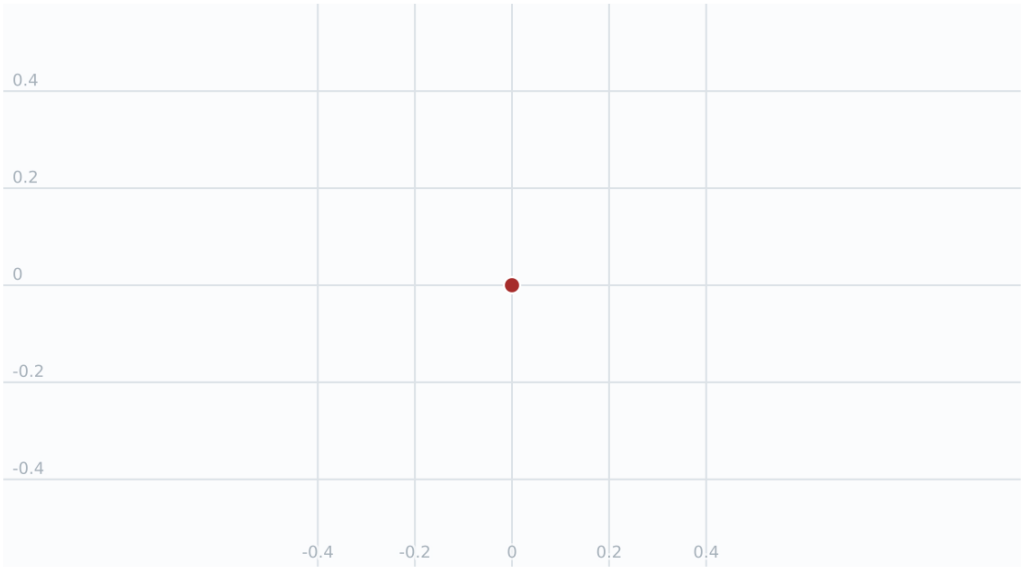
Code

```
SELECT ST_PointFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 4);
```

Output

```
POINT(-115.13671875 36.123046875)
```

Figure



Output

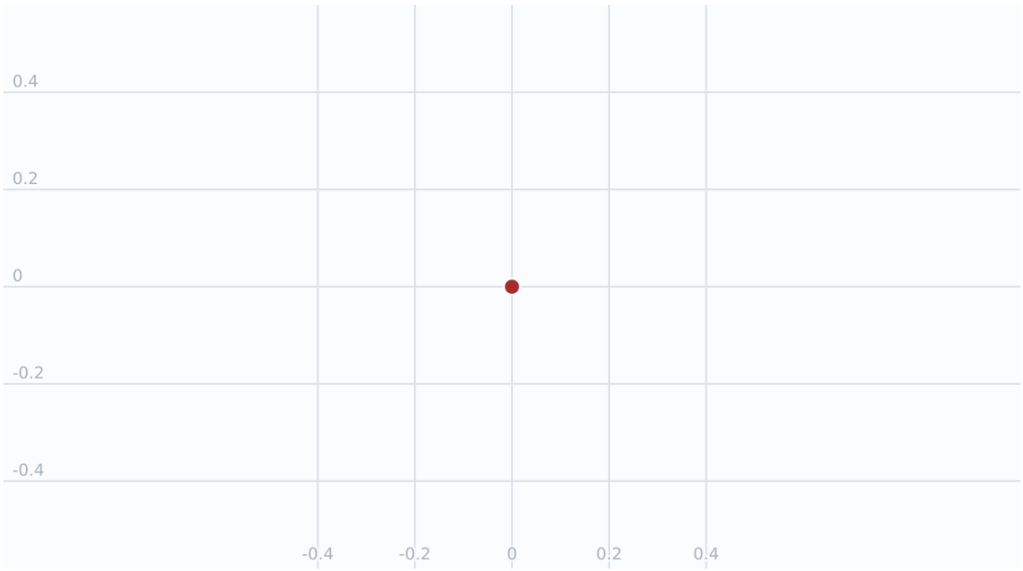
Code

```
SELECT ST_PointFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 10);
```

Output

```
POINT(-115.172815918922 36.1146435141563)
```

Figure



Output

See Also

ST_GeoHash, ST_Box2dFromGeoHash, ST_GeomFromGeoHash

7.8.3.10 ST_FromFlatGeobufToTable

ST_FromFlatGeobufToTable — Creates a table based on the structure of FlatGeobuf data.

Synopsis

void **ST_FromFlatGeobufToTable**(text schemaname, text tablename, bytea FlatGeobuf input data);

Description

Creates a table based on the structure of FlatGeobuf data. (http://flatgeobuf.org).

`schema` Schema name.

`table` Table name.

`data` Input FlatGeobuf data.

Availability: 3.2.0

7.8.3.11 ST_FromFlatGeobuf

ST_FromFlatGeobuf — Reads FlatGeobuf data.

Synopsis

setof anyelement **ST_FromFlatGeobuf**(anyelement Table reference, bytea FlatGeobuf input data);

Description

Reads FlatGeobuf data (http://flatgeobuf.org). NOTE: PostgreSQL bytea cannot exceed 1GB.

`tabletype` reference to a table type.

`data` input FlatGeobuf data.

Availability: 3.2.0

7.9 Geometry Output

7.9.1 Well-Known Text (WKT)

7.9.1.1 ST_AsEWKT

ST_AsEWKT — Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.

Synopsis

text **ST_AsEWKT**(geometry g1);
text **ST_AsEWKT**(geometry g1, integer maxdecimaldigits=15);
text **ST_AsEWKT**(geography g1);
text **ST_AsEWKT**(geography g1, integer maxdecimaldigits=15);

Description

Returns the Well-Known Text representation of the geometry prefixed with the SRID. The optional *maxdecimaldigits* argument may be used to reduce the maximum number of decimal digits after floating point used in output (defaults to 15).

To perform the inverse conversion of EWKT representation to PostGIS geometry use ST_GeomFromEWKT.



Warning
Using the *maxdecimaldigits* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.



Note
The WKT spec does not include the SRID. To get the OGC WKT format use ST_AsText.



Warning
WKT format does not maintain precision so to prevent floating truncation, use ST_AsBinary or ST_AsEWKB format for transport.

Enhanced: 3.1.0 support for optional precision parameter.

Enhanced: 2.0.0 support for Geography, Polyhedral surfaces, Triangles and TIN was introduced.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

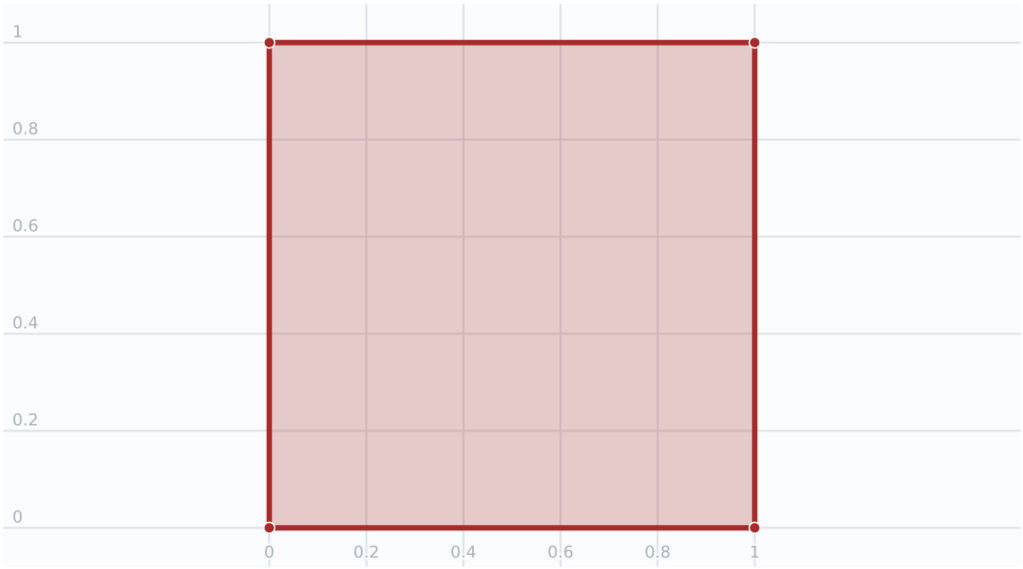
Code

```
SELECT ST_AsEWKT('0103000020 ↩
    e610000001000000050000000000000000000000000000000000000000000000000000000000000000000000000000000000f03f000000000
    '::geometry);
```

Output

```
SRID=4326;POLYGON((0 0,0 1,1 1,1 0,0 0))
```

Figure



Output

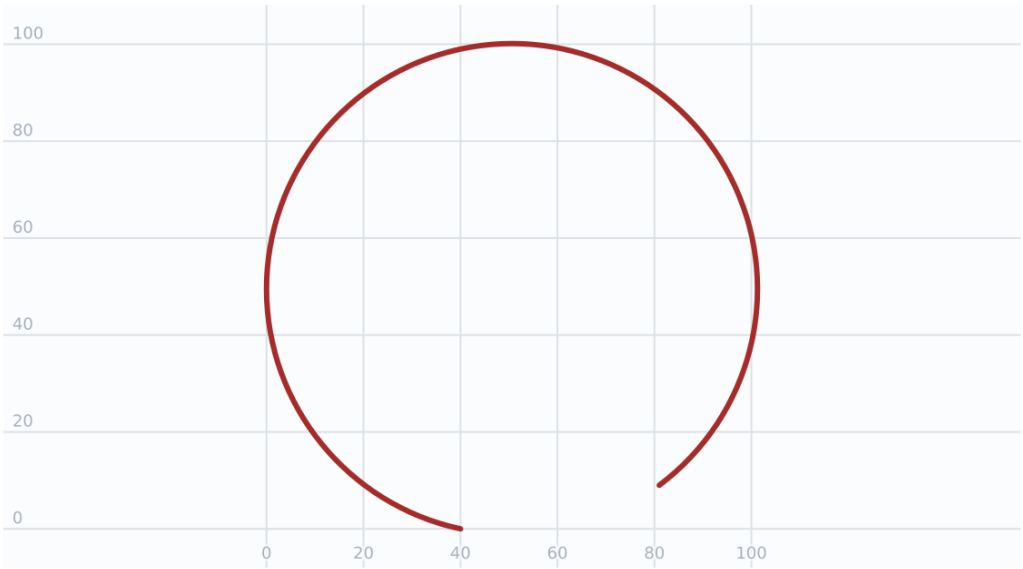
Code

```
SELECT ST_AsEWKT('0108000080030000000000000060 ↩
    E30A4100000000785C0241000000000000F03F0000000018E20A4100000000485F0241000000000000004000000000018
    ');
```

Output

```
CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)
```

Figure



Output

See Also

ST_AsBinary, ST_AsEWKB, ST_AsText, ST_GeomFromEWKT

7.9.1.2 ST_AsText

ST_AsText — Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.

Synopsis

text **ST_AsText**(geometry g1);
text **ST_AsText**(geometry g1, integer maxdecimaldigits = 15);
text **ST_AsText**(geography g1);
text **ST_AsText**(geography g1, integer maxdecimaldigits = 15);

Description

Returns the OGC Well-Known Text (WKT) representation of the geometry/geography. The optional `maxdecimaldigits` argument may be used to limit the number of digits after the decimal point in output ordinates (defaults to 15).

To perform the inverse conversion of WKT representation to PostGIS geometry use ST_GeomFromText.



Note
The standard OGC WKT representation does not include the SRID. To include the SRID as part of the output representation, use the non-standard PostGIS function ST_AsEWKT



Warning
The textual representation of numbers in WKT may not maintain full floating-point precision. To ensure full accuracy for data storage or transport it is best to use Well-Known Binary (WKB) format (see ST_AsBinary and `maxdecimaldigits`).



Warning
Using the `maxdecimaldigits` parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

Availability: 1.5 - support for geography was introduced.

Enhanced: 2.5 - optional parameter precision introduced.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

This method implements the SQL/MM specification. SQL-MM 3: 5.1.25

This method supports Circular Strings and Curves.

Examples

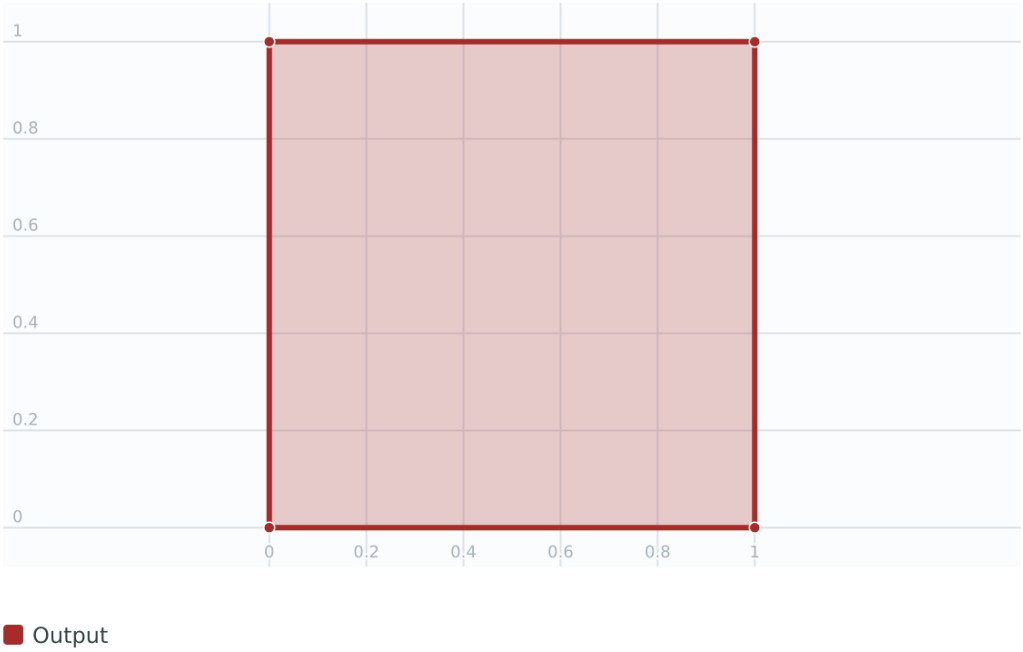
Code

```
SELECT ST_AsText(' ↩
    0103000000010000000500000000000000000000000000000000000000000000000000000000000000000000000000000000 ↩
    f03f000000000000f03f000000000000f03f000000000000f03f000000000000000000000000000000000000000000000000000
    ');
```

Output

```
POLYGON((0 0,0 1,1 1,1 0,0 0))
```

Figure



Full precision output is the default.

Code

```
SELECT ST_AsText('POINT(111.1111111 1.1111111)'::geometry);
```

Output

```
POINT(111.1111111 1.1111111)
```

The *maxdecimaldigits* argument can be used to limit output precision.

Code

```
SELECT ST_AsText('POINT(111.1111111 1.1111111)'::geometry, 2);
```

Output

```
POINT(111.11 1.11)
```

See Also

ST_AsBinary, ST_AsEWKB, ST_AsEWKT, ST_GeomFromText

7.9.2 Well-Known Binary (WKB)

7.9.2.1 ST_AsBinary

ST_AsBinary — Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

Synopsis

bytea **ST_AsBinary**(geometry g1);
bytea **ST_AsBinary**(geometry g1, text NDR_or_XDR);
bytea **ST_AsBinary**(geography g1);
bytea **ST_AsBinary**(geography g1, text NDR_or_XDR);

Description

Returns the OGC/ISO Well-Known Binary (WKB) representation of the geometry. The first function variant defaults to encoding using server machine endian. The second function variant takes a text argument specifying the endian encoding: either 'NDR' for little-endian; or 'XDR' for big-endian. Supplying unknown arguments will result in little-endian output.

WKB format is useful to read geometry data from the database and maintaining full numeric precision. This avoids the precision rounding that can happen with text formats such as WKT.

To perform the inverse conversion of WKB to PostGIS geometry use ST_GeomFromWKB.



Note
The OGC/ISO WKB format does not include the SRID. To get the EWKB format which does include the SRID use ST_AsEWKB



Note
PostgreSQL outputs `bytea` values in hex encoding by default. If your GUI tools require the old behavior, then SET `bytea_output='escape'` in your database.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Enhanced: 2.0.0 support for higher coordinate dimensions was introduced.

Enhanced: 2.0.0 support for specifying endian with geography was introduced.

Availability: 1.5.0 geography support was introduced.

Changed: 2.0.0 Inputs to this function can not be unknown -- must be geometry. Constructs such as `ST_AsBinary('POINT(1 2)')` are no longer valid and you will get an `n ST_AsBinary(unknown) is not unique error`. Code like that needs to be changed to `ST_AsBinary('POINT(1 2)'::geometry);`. If that is not possible, then install `legacy.sql`.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1


This method implements the SQL/MM specification. SQL-MM 3: 5.1.37


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).


This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_AsBinary(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
\x0103000000010000000500000000000000000000000000000000000000000000000000000000000000000000000000000000
000000f03f000000000000f03f000000000000f03f000000000000f03f000000000000000000000000
000000000000000000000000
```

Code

```
SELECT ST_AsBinary(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326), 'XDR');
```

Output

```
\x0000000003000000010000000500000000000000000000000000000000000000000000000000000000000000000003ff000
00000000003ff0000000000000003ff0000000000000003ff0000000000000000000000000000000000000
000000000000000000000000
```

Use the PostgreSQL `encode` function to produce a hexadecimal WKB string without the `\x` prefix used for bytea display. Specifying `NDR` makes the byte order deterministic.

Code

```
SELECT encode(
    ST_AsBinary('POINT(15.123 21.456)'::geometry, 'NDR'),
    'hex'
) AS wkb_hex;
```

Output

```
0101000000e5d022dbf93e2e40dbf97e6abc743540
```

See Also

ST_GeomFromWKB, ST_AsEWKB, ST_AsTWKB, ST_AsText,

7.9.2.2 ST_AsEWKB

ST_AsEWKB — Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.

Synopsis

bytea **ST_AsEWKB**(geometry g1);
bytea **ST_AsEWKB**(geometry g1, text NDR_or_XDR);

Description

Returns the Extended Well-Known Binary (EWKB) representation of the geometry with SRID metadata. The first function variant defaults to encoding using server machine endian. The second function variant takes a text argument specifying the endian encoding: either 'NDR' for little-endian; or 'XDR' for big-endian. Supplying unknown arguments will result in little-endian output.

WKB format is useful to read geometry data from the database and maintaining full numeric precision. This avoids the precision rounding that can happen with text formats such as WKT.

To perform the inverse conversion of EWKB to PostGIS geometry use ST_GeomFromEWKB.



Note
To get the OGC/ISO WKB format use ST_AsBinary. Note that OGC/ISO WKB format does not include the SRID.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Code

```
SELECT ST_AsEWKB(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
\x0103000020e6100000010000000500000000000000000000000000000000000000000000000000000000000000000000
0000000000000f03f000000000000f03f000000000000f03f000000000000f03f000000000000000000000000000000000
00000000000000000000000000000000
```

Code

```
SELECT ST_AsEWKB(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326), 'XDR');
```

Output

```
\x00200000003000010e60000000100000005000000000000000000000000000000000000000000000000000000000000
003ff0000000000003ff0000000000003ff0000000000003ff0000000000000000000000000000000000
00000000000000000000000000000000
```

See Also

ST_AsBinary, ST_GeomFromEWKB, ST_SRID

7.9.2.3 ST_AsHEXEWKB

ST_AsHEXEWKB — Returns a Geometry in HEXEWKB format (as text) using either little-endian (NDR) or big-endian (XDR) encoding.

Synopsis

text **ST_AsHEXEWKB**(geometry g1, text NDRorXDR);
text **ST_AsHEXEWKB**(geometry g1);

Description

Returns a Geometry in HEXEWKB format (as text) using either little-endian (NDR) or big-endian (XDR) encoding. If no encoding is specified, then NDR is used.

Availability: 1.2.2

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_AsHEXEWKB(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
0103000020E6100000010000000500
0000000000000000000000000000000000000000
0000000000000000000000000000000000F03F
000000000000F03F000000000000F03F000000000000F03
F0000000000000000000000000000000000000000000000000000000
```

Which gives the same answer as:

Code

```
SELECT ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326)::text;
```

7.9.3 Other Formats

7.9.3.1 ST_AsEncodedPolyline

ST_AsEncodedPolyline — Returns an Encoded Polyline from a LineString geometry.

Synopsis

text **ST_AsEncodedPolyline**(geometry geom, integer precision=5);

Description

Returns the geometry in the Encoded Polyline Algorithm Format. The format stores latitude and longitude deltas as integers after multiplying coordinates by 10 raised to the requested `precision`.

The informal names *polyline5* and *polyline6* refer to the same encoding with precision 5 and 6. The precision must match when encoding and decoding; otherwise the coordinates are scaled incorrectly.

The input geometry must use SRID 4326, with longitude as X and latitude as Y. Use ST_Transform to transform geometries in other coordinate systems before encoding.

The OSRM API returns precision 5 for `geometries=polyline` and precision 6 for `geometries=polyline6`. Decode the resulting `route.geometry` with ST_LineFromEncodedPolyline using the matching precision.

Availability: 2.2.0

Examples

Encode the same WGS 84 LineString using precision 5 and 6.

Code

```
WITH route AS (
  SELECT ST_GeomFromEWKT(
    'SRID=4326;LINESTRING(-120.2 38.5,-120.95 40.7,-126.453 43.252)'
  ) AS geom
)
SELECT ST_AsEncodedPolyline(geom, 5) AS polyline5,
       ST_AsEncodedPolyline(geom, 6) AS polyline6
FROM route;
```

Output

```
         polyline5          |            polyline6
----------------------------+---------------------------------
 _p~iF~ps|U_ulLnnqC_mqNvxq`@ | _izlhA~rlgdF_{geC~ywl@_kwzCn`{nI
```

Decode an OSRM `route.geometry` returned with `geometries=polyline6`.

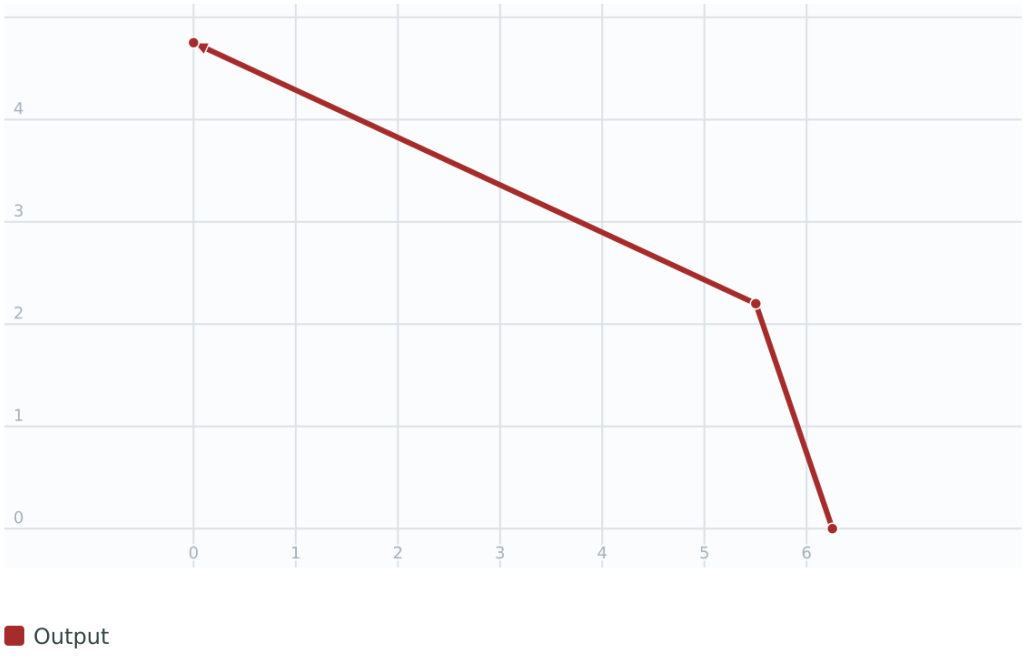
Code

```
SELECT ST_AsEWKT(ST_LineFromEncodedPolyline(
  '_izlhA~rlgdF_{geC~ywl@_kwzCn`{nI',
  6
)) AS route;
```

Output

```
SRID=4326;LINESTRING(-120.2 38.5,-120.95 40.7,-126.453 43.252)
```

Figure



The Google Maps JavaScript Geometry library decodes precision-5 strings. This browser example assumes that `map` is an initialized Google map.

Code

```
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&libraries=geometry"></ ↩
    script>
<script>
  const encodedPath = "_p~iF~ps|U_ulLnnqC_mqNvxq`@";
  const flightPath = new google.maps.Polyline({
    path: google.maps.geometry.encoding.decodePath(encodedPath),
    map: map,
    strokeColor: "#0000CC",
    strokeOpacity: 1.0,
    strokeWeight: 4
  });
</script>
```

See Also

ST_LineFromEncodedPolyline, ST_Transform

7.9.3.2 ST_AsFlatGeobuf

ST_AsFlatGeobuf — Return a FlatGeobuf representation of a set of rows.

Synopsis

bytea **ST_AsFlatGeobuf**(anyelement set row);
bytea **ST_AsFlatGeobuf**(anyelement row, bool index);
bytea **ST_AsFlatGeobuf**(anyelement row, bool index, text geom_name);

Description

Return a FlatGeobuf representation (http://flatgeobuf.org) of a set of rows corresponding to a FeatureCollection. NOTE: PostgreSQL bytea cannot exceed 1GB.

`row` row data with at least a geometry column.

`index` toggle spatial index creation. Default is false.

`geom_name` is the name of the geometry column in the row data. If NULL it will default to the first found geometry column.

Availability: 3.2.0

7.9.3.3 ST_AsGeobuf

ST_AsGeobuf — Return a Geobuf representation of a set of rows.

Synopsis

bytea **ST_AsGeobuf**(anyelement set row);
bytea **ST_AsGeobuf**(anyelement row, text geom_name);

Description

Return a Geobuf representation (https://github.com/mapbox/geobuf) of a set of rows corresponding to a FeatureCollection. Every input geometry is analyzed to determine maximum precision for optimal storage. Note that Geobuf in its current form cannot be streamed so the full output will be assembled in memory.

`row` row data with at least a geometry column.

`geom_name` is the name of the geometry column in the row data. If NULL it will default to the first found geometry column.

Availability: 2.4.0

Examples

Code

```
SELECT encode(ST_AsGeobuf(q, 'geom'), 'base64')
FROM (SELECT 'POLYGON((0 0,0 1,1 1,1 0,0 0))'::geometry AS geom) AS q;
```

Output

```
GAAiEAoOCgwIBBoIAAAAAgIAAAE=
```

7.9.3.4 ST_AsGeoJSON

ST_AsGeoJSON — Return a geometry or feature in GeoJSON format.

Synopsis

text **ST_AsGeoJSON**(record feature, text geom_column="", integer maxdecimaldigits=9, boolean pretty_bool=false, text id_column='')
text **ST_AsGeoJSON**(geometry geom, integer maxdecimaldigits=9, integer options=8);
text **ST_AsGeoJSON**(geography geog, integer maxdecimaldigits=9, integer options=0);

Description

Returns a geometry as a GeoJSON "geometry" object, or a row as a GeoJSON "feature" object.

The resulting GeoJSON geometry and feature representations conform with the GeoJSON specifications RFC 7946, except when the parsed geometries are referenced with a CRS other than WGS84 longitude and latitude (EPSG:4326, urn:ogc:def:crs:OGC::CRS84); the GeoJSON geometry object will then have a short CRS SRID identifier attached by default. 2D and 3D Geometries are both supported. GeoJSON only supports SFS 1.1 geometry types (no curve support for example).

The `geom_column` parameter is used to distinguish between multiple geometry columns. If omitted, the first geometry column in the record will be determined. Conversely, passing the parameter will save column type lookups.

The `maxdecimaldigits` argument may be used to reduce the maximum number of decimal places used in output (defaults to 9). If you are using EPSG:4326 and are outputting the geometry only for display, `maxdecimaldigits`=6 can be a good choice for many maps.



Warning
Using the *`maxdecimaldigits`* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

The `options` argument can be used to add BBOX or CRS in GeoJSON output:

- 0: means no option

- 1: GeoJSON BBOX
- 2: GeoJSON Short CRS (e.g EPSG:4326)
- 4: GeoJSON Long CRS (e.g urn:ogc:def:crs:EPSG::4326)
- 8: GeoJSON Short CRS if not EPSG:4326 (default)

The `id_column` parameter is used to set the "id" member of the returned GeoJSON features. As per GeoJSON RFC, this SHOULD be used whenever a feature has a commonly used identifier, such as a primary key. When not specified, the produced features will not get an "id" member and any columns other than the geometry, including any potential keys, will just end up inside the feature's "properties" member.

When the input record includes duplicate column names, the resulting "properties" object will contain repeated keys. `ST_AsGeoJSON` raises a warning in that case because PostgreSQL jsonb keeps only the last value for duplicate keys.

The GeoJSON specification states that polygons are oriented using the Right-Hand Rule, and some clients require this orientation. This can be ensured by using ST_ForcePolygonCCW. The specification also requires that geometry be in the WGS84 coordinate system (SRID = 4326). If necessary geometry can be projected into WGS84 using ST_Transform: `ST_Transform(geom, 4326 )`.

GeoJSON can be tested and viewed online at geojson.io and geojsonlint.com. It is widely supported by web mapping frameworks:

- OpenLayers GeoJSON Example
- Leaflet GeoJSON Example
- Mapbox GL GeoJSON Example

Availability: 1.3.4

Availability: 1.5.0 geography support was introduced.

Changed: 2.0.0 support default args and named args.

Changed: 3.0.0 support records as input

Changed: 3.0.0 output SRID if not EPSG:4326.

Changed: 3.5.0 allow specifying the column containing the feature id

Changed: 3.7.0 added warning about duplicate keys

 This function supports 3d and will not drop the z-index.

Examples

Generate a FeatureCollection:

Code

```
SELECT json_build_object(
    'type', 'FeatureCollection',
    'features', json_agg(ST_AsGeoJSON(t.*, id_column => 'id')::json)
    )
FROM ( VALUES (1, 'one', 'POINT(1 1)'::geometry),
              (2, 'two', 'POINT(2 2)'),
              (3, 'three', 'POINT(3 3)')
     ) as t(id, name, geom);
```

Output

```
{"type" : "FeatureCollection", "features" : [{"type": "Feature", "geometry": {"type":"Point","coordinates":[1,1]}, "id ↩
    ": 1, "properties": {"name": "one"}}, {"type": "Feature", "geometry": {"type":"Point","coordinates":[2,2]}, "id": ↩
    2, "properties": {"name": "two"}}, {"type": "Feature", "geometry": {"type":"Point","coordinates":[3,3]}, "id": 3, ↩
    "properties": {"name": "three"}}]}
```

Generate a Feature:

Code

```
SELECT ST_AsGeoJSON(t.*, id_column => 'id')
FROM (VALUES (1, 'one', 'POINT(1 1)'::geometry)) AS t(id, name, geom);
```

Output

```
{"type": "Feature", "geometry": {"type":"Point","coordinates":[1,1]}, "id": 1, "properties": {"name": "one"}}
```

Don't forget to transform your data to WGS84 longitude, latitude to conform with the GeoJSON specification:

Code

```
SELECT ST_AsGeoJSON(ST_Transform(geom, 4326)) FROM fe_edges LIMIT 1;
```

Output

```
{"type":"MultiLineString","coordinates":[[[-89.734634999999997,31.492072000000000],
[-89.734955999999997,31.492237999999997]]]}
```

3D geometries are supported:

Code

```
SELECT ST_AsGeoJSON('LINESTRING(1 2 3,4 5 6)');
```

Output

```
{"type":"LineString","coordinates":[[1,2,3],[4,5,6]]}
```

Options argument can be used to add BBOX and CRS in GeoJSON output:

Code

```
SELECT ST_AsGeoJSON(ST_SetSRID('POINT(1 1)'::geometry, 4326), 9, 4|1);
```

Output

```
{"type":"Point","crs":{"type":"name","properties":{"name":"urn:ogc:def:crs:EPSG::4326"}},"bbox ↩
    ":[1.000000000,1.000000000,1.000000000,1.000000000],"coordinates":[1,1]}
```

See Also

ST_GeomFromGeoJSON, ST_ForcePolygonCCW, ST_Transform

7.9.3.5 ST_AsGML

ST_AsGML — Return the geometry as a GML version 2 or 3 element.

Synopsis

text **ST_AsGML**(geometry geom, integer maxdecimaldigits=15, integer options=0);
text **ST_AsGML**(geography geog, integer maxdecimaldigits=15, integer options=0, text nprefix=null, text id=null);
text **ST_AsGML**(integer version, geometry geom, integer maxdecimaldigits=15, integer options=0, text nprefix=null, text id=null);
text **ST_AsGML**(integer version, geography geog, integer maxdecimaldigits=15, integer options=0, text nprefix=null, text id=null);

Description

Return the geometry as a Geography Markup Language (GML) element. The version parameter, if specified, may be either 2 or 3. If no version parameter is specified then the default is assumed to be 2. The `maxdecimaldigits` argument may be used to reduce the maximum number of decimal places used in output (defaults to 15).



Warning
Using the *`maxdecimaldigits`* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

GML 2 refer to 2.1.2 version, GML 3 to 3.1.1 version

The 'options' argument is a bitfield. It could be used to define CRS output type in GML output, and to declare data as lat/lon:

- 0: GML Short CRS (e.g EPSG:4326), default value
- 1: GML Long CRS (e.g urn:ogc:def:crs:EPSG::4326)
- 2: For GML 3 only, remove srsDimension attribute from output.
- 4: For GML 3 only, use <LineString> rather than <Curve> tag for lines.
- 16: Declare that data are lat/lon (e.g srid=4326). Default is to assume that data are planars. This option is useful for GML 3.1.1 output only, related to axis order. So if you set it, it will swap the coordinates so order is lat lon instead of database lon lat.
- 32: Output the box of the geometry (envelope).

The 'namespace prefix' argument may be used to specify a custom namespace prefix or no prefix (if empty). If null or omitted 'gml' prefix is used

Availability: 1.3.2

Availability: 1.5.0 geography support was introduced.

Enhanced: 2.0.0 prefix support was introduced. Option 4 for GML3 was introduced to allow using LineString instead of Curve tag for lines. GML3 Support for Polyhedral surfaces and TINS was introduced. Option 32 was introduced to output the box.

Changed: 2.0.0 use default named args

Enhanced: 2.1.0 id support was introduced, for GML 3.



Note
Only version 3+ of ST_AsGML supports Polyhedral Surfaces and TINS.

 This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 17.2

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Version 2.

Code

```
SELECT ST_AsGML(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
<gml:Polygon srsName="EPSG:4326"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates>0,0 0,1 1,1 1,0 0,0</gml: ↩
    coordinates></gml:LinearRing></gml:outerBoundaryIs></gml:Polygon>
```

Version 3.

Flip coordinates and output extended EPSG (16 | 1)

Code

```
SELECT ST_AsGML(3, ST_GeomFromText('POINT(5.234234233242 6.34534534534)', 4326), 5, 17);
```

Output

```
<gml:Point srsName="urn:ogc:def:crs:EPSG::4326"><gml:pos srsDimension="2">6.34535 5.23423</gml:pos></gml:Point>
```

Output the envelope (32).

Code

```
SELECT ST_AsGML(3, ST_GeomFromText('LINESTRING(1 2, 3 4, 10 20)',4326), 5, 32);
```

Output

```
<gml:Envelope srsName="EPSG:4326" srsDimension="2"><gml:lowerCorner>1 2</gml:lowerCorner><gml:upperCorner>10 20</gml: ↩
    upperCorner></gml:Envelope>
```

Output the envelope (32), reverse (lat lon instead of lon lat) (16), long srs (1) = 32 | 16 | 1 = 49.

Code

```
SELECT ST_AsGML(3, ST_GeomFromText('LINESTRING(1 2, 3 4, 10 20)',4326), 5, 49);
```

Output

```
<gml:Envelope srsName="urn:ogc:def:crs:EPSG::4326" srsDimension="2"><gml:lowerCorner>2 1</gml:lowerCorner><gml: ↩
    upperCorner>20 10</gml:upperCorner></gml:Envelope>
```

This example uses a PolyhedralSurface.

Code

```
SELECT ST_AsGML(
3,
'POLYHEDRALSURFACE( ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )'::geometry);
```

Output

```
 <gml:PolyhedralSurface>
<gml:polygonPatches>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3">0 0 0 0 0 1 0 1 1 0 1 0 0 0 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3">0 0 0 0 1 0 1 1 0 1 0 0 0 0 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3">0 0 0 1 0 0 1 0 1 0 0 1 0 0 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3">1 1 0 1 1 1 1 0 1 1 0 0 1 1 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3">0 1 0 0 1 1 1 1 1 1 1 0 0 1 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3">0 0 1 1 0 1 1 1 1 0 1 1 0 0 1</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
</gml:polygonPatches>
</gml:PolyhedralSurface>
```

See Also

ST_GeomFromGML

7.9.3.6 ST_AsKML

ST_AsKML — Return the geometry as a KML element.

Synopsis

text **ST_AsKML**(geometry geom, integer maxdecimaldigits=15, text nprefix=NULL);
text **ST_AsKML**(geography geog, integer maxdecimaldigits=15, text nprefix=NULL);

Description

Return the geometry as a Keyhole Markup Language (KML) element. default maximum number of decimal places is 15, default namespace is no prefix.

Warning
Using the *maxdecimaldigits* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

Availability: 1.2.2 - later variants that include version param came in 1.3.2

Enhanced: 2.0.0 - Add prefix namespace, use default and named args

Changed: 3.0.0 - Removed the "versioned" variant signature

Note
Requires PostGIS be compiled with Proj support. Use PostGIS_Full_Version to confirm you have proj support compiled in.

Note
AsKML output will not work with geometries that do not have an SRID

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_AsKML(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
<Polygon><outerBoundaryIs><LinearRing><coordinates>0,0 0,1 1,1 1,0 0,0</coordinates></LinearRing></outerBoundaryIs></ ↩
    Polygon>
```

This example outputs a 3D LineString.

Code

```
SELECT ST_AsKML('SRID=4326;LINESTRING(1 2 3,4 5 6)');
```

Output

```
<LineString><coordinates>1,2,3 4,5,6</coordinates></LineString>
```

See Also

ST_AsSVG, ST_AsGML

7.9.3.7 ST_AsLatLonText

ST_AsLatLonText — Return the Degrees, Minutes, Seconds representation of the given point.

Synopsis

text **ST_AsLatLonText**(geometry pt, text format='');

Description

Returns the Degrees, Minutes, Seconds representation of the point.



Note
It is assumed the point is in a lat/lon projection. The X (lon) and Y (lat) coordinates are normalized in the output to the "normal" range (-180 to +180 for lon, -90 to +90 for lat).

The text parameter is a format string containing the format for the resulting text, similar to a date format string. Valid tokens are "D" for degrees, "M" for minutes, "S" for seconds, and "C" for cardinal direction (NSEW). DMS tokens may be repeated to indicate desired width and precision ("SSS.SSSS" means " 1.0023").

"M", "S", and "C" are optional. If "C" is omitted, degrees are shown with a "-" sign if south or west. If "S" is omitted, minutes will be shown as decimal with as many digits of precision as you specify. If "M" is also omitted, degrees are shown as decimal with as many digits precision as you specify.

If the format string is omitted (or zero-length) a default format will be used.

Availability: 2.0

Examples

Default format.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)'));
```

Output

```
2\textdegree{}19'29.928"S 3\textdegree{}14'3.243"W
```

Providing a format (same as the default).

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D\textdegree{}M''S.SSS"C'));
```

Output

```
2\textdegree{}19'29.928"S 3\textdegree{}14'3.243"W
```

Characters other than D, M, S, C and . are just passed through.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D degrees, M minutes, S seconds to ↩
    the C'));
```

Output

```
2 degrees, 19 minutes, 30 seconds to the S 3 degrees, 14 minutes, 3 seconds to the W
```

Signed degrees instead of cardinal directions.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D\textdegree{}M''S.SSS"'));
```

Output

```
-2\textdegree{}19'29.928" -3\textdegree{}14'3.243"
```

Decimal degrees.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D.DDDD degrees C'));
```

Output

```
2.3250 degrees S 3.2342 degrees W
```

To convert parsed degree, minute, and second components to decimal degrees, apply the cardinal-direction sign explicitly.

Code

```
WITH dms(label, degrees, minutes, seconds, hemisphere) AS (
  VALUES
    ('latitude', 43::numeric, 0::numeric, 50.60::numeric, 'S'),
    ('longitude', 147::numeric, 12::numeric, 18.20::numeric, 'E')
)
SELECT label,
       round(
         (CASE WHEN hemisphere IN ('S', 'W') THEN -1 ELSE 1 END)
         * (degrees + minutes / 60 + seconds / 3600),
         9
       ) AS decimal_degrees
FROM dms;
```

Output

```
   label   | decimal_degrees
-----------+-----------------
 latitude  |   -43.014055556
 longitude |   147.205055556
```

Excessively large values are normalized.

Code

```
SELECT (ST_AsLatLonText('POINT (-302.2342342 -792.32498)'));
```

Output

```
72\textdegree{}19'29.928"S 57\textdegree{}45'56.757"E
```

7.9.3.8 ST_AsMARC21

ST_AsMARC21 — Returns geometry as a MARC21/XML record with a geographic datafield (034).

Synopsis

text **ST_AsMARC21** (geometry geom , text format='hdddmmss');

Description

This function returns a MARC21/XML record with Coded Cartographic Mathematical Data representing the bounding box of a given geometry. The `format` parameter allows to encode the coordinates in subfields `$d`,`$e`,`$f` and `$g` in all formats supported by the MARC21/XML standard. Valid formats are:

- cardinal direction, degrees, minutes and seconds (default): `hdddmmss`
- decimal degrees with cardinal direction: `hddd.dddddd`
- decimal degrees without cardinal direction: `ddd.dddddd`
- decimal minutes with cardinal direction: `hdddmm.mmmm`
- decimal minutes without cardinal direction: `dddmm.mmmm`
- decimal seconds with cardinal direction: `hdddmmss.sss`

The decimal sign may be also a comma, e.g. `hdddmm,mmmm`.

The precision of decimal formats can be limited by the number of characters after the decimal sign, e.g. `hdddmm.mm` for decimal minutes with a precision of two decimals.

This function ignores the Z and M dimensions.

LOC MARC21/XML versions supported:

- MARC21/XML 1.1

Availability: 3.3.0



Note
This function does not support non lon/lat geometries, as they are not supported by the MARC21/XML standard (Coded Cartographic Mathematical Data).



Note
The MARC21/XML Standard does not provide any means to annotate the spatial reference system for Coded Cartographic Mathematical Data, which means that this information will be lost after conversion to MARC21/XML.

Examples

Converting a `POINT` to MARC21/XML formatted as hdddmmss (default)

Code

```
SELECT ST_AsMARC21('SRID=4326;POINT(-4.504289 54.253312)'::geometry);
```

Output

```
                              st_asmarc21
-----------------------------------------------------
<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a">a</subfield>
        <subfield code="d">W0043015</subfield>
        <subfield code="e">W0043015</subfield>
        <subfield code="f">N0541512</subfield>
        <subfield code="g">N0541512</subfield>
    </datafield>
</record>
```

Converting a `POLYGON` to MARC21/XML formatted in decimal degrees

Code

```
SELECT ST_AsMARC21('SRID=4326;POLYGON((-4.5792388916015625 ↩
    54.18172660239091,-4.56756591796875 54.196993557130355,-4.546623229980469 ↩
    54.18313300502024,-4.5792388916015625 54.18172660239091))'::geometry,'hddd.dddd');

<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a">a</subfield>
        <subfield code="d">W004.5792</subfield>
        <subfield code="e">W004.5466</subfield>
        <subfield code="f">N054.1970</subfield>
        <subfield code="g">N054.1817</subfield>
    </datafield>
</record>
```

Converting a GEOMETRYCOLLECTION to MARC21/XML formatted in decimal minutes. The geometries order in the MARC21/XML output correspond to their order in the collection.

Code

```
SELECT ST_AsMARC21('SRID=4326;GEOMETRYCOLLECTION(POLYGON((13.1 52.65,13.516666666666667 ↩
    52.65,13.516666666666667 52.38333333333333,13.1 52.38333333333333,13.1 52.65)),POINT ↩
    (-4.5 54.25))'::geometry,'hdddmm.mmmm');
```

Output

```
                              st_asmarc21
-----------------------------------------------------
<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a">a</subfield>
        <subfield code="d">E01307.0000</subfield>
        <subfield code="e">E01331.0000</subfield>
        <subfield code="f">N05240.0000</subfield>
        <subfield code="g">N05224.0000</subfield>
    </datafield>
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a">a</subfield>
        <subfield code="d">W00430.0000</subfield>
        <subfield code="e">W00430.0000</subfield>
        <subfield code="f">N05415.0000</subfield>
        <subfield code="g">N05415.0000</subfield>
    </datafield>
</record>
```

See Also

ST_GeomFromMARC21

7.9.3.9 ST_AsMVTGeom

ST_AsMVTGeom — Transforms a geometry into the coordinate space of a MVT tile.

Synopsis

geometry **ST_AsMVTGeom**(geometry geom, box2d bounds, integer extent=4096, integer buffer=256, boolean clip_geom=true);

Description

Transforms a geometry into the coordinate space of a MVT (Mapbox Vector Tile) tile, clipping it to the tile bounds if required. The geometry must be in the coordinate system of the target map (using ST_Transform if needed). Commonly this is Web Mercator (SRID:3857).

The function attempts to preserve geometry validity, and corrects it if needed. This may cause the result geometry to collapse to a lower dimension.

The function performs geometry simplification appropriate to the integer tile coordinate space. After transforming to tile coordinates it snaps coordinates to the integer grid, removes repeated points, and removes points that lie on straight lines. It does not perform general-purpose, scale-dependent simplification such as ST_Simplify. To reduce detail before tile conversion, simplify the geometry in map coordinates before calling `ST_AsMVTGeom`.

The rectangular bounds of the tile in the target map coordinate space must be provided, so the geometry can be transformed, and clipped if required. The bounds can be generated using ST_TileEnvelope.

This function is used to convert geometry into the tile coordinate space required by ST_AsMVT.

`geom` is the geometry to transform, in the coordinate system of the target map.

`bounds` is the rectangular bounds of the tile in map coordinate space, with no buffer.

`extent` is the tile extent size in tile coordinate space as defined by the MVT specification. Defaults to 4096.

`buffer` is the buffer size in tile coordinate space for geometry clippig. Defaults to 256.

`clip_geom` is a boolean to control if geometries are clipped or encoded as-is. Defaults to true.

Availability: 2.4.0



Note
From 3.0, Wagyu can be chosen at configure time to clip and validate MVT polygons. This library is faster and produces more correct results than the GEOS default, but it might drop small polygons.

Examples

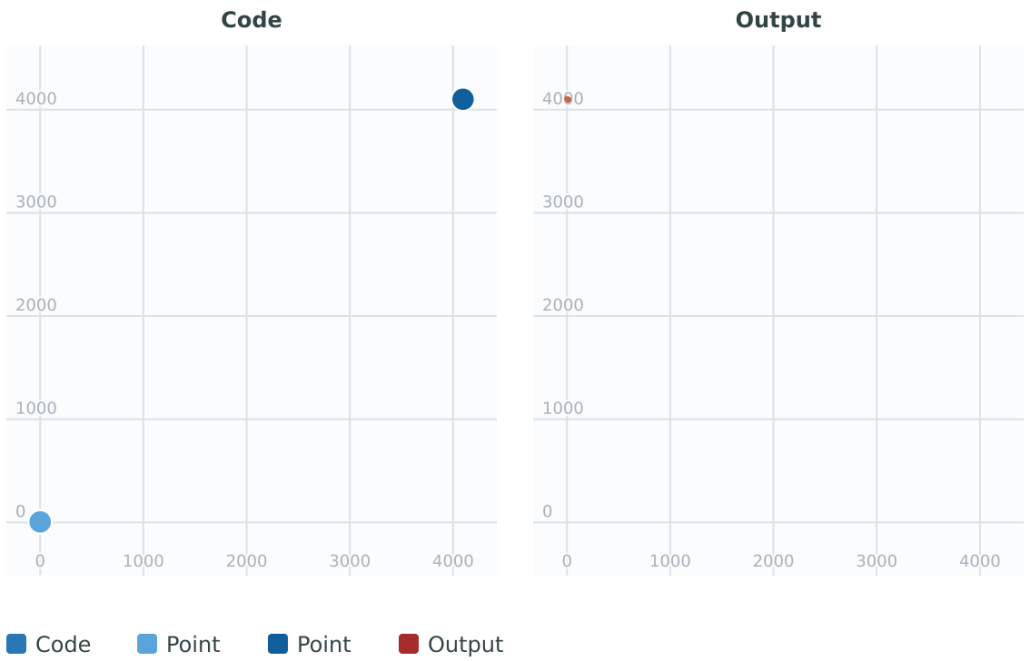
Code

```
SELECT ST_AsMVTGeom('POLYGON ((0 0,10 0,10 5,0 -5,0 0))',
    ST_MakeBox2D(ST_Point(0, 0), ST_Point(4096, 4096)),
    4096, 0, false);
```

Output

```
MULTIPOLYGON(((5 4096,10 4091,10 4096,5 4096)),((5 4096,0 4101,0 4096,5 4096)))
```

Figure



Canonical example for a Web Mercator tile using a computed tile bounds to query and clip geometry. This assumes the data.geom column has srid of 4326.

Code

```
SELECT ST_AsMVTGeom(ST_Transform(geom, 3857 ),
            ST_TileEnvelope(12, 513, 412),
            extent => 4096,
            buffer => 64) AS geom
  FROM data
  WHERE geom && ST_Transform(ST_TileEnvelope(12, 513, 412, margin => (64.0 / 4096)), 4326)
```

See Also

ST_AsMVT, ST_TileEnvelope, PostGIS_Wagyu_Version

7.9.3.10 ST_AsMVT

ST_AsMVT — Aggregate function returning a MVT representation of a set of rows.

Synopsis

bytea **ST_AsMVT**(anyelement set row);
bytea **ST_AsMVT**(anyelement row, text name);
bytea **ST_AsMVT**(anyelement row, text name, integer extent);
bytea **ST_AsMVT**(anyelement row, text name, integer extent, text geom_name);
bytea **ST_AsMVT**(anyelement row, text name, integer extent, text geom_name, text feature_id_name);

Description

An aggregate function which returns a binary Mapbox Vector Tile representation of a set of rows corresponding to a tile layer. The rows must contain a geometry column which will be encoded as a feature geometry. The geometry must be in tile coordinate space and valid as per the MVT specification. ST_AsMVTGeom can be used to transform geometry into tile coordinate space. Other row columns are encoded as feature attributes.

The Mapbox Vector Tile format can store features with varying sets of attributes. To use this capability supply a JSONB column in the row data containing Json objects one level deep. The keys and values in the JSONB values will be encoded as feature attributes.

Tiles with multiple layers can be created by concatenating multiple calls to this function using `||` or `STRING_AGG`.



Important
Do not call with a `GEOMETRYCOLLECTION` as an element in the row. However you can use ST_AsMVTGeom to prepare a geometry collection for inclusion.

`row` row data with at least a geometry column.

`name` is the name of the layer. Default is the string "default".

`extent` is the tile extent in screen space as defined by the specification. Default is 4096.

`geom_name` is the name of the geometry column in the row data. Default is the first geometry column. Note that PostgreSQL by default automatically folds unquoted identifiers to lower case, which means that unless the geometry column is quoted, e.g. `"MyMVTGeom"`, this parameter must be provided as lowercase.

`feature_id_name` is the name of the Feature ID column in the row data. If NULL or negative the Feature ID is not set. The first column matching name and valid type (smallint, integer, bigint) will be used as Feature ID, and any subsequent column will be added as a property. JSON properties are not supported.

Enhanced: 3.0 - added support for Feature ID.

Enhanced: 2.5.0 - added support parallel query.

Availability: 2.4.0

Examples

Code

```
WITH mvtgeom AS
(
  SELECT ST_AsMVTGeom(geom, ST_TileEnvelope(12, 513, 412), extent => 4096, buffer => 64) AS ↩
    geom, name, description
  FROM points_of_interest
  WHERE geom && ST_TileEnvelope(12, 513, 412, margin => (64.0 / 4096))
)
SELECT ST_AsMVT(mvtgeom.*)
FROM mvtgeom;
```

See Also

ST_AsMVTGeom, ST_TileEnvelope

7.9.3.11 ST_AsSVG

ST_AsSVG — Returns SVG path data for a geometry.

Synopsis

text **ST_AsSVG**(geometry geom, integer rel=0, integer maxdecimaldigits=15);
text **ST_AsSVG**(geography geog, integer rel=0, integer maxdecimaldigits=15);

Description

Return the geometry as Scalar Vector Graphics (SVG) path data. Use 1 as second argument to have the path data implemented in terms of relative moves, the default (or 0) uses absolute moves. Third argument may be used to reduce the maximum number of decimal digits used in output (defaults to 15). Point geometries will be rendered as cx/cy when 'rel' arg is 0, x/y when 'rel' is 1. Multipoint geometries are delimited by commas (","), GeometryCollection geometries are delimited by semicolons (";").

For working with PostGIS SVG graphics, checkout pg_svg library which provides plpgsql functions for working with outputs from ST_AsSVG.

Enhanced: 3.4.0 to support all curve types

Changed: 2.0.0 to use default args and support named args

Availability: 1.2.2. Availability: 1.4.0 Changed in PostGIS 1.4.0 to include L command in absolute path to conform to http://www.w3.org/TR/SVG/paths.html#PathDataBNF

 This method supports Circular Strings and Curves.

Examples

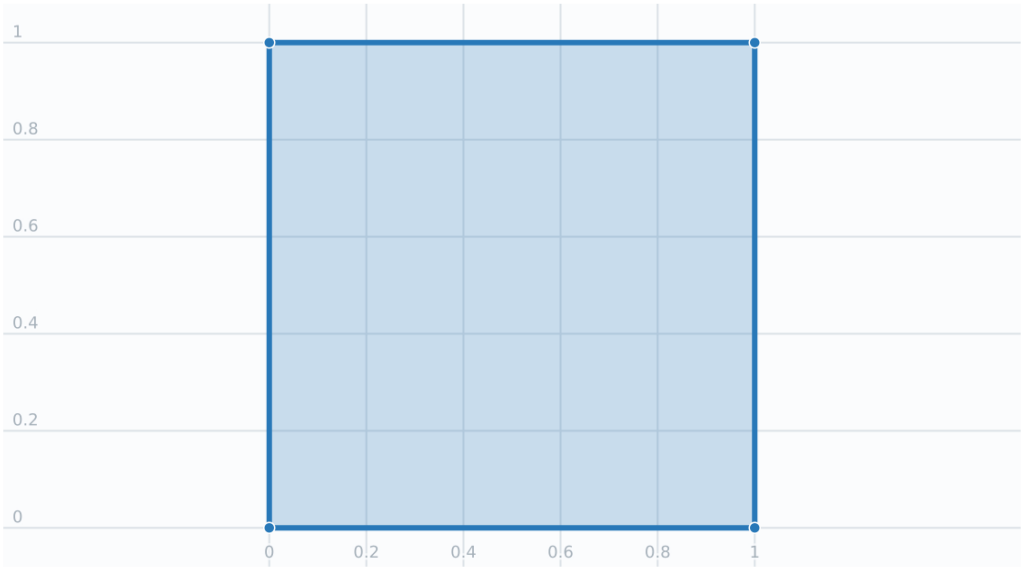
Code

```
SELECT ST_AsSVG('POLYGON((0 0,0 1,1 1,1 0,0 0))'::geometry);
```

Output

```
M 0 0 L 0 -1 1 -1 1 0 Z
```

Figure



Code

Circular string

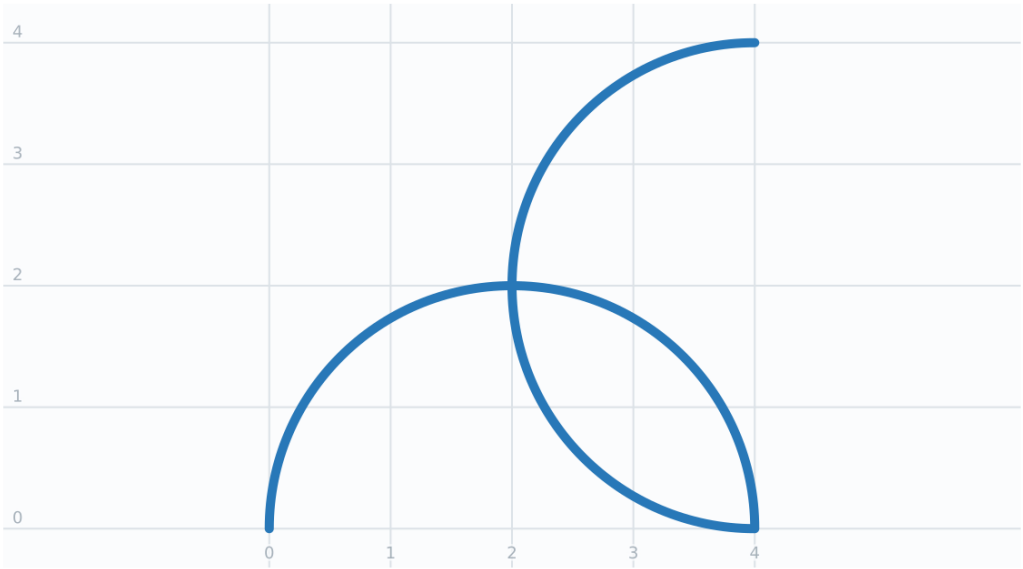
Code

```
SELECT ST_AsSVG('CIRCULARSTRING(-2 0,0 2,2 0,0 2,2 4)'::geometry);
```

Output

```
M -2 0 A 2 2 0 0 1 2 0 A 2 2 0 0 1 2 -4
```

Figure



Code

Multi-curve

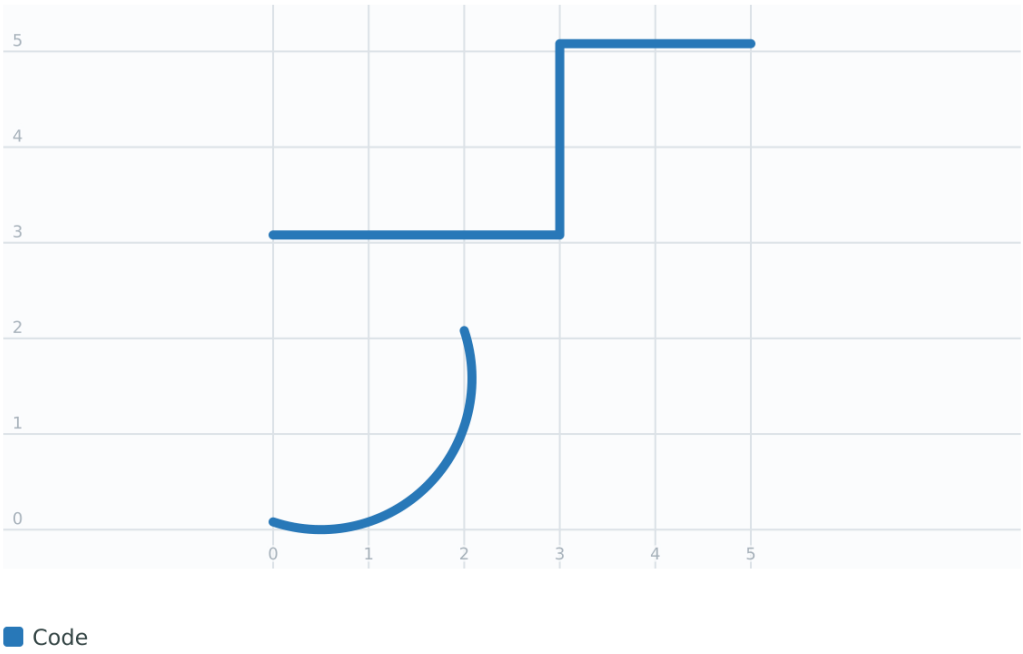
Code

```
SELECT ST_AsSVG('MULTICURVE((5 5,3 5,3 3,0 3),
CIRCULARSTRING(0 0,2 1,2 2))'::geometry, 0, 0);
```

Output

```
M 5 -5 L 3 -5 3 -3 0 -3 M 0 0 A 2 2 0 0 0 2 -2
```

Figure



Multi-surface

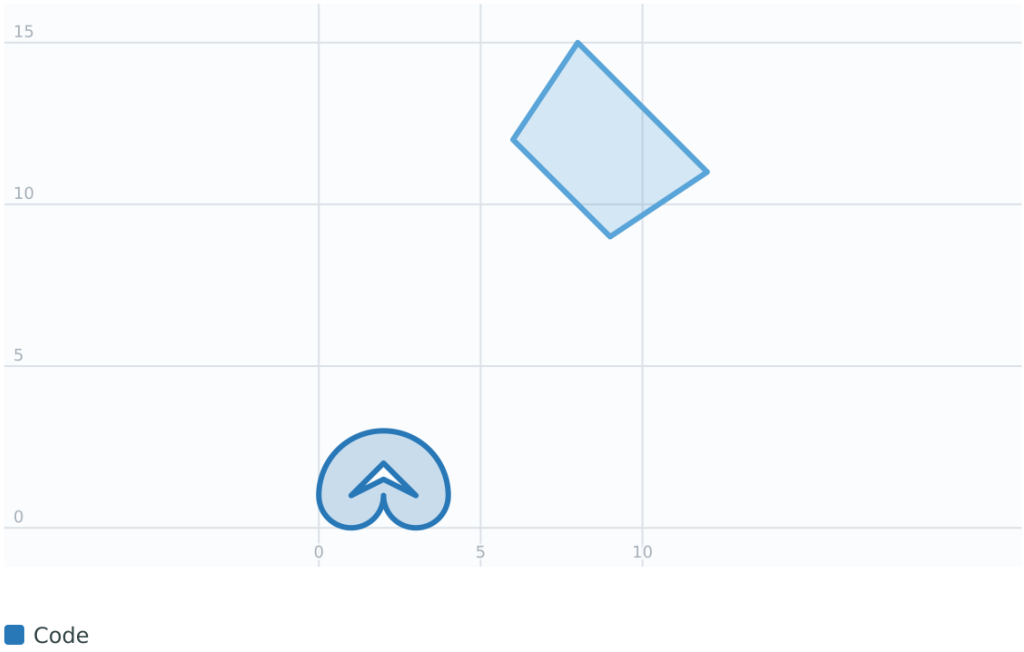
Code

```
SELECT ST_AsSVG('MULTISURFACE(
CURVEPOLYGON(CIRCULARSTRING(-2 0,-1 -1,0 0,1 -1,2 0,0 2,-2 0),
    (-1 0,0 0.5,1 0,0 1,-1 0)),
((7 8,10 10,6 14,4 11,7 8)))'::geometry, 0, 2);
```

Output

```
M -2 0 A 1 1 0 0 0 0 0 A 1 1 0 0 0 2 0 A 2 2 0 0 0 -2 0 Z
M -1 0 L 0 -0.5 1 0 0 -1 -1 0 Z
M 7 -8 L 10 -10 6 -14 4 -11 Z
```

Figure



7.9.3.12 ST_AsTWKB

ST_AsTWKB — Returns the geometry as TWKB, aka "Tiny Well-Known Binary"

Synopsis

bytea **ST_AsTWKB**(geometry geom, integer prec=0, integer prec_z=0, integer prec_m=0, boolean with_sizes=false, boolean with_boxes=false);
bytea **ST_AsTWKB**(geometry[] geom, bigint[] ids, integer prec=0, integer prec_z=0, integer prec_m=0, boolean with_sizes=false, boolean with_boxes=false);

Description

Returns the geometry in TWKB (Tiny Well-Known Binary) format. TWKB is a compressed binary format with a focus on minimizing the size of the output.

The decimal digits parameters control how much precision is stored in the output. By default, values are rounded to the nearest unit before encoding. If you want to transfer more precision, increase the number. For example, a value of 1 implies that the first digit to the right of the decimal point will be preserved.

The sizes and bounding boxes parameters control whether optional information about the encoded length of the object and the bounds of the object are included in the output. By default they are not. Do not turn them on unless your client software has a use for them, as they just use up space (and saving space is the point of TWKB).

The array-input form of the function is used to convert a collection of geometries and unique identifiers into a TWKB collection that preserves the identifiers. This is useful for clients that expect to unpack a collection and then access further information about the objects inside. You can create the arrays using the array_agg function. The other parameters operate the same as for the simple form of the function.



Note
The format specification is available online at https://github.com/TWKB/Specification, and code for building a JavaScript client can be found at https://github.com/TWKB/twkb.js.

Enhanced: 2.4.0 memory and speed improvements.

Availability: 2.2.0

Examples

Code

```
SELECT ST_AsTWKB('LINESTRING(1 1,5 5)'::geometry);
```

Output

```
\x02000202020808
```

To create an aggregate TWKB object including identifiers aggregate the desired geometries and objects first, using "array_agg()", then call the appropriate TWKB function.

Code

```
SELECT ST_AsTWKB(array_agg(geom), array_agg(gid)) FROM mytable;
```

Output

```
\x040402020400000202
```

See Also

ST_GeomFromTWKB, ST_AsBinary, ST_AsEWKB, ST_AsEWKT, ST_GeomFromText

7.9.3.13 ST_AsX3D

ST_AsX3D — Returns a Geometry in X3D xml node element format: ISO-IEC-19776-1.2-X3DEncodings-XML

Synopsis

text **ST_AsX3D**(geometry g1, integer maxdecimaldigits=15, integer options=0);

Description

Returns a geometry as an X3D xml formatted node element http://www.web3d.org/standards/number/19776-1. If `maxdecimaldigits` (precision) is not specified then defaults to 15.



Note
There are various options for translating PostGIS geometries to X3D since X3D geometry types don't map directly to PostGIS geometry types and some newer X3D types that might be better mappings we have avoided since most rendering tools don't currently support them. These are the mappings we have settled on. Feel free to post a bug ticket if you have thoughts on the idea or ways we can allow people to denote their preferred mappings.
Below is how we currently map PostGIS 2D/3D types to X3D types

The 'options' argument is a bitfield that denotes whether to represent coordinates with the X3D GeoCoordinates geospatial node and whether to flip the x/y axis. By default, `ST_AsX3D` outputs in database form (long,lat or X,Y), but the X3D default of lat/lon, y/x may be preferred.

- 0: X/Y in database order (e.g. long/lat = X,Y is standard database order), default value, and non-spatial coordinates (just regular old Coordinate tag).
- 1: Flip X and Y. If used in conjunction with the GeoCoordinate option switch, then output will be default "latitude_first" and coordinates will be flipped as well.
- 2: Output coordinates in GeoSpatial GeoCoordinates. This option will throw an error if geometries are not in WGS 84 long lat (srid: 4326). This is currently the only GeoCoordinate type supported. Refer to X3D specs specifying a spatial reference system.. Default output will be `GeoCoordinate geoSystem='"GD" "WE" "longitude_first"'`. If you prefer the X3D default of `GeoCoordinate geoSystem='"GD" "WE" "latitude_first"'` use `(2 + 1) = 3`

PostGIS Type	2D X3D Type	3D X3D Type
LINESTRING	LineSet (coordinates are promoted to 3D)	LineSet
MULTILINESTRING	IndexedLineSet (coordinates are promoted to 3D)	IndexedLineSet
MULTIPOINT	Polypoint2D	PointSet
POINT	outputs the space delimited coordinates	outputs the space delimited coordinates
(MULTI) POLYGON, POLYHEDRALSURFACE	IndexedFaceSet (coordinates are promoted to 3D; inner rings currently output as another faceset)	IndexedFaceSet (inner rings currently output as another faceset)
TIN	TriangleSet2D (Not Yet Implemented)	IndexedTriangleSet
CIRCULARSTRING, COMPOUNDCURVE, CURVEPOLYGON, MULTICURVE, MULTISURFACE	linearized, then emitted with the linear geometry mapping	linearized, then emitted with the linear geometry mapping
NURBSCURVE	NurbsCurve with controlPoint, knot, weight, and order fields	NurbsCurve with controlPoint, knot, weight, and order fields



Note

2D geometry support not yet complete. Inner rings currently just drawn as separate polygons. We are working on these.

M coordinates are not represented in X3D output.

Lots of advancements happening in 3D space particularly with X3D Integration with HTML5

There is also a nice open source X3D viewer you can use to view rendered geometries. Free Wrl http://freewrl.sourceforge.net/ binaries available for Mac, Linux, and Windows. Use the FreeWRL_Launcher packaged to view the geometries.

Also check out PostGIS minimalist X3D viewer that utilizes this function and x3dDom html/js open source toolkit.

Availability: 2.0.0: ISO-IEC-19776-1.2-X3DEncodings-XML

Enhanced: 2.2.0: Support for GeoCoordinates and axis (x/y, long/lat) flipping. Look at options for details.

Enhanced: 3.7.0: Support for curve geometries. NURBSCURVE is emitted as X3D NurbsCurve; other curve types are linearized before output.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Create a complete X3D document containing a cube that is viewable in FreeWRL and other X3D viewers.

Code

```
SELECT '<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.web3d.org/specifications/x3d ↩
    -3.0.dtd">
<X3D>
  <Scene>
    <Transform>
      <Shape>
       <Appearance>
            <Material emissiveColor=''0 0 1''/>
       </Appearance> ' ||
       ST_AsX3D(ST_GeomFromEWKT('POLYHEDRALSURFACE( ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )')) ||
      '</Shape>
    </Transform>
  </Scene>
</X3D>' As x3ddoc;
```

Output

```
    x3ddoc
    --------
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D>
  <Scene>
    <Transform>
      <Shape>
       <Appearance>
            <Material emissiveColor='0 0 1'/>
       </Appearance>
       <IndexedFaceSet  coordIndex='0 1 2 3 -1 4 5 6 7 -1 8 9 10 11 -1 12 13 14 15 -1 16 17 18 19 -1 20 21 22 23'>
            <Coordinate point='0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 1 1 0 1 1 1 1 0  ↩
    1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 1 0 1 1' />
      </IndexedFaceSet>
      </Shape>
    </Transform>
  </Scene>
</X3D>
```

PostGIS buildings.

Copy and paste the output of this query to x3d scene viewer and click Show

Code

```
SELECT string_agg('<Shape>' || ST_AsX3D(CG_Extrude(geom, 0,0, i*0.5)) ||
    '<Appearance>
          <Material diffuseColor="' || (0.01*i)::text || ' 0.8 0.2" specularColor="' ||  ↩
    (0.05*i)::text || ' 0 0.5"/>
        </Appearance>
    </Shape>', '')
FROM ST_Subdivide(ST_Letters('PostGIS'),20) WITH ORDINALITY AS f(geom,i);
```

The following preview shows the extruded geometry generated from a simplified version of the same source lettering. It is a geometry preview, not X3D markup; use the preceding `ST_AsX3D` query output in an X3D viewer to inspect the native X3D scene.

Code

```
SELECT
    CG_Extrude(
        ST_SimplifyPreserveTopology(ST_Letters('PostGIS'), 2),
        0, 0, 10
    ) AS buildings;
```

Figure



An Octagon elevated 3 Units and decimal precision of 6.

Code

```
SELECT ST_AsX3D(ST_Translate(ST_Force3D(ST_Buffer(ST_Point(10,10),5, 'quad_segs=2')), 0,0,
    3),
  6) As x3dfrag;
```

Output

```
x3dfrag
--------
<IndexedFaceSet  convex='false' coordIndex='0 1 2 3 4 5 6 7'><Coordinate point='15 10 3 13.535534 6.464466 3 10 5 3  ↩
     6.464466 6.464466 3 5 10 3 6.464466 13.535534 3 10 15 3 13.535534 13.535534 3 ' /></IndexedFaceSet>
```

TIN.

Code

```
SELECT ST_AsX3D(ST_GeomFromEWKT('TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
            )), ((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
            ))
            )')) As x3dfrag;
```

Output

```
    x3dfrag
    --------
<IndexedTriangleSet  index='0 1 2 3 4 5'><Coordinate point='0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 1 0'/></IndexedTriangleSet>
```

Closed multilinestring (the boundary of a polygon with holes).

Code

```
SELECT ST_AsX3D(ST_GeomFromEWKT('MULTILINESTRING((20 0 10,16 -12 10,0 -16 10,-12 -12 10,-20 ↩
    0 10,-12 16 10,0 24 10,16 16 10,20 0 10),
  (12 0 10,8 8 10,0 12 10,-8 8 10,-8 0 10,-8 -4 10,0 -8 10,8 -4 10,12 0 10))')
) As x3dfrag;
```

Output

```
    x3dfrag
    --------
<IndexedLineSet  coordIndex='0 1 2 3 4 5 6 7 0 -1 8 9 10 11 12 13 14 15 8'>
    <Coordinate point='20 0 10 16 -12 10 0 -16 10 -12 -12 10 -20 0 10 -12 16 10 0 24 10 16 16 10 12 0 10 8 8 10 0 12 10 ↩
      -8 8 10 -8 0 10 -8 -4 10 0 -8 10 8 -4 10 ' />
 </IndexedLineSet>
```

7.9.3.14 ST_GeoHash

ST_GeoHash — Return a GeoHash representation of the geometry.

Synopsis

text **ST_GeoHash**(geometry geom, integer maxchars=full_precision_of_point);

Description

Computes a GeoHash representation of a geometry. A GeoHash encodes a geographic Point into a text form that is sortable and searchable based on prefixing. A shorter GeoHash is a less precise representation of a point. It can be thought of as a box that contains the point.

Non-point geometry values with non-zero extent can also be mapped to GeoHash codes. The precision of the code depends on the geographic extent of the geometry.

If `maxchars` is not specified, the returned GeoHash code is for the smallest cell containing the input geometry. Points return a GeoHash with 20 characters of precision (about enough to hold the full double precision of the input). Other geometric types may return a GeoHash with less precision, depending on the extent of the geometry. Larger geometries are represented with less precision, smaller ones with more precision. The box determined by the GeoHash code always contains the input feature.

If `maxchars` is specified the returned GeoHash code has at most that many characters. It maps to a (possibly) lower precision representation of the input geometry. For non-points, the starting point of the calculation is the center of the bounding box of the geometry.

Availability: 1.4.0



Note
ST_GeoHash requires input geometry to be in geographic (lon/lat) coordinates.



This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_GeoHash(ST_Point(-126, 48));
```

Output

```
c0w3hf1s70w3hf1s70w3
```

Code

```
SELECT ST_GeoHash(ST_Point(-126, 48), 5);
```

Output

```
c0w3h
```

This example shows that the GeoHash for a line containing a point is a prefix of the point code.

Code

```
SELECT ST_GeoHash('LINESTRING(-126 48,-126.1 48.1)'::geometry);
```

Output

```
c0w3
```

See Also

ST_GeomFromGeoHash, ST_PointFromGeoHash, ST_Box2dFromGeoHash

7.10 Operators

7.10.1 Bounding Box Operators

7.10.1.1 &&

&& — Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.

Synopsis

boolean **&&**(geometry A , geometry B);
boolean **&&**(geography A , geography B);

Description

The && operator returns TRUE if the 2D bounding box of geometry A intersects the 2D bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Enhanced: 2.0.0 support for Polyhedral surfaces was introduced.

Availability: 1.5.0 support for geography was introduced.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

Examples

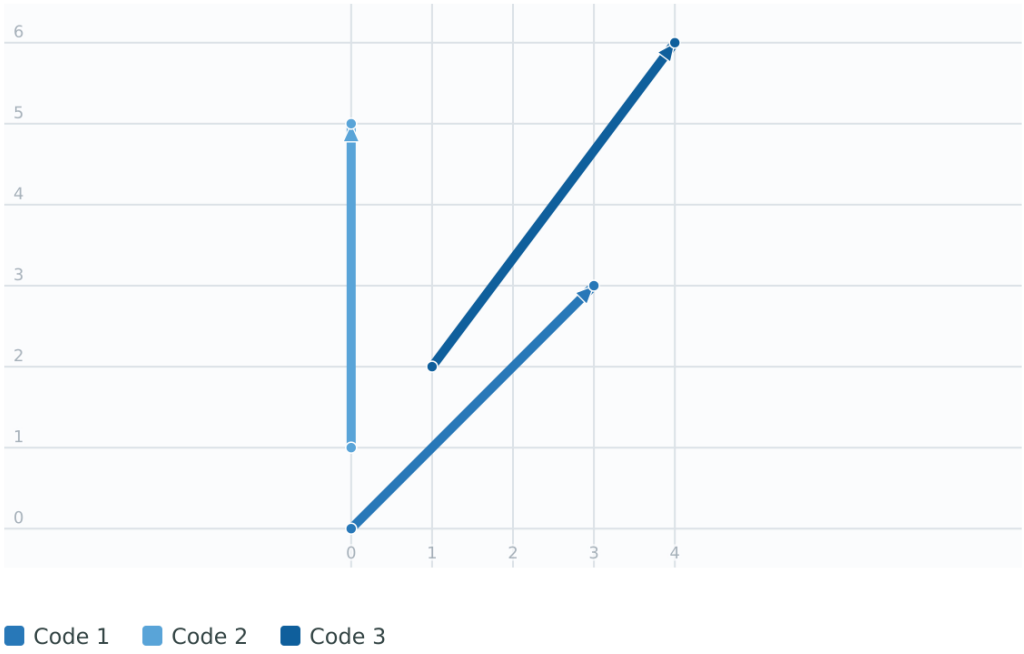
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 && tbl2.column2 AS overlaps
FROM ( VALUES
    (1, 'LINESTRING(0 0, 3 3)'::geometry),
    (2, 'LINESTRING(0 1, 0 5)'::geometry)) AS tbl1,
( VALUES
    (3, 'LINESTRING(1 2, 4 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overlaps
---------+---------+----------
       1 |       3 | t
       2 |       3 | f
(2 rows)
```

Figure



See Also

ST_Intersects, ST_Extent, |&>, &>, &<|, &<, ~, @

7.10.1.2 &&(geometry,box2df)

&&(geometry,box2df) — Returns `TRUE` if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).

Synopsis

boolean **&&**(geometry A , box2df B);

Description

The `&&` operator returns `TRUE` if the cached 2D bounding box of geometry A intersects the 2D bounding box B, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_Point(1,1) && ST_MakeBox2D(ST_Point(0,0), ST_Point(2,2)) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

See Also

&&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.3 &&(box2df,geometry)

&&(box2df,geometry) — Returns `TRUE` if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.

Synopsis

boolean **&&**(box2df A , geometry B);

Description

The `&&` operator returns `TRUE` if the 2D bounding box A intersects the cached 2D bounding box of geometry B, using float precision. This means that if A is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_MakeBox2D(ST_Point(0,0), ST_Point(2,2)) && ST_Point(1,1) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

See Also

&&(geometry,box2df), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.4 &&(box2df,box2df)

&&(box2df,box2df) — Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.

Synopsis

boolean **&&**(box2df A , box2df B);

Description

The && operator returns TRUE if two 2D bounding boxes A and B intersect each other, using float precision. This means that if A (or B) is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_MakeBox2D(ST_Point(0,0), ST_Point(2,2)) && ST_MakeBox2D(ST_Point(1,1), ST_Point ↩
    (3,3)) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

See Also

&&(geometry,box2df), &&(box2df,geometry), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.5 &&&

&&& — Returns TRUE if A's n-D bounding box intersects B's n-D bounding box.

Synopsis

boolean **&&&**(geometry A , geometry B);

Description

The &&& operator returns TRUE if the n-D bounding box of geometry A intersects the n-D bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Availability: 2.0.0

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

 This function supports 3d and will not drop the z-index.

Examples

3D LineStrings.

Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &&& tbl2.column2 AS overlaps_3d,
                        tbl1.column2 && tbl2.column2 AS overlaps_2d
FROM ( VALUES
    (1, 'LINESTRING Z(0 0 1, 3 3 2)'::geometry),
    (2, 'LINESTRING Z(1 2 0, 0 5 -1)'::geometry)) AS tbl1,
( VALUES
    (3, 'LINESTRING Z(1 2 1, 4 6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overlaps_3d | overlaps_2d
---------+---------+-------------+-------------
       1 |       3 | t           | t
       2 |       3 | f           | t
```

Figure



3M LineStrings.

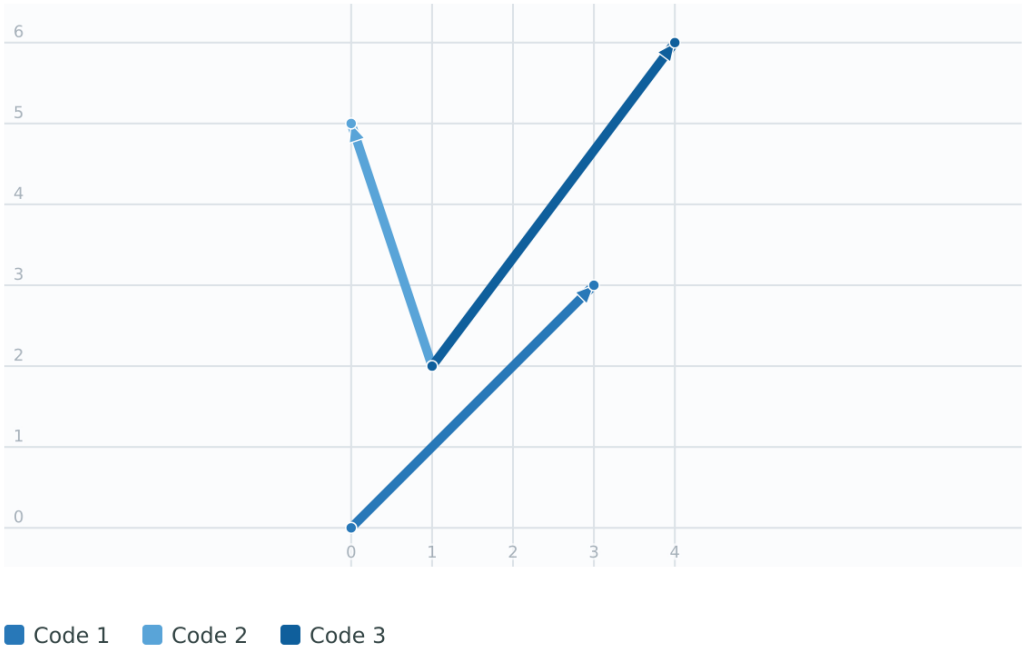
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &&& tbl2.column2 AS overlaps_3zm,
                    tbl1.column2 && tbl2.column2 AS overlaps_2d
FROM ( VALUES
    (1, 'LINESTRING M(0 0 1, 3 3 2)'::geometry),
    (2, 'LINESTRING M(1 2 0, 0 5 -1)'::geometry)) AS tbl1,
( VALUES
    (3, 'LINESTRING M(1 2 1, 4 6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overlaps_3zm | overlaps_2d
---------+---------+--------------+-------------
       1 |       3 | t            | t
       2 |       3 | f            | t
```

Figure



See Also

&&

7.10.1.6 &&&(geometry,gidx)

&&&(geometry,gidx) — Returns `TRUE` if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).

Synopsis

boolean **&&&**(geometry A , gidx B);

Description

The `&&&` operator returns `TRUE` if the cached n-D bounding box of geometry A intersects the n-D bounding box B, using float precision. This means that if B is a (double precision) box3d, it will be internally converted to a float precision 3D bounding box (GIDX)



Note
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_MakePoint(1,1,1) &&& ST_3DMakeBox(ST_MakePoint(0,0,0), ST_MakePoint(2,2,2)) AS ↩
    overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

See Also

&&&(gidx,geometry), &&&(gidx,gidx)

7.10.1.7 &&&(gidx,geometry)

&&&(gidx,geometry) — Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.

Synopsis

boolean **&&&**(gidx A , geometry B);

Description

The &&& operator returns TRUE if the n-D bounding box A intersects the cached n-D bounding box of geometry B, using float precision. This means that if A is a (double precision) box3d, it will be internally converted to a float precision 3D bounding box (GIDX)



Note
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_3DMakeBox(ST_MakePoint(0,0,0), ST_MakePoint(2,2,2)) &&& ST_MakePoint(1,1,1) AS ↩
    overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

See Also

&&&(geometry,gidx), &&&(gidx,gidx)

7.10.1.8 &&&(gidx,gidx)

&&&(gidx,gidx) — Returns `TRUE` if two n-D float precision bounding boxes (GIDX) intersect each other.

Synopsis

boolean **&&&**(gidx A , gidx B);

Description

The `&&&` operator returns `TRUE` if two n-D bounding boxes A and B intersect each other, using float precision. This means that if A (or B) is a (double precision) box3d, it will be internally converted to a float precision 3D bounding box (GIDX)



Note
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

 This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT ST_3DMakeBox(ST_MakePoint(0,0,0), ST_MakePoint(2,2,2)) &&& ST_3DMakeBox(ST_MakePoint ↩
    (1,1,1), ST_MakePoint(3,3,3)) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

See Also

&&&(geometry,gidx), &&&(gidx,geometry)

7.10.1.9 &<

&< — Returns TRUE if A's bounding box overlaps or is to the left of B's.

Synopsis

boolean **&<**(geometry A , geometry B);

Description

The &< operator returns TRUE if the bounding box of geometry A overlaps or is to the left of the bounding box of geometry B, or more accurately, overlaps or is NOT to the right of the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

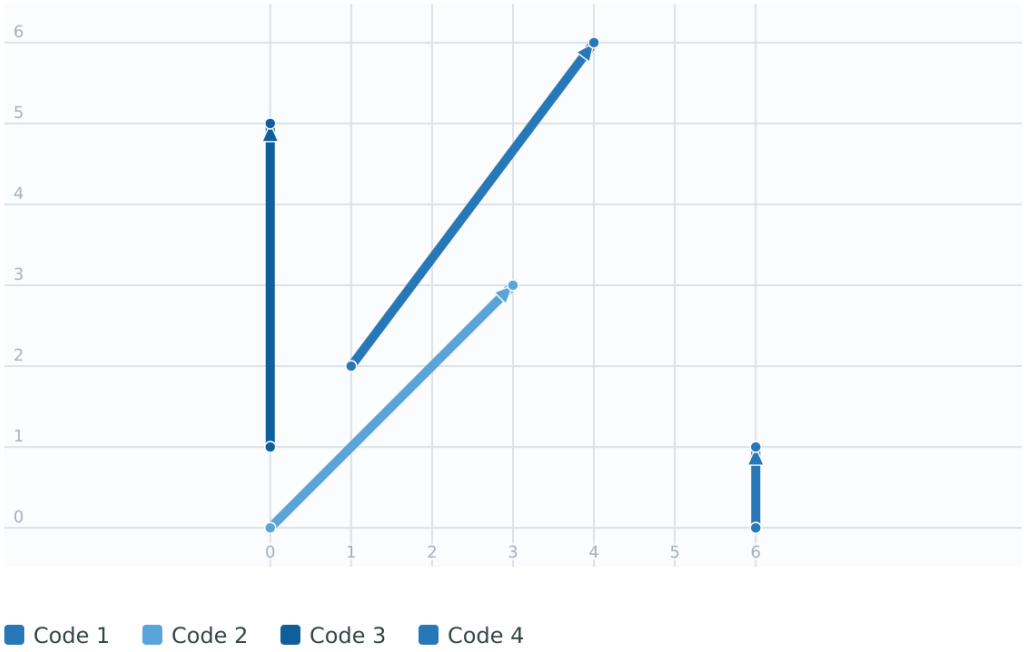
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &< tbl2.column2 AS overleft
FROM
  ( VALUES
    (1, 'LINESTRING(1 2, 4 6)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0, 3 3)'::geometry),
    (3, 'LINESTRING(0 1, 0 5)'::geometry),
    (4, 'LINESTRING(6 0, 6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overleft
---------+---------+----------
       1 |       2 | f
       1 |       3 | f
       1 |       4 | t
(3 rows)
```

Figure



See Also

&&, |&>, &>, &<|

7.10.1.10 &<|

&<| — Returns TRUE if A's bounding box overlaps or is below B's.

Synopsis

boolean **&<|**(geometry A , geometry B);

Description

The &<| operator returns TRUE if the bounding box of geometry A overlaps or is below of the bounding box of geometry B, or more accurately, overlaps or is NOT above the bounding box of geometry B.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

Note
This operand will make use of any indexes that may be available on the geometries.

Examples

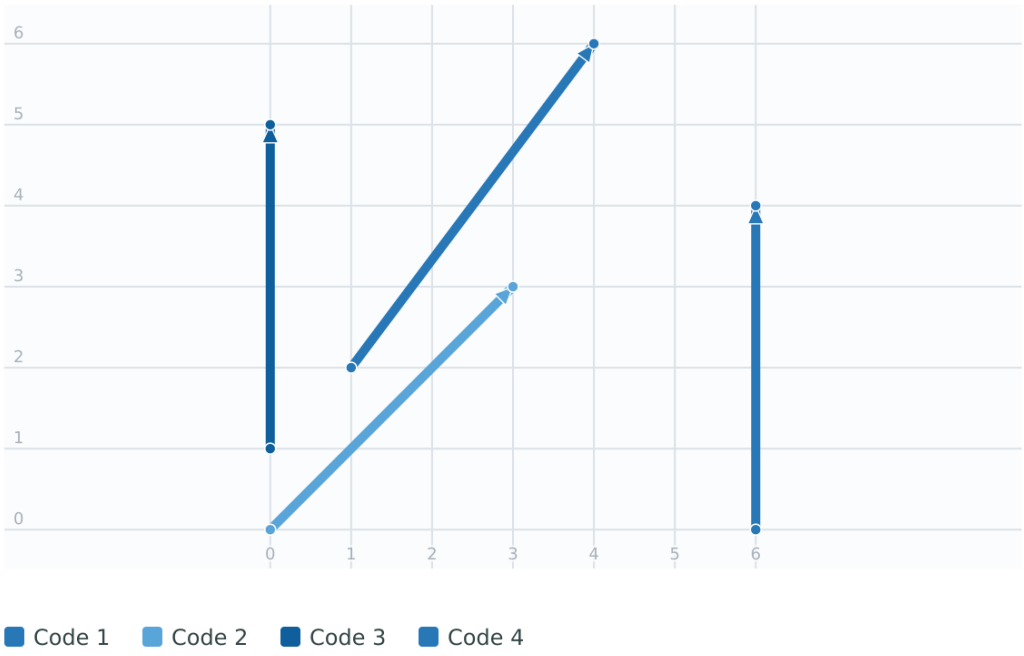
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &<| tbl2.column2 AS overbelow
FROM
  ( VALUES
    (1, 'LINESTRING(6 0, 6 4)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0, 3 3)'::geometry),
    (3, 'LINESTRING(0 1, 0 5)'::geometry),
    (4, 'LINESTRING(1 2, 4 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overbelow
---------+---------+-----------
       1 |       2 | f
       1 |       3 | t
       1 |       4 | t
(3 rows)
```

Figure



See Also

&&, |&>, &>, &<

7.10.1.11 &>

&> — Returns TRUE if A' bounding box overlaps or is to the right of B's.

Synopsis

boolean **&>**(geometry A , geometry B);

Description

The `&>` operator returns `TRUE` if the bounding box of geometry A overlaps or is to the right of the bounding box of geometry B, or more accurately, overlaps or is NOT to the left of the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

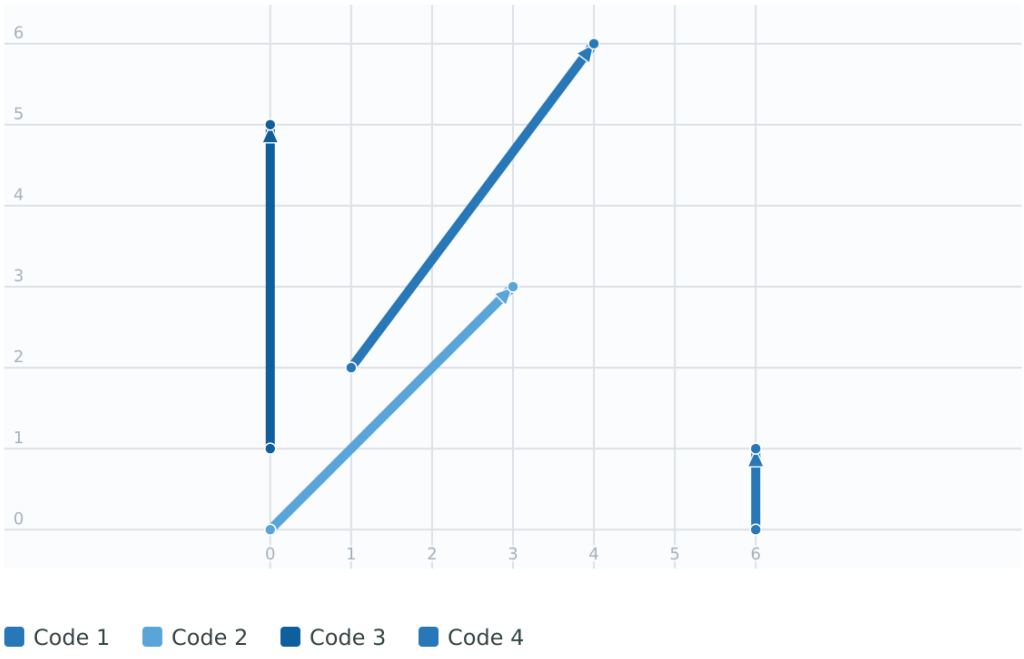
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &> tbl2.column2 AS overright
FROM
  ( VALUES
    (1, 'LINESTRING(1 2,4 6)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0,3 3)'::geometry),
    (3, 'LINESTRING(0 1,0 5)'::geometry),
    (4, 'LINESTRING(6 0,6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overright
---------+---------+-----------
       1 |       2 | t
       1 |       3 | t
       1 |       4 | f
(3 rows)
```

Figure



See Also

&&, |&>, &<|, &<

7.10.1.12 <<

<< — Returns TRUE if A's bounding box is strictly to the left of B's.

Synopsis

boolean **<<**(geometry A , geometry B);

Description

The << operator returns TRUE if the bounding box of geometry A is strictly to the left of the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

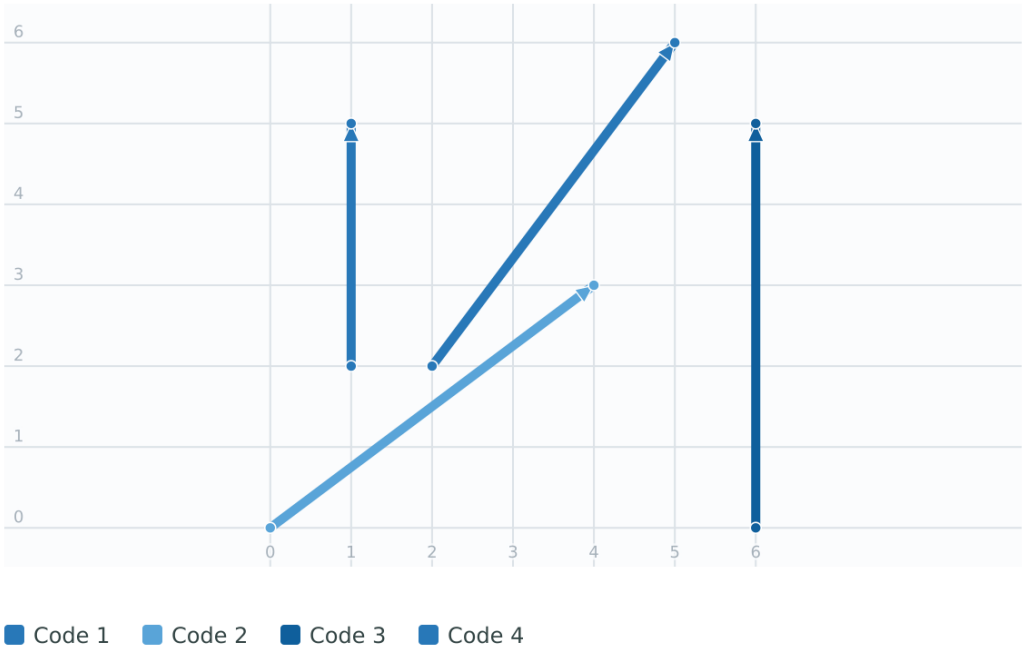
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 << tbl2.column2 AS left
FROM
  ( VALUES
    (1, 'LINESTRING (1 2, 1 5)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0, 4 3)'::geometry),
    (3, 'LINESTRING (6 0, 6 5)'::geometry),
    (4, 'LINESTRING (2 2, 5 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | left
---------+---------+------
       1 |       2 | f
       1 |       3 | t
       1 |       4 | t
(3 rows)
```

Figure



See Also

>>, |>>, <<|

7.10.1.13 <<|

<<| — Returns TRUE if A's bounding box is strictly below B's.

Synopsis

boolean **<<|**(geometry A , geometry B);

Description

The <<| operator returns TRUE if the bounding box of geometry A is strictly below the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

Code

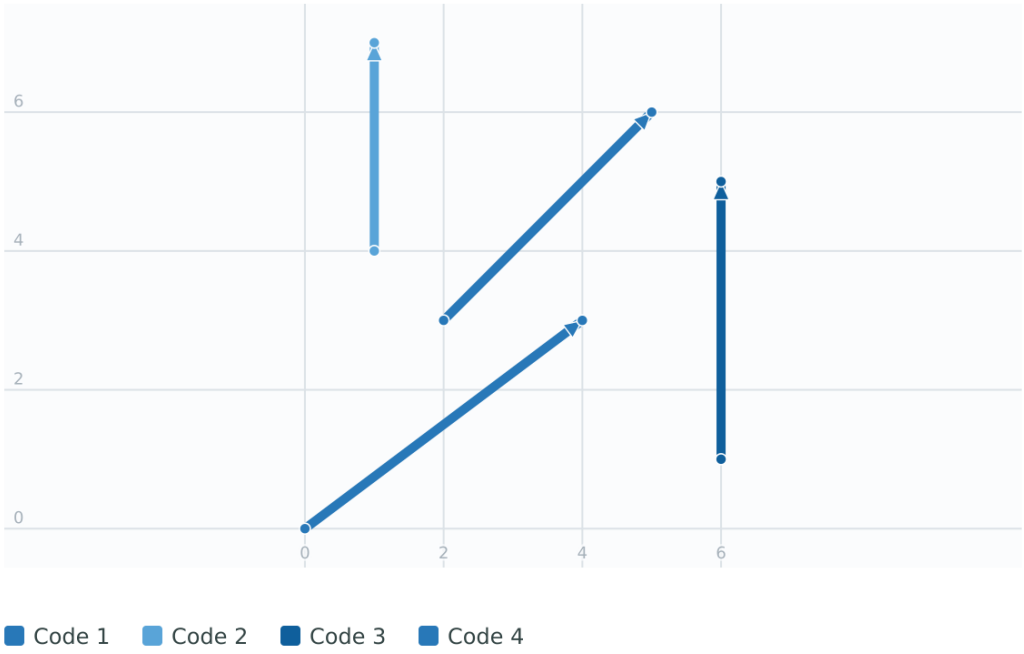
```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 <<| tbl2.column2 AS below
FROM
  ( VALUES
    (1, 'LINESTRING (0 0, 4 3)'::geometry)) AS tbl1,
  ( VALUES
```

```
    (2, 'LINESTRING (1 4, 1 7)'::geometry),
    (3, 'LINESTRING (6 1, 6 5)'::geometry),
    (4, 'LINESTRING (2 3, 5 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | below
---------+---------+-------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



See Also

<<, >>, |>>

7.10.1.14 =

= — Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.

Synopsis

boolean =(geometry A , geometry B);
boolean =(geography A , geography B);

Description

The = operator returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B. PostgreSQL uses the =, <, and > operators defined for geometries to perform internal orderings and comparison of geometries (ie. in a GROUP BY or ORDER BY clause).



Note
Only geometry/geography that are exactly equal in all respects, with the same coordinates, in the same order, are considered equal by this operator. For "spatial equality", that ignores things like coordinate order, and can detect features that cover the same spatial area with different representations, use ST_OrderingEquals or ST_Equals



Caution
This operand will NOT make use of any indexes that may be available on the geometries. For an index assisted exact equality test, combine = with &&.

Changed: 2.4.0, in prior versions this was bounding box equality not a geometric equality. If you need bounding box equality, use ~= instead.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

Examples

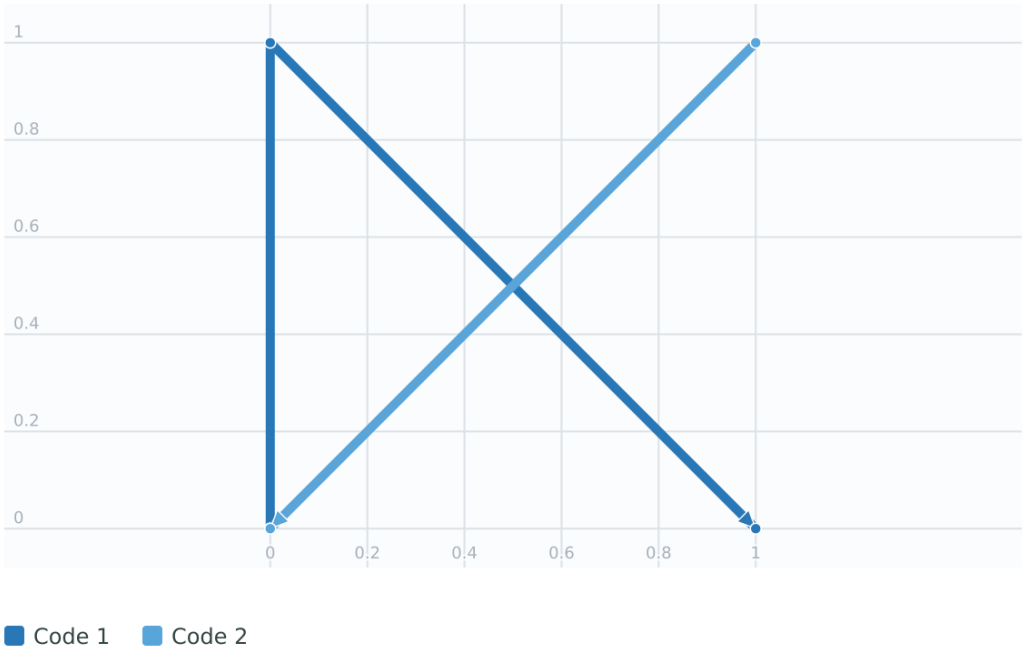
Code

```
SELECT 'LINESTRING(0 0,0 1,1 0)'::geometry = 'LINESTRING(1 1,0 0)'::geometry;
```

Output

```
 ?column?
----------
 f
(1 row)
```

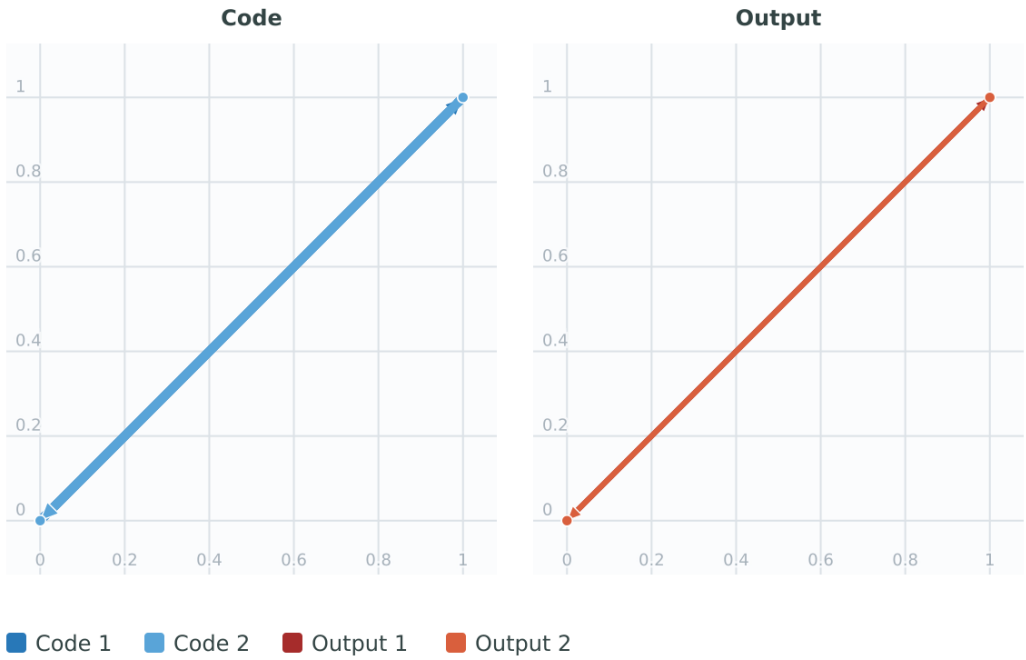
Figure



Code

```
SELECT column1
FROM ( VALUES
    ('LINESTRING(0 0, 1 1)'::geometry),
    ('LINESTRING(1 1, 0 0)'::geometry)) AS foo;
```

Figure

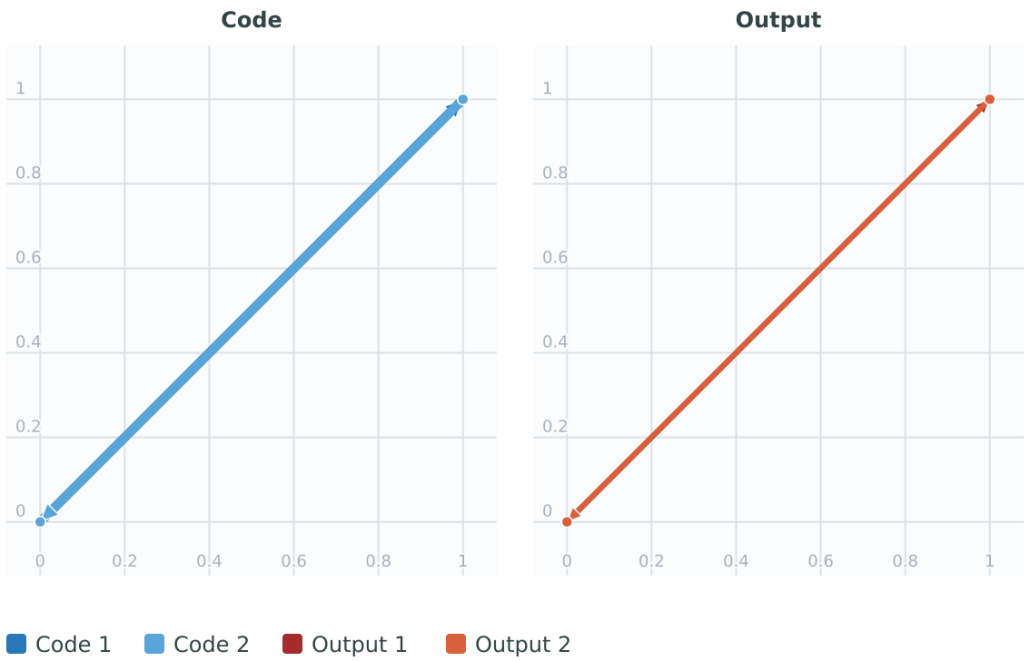


Note: the GROUP BY uses the "=" to compare for geometry equivalency.

Code

```
SELECT column1
FROM ( VALUES
    ('LINESTRING(0 0, 1 1)'::geometry),
    ('LINESTRING(1 1, 0 0)'::geometry)) AS foo
GROUP BY column1;
```

Figure



In versions prior to 2.0, this used to return true.

Code

```
SELECT 'POINT(1707296.37 4820536.77)'::geometry =
   'POINT(1707296.27 4820536.87)'::geometry As pt_intersect;
```

Output

```
 pt_intersect
--------------
 f
(1 row)
```

See Also

ST_Equals, ST_OrderingEquals, ~=

7.10.1.15 >>

>> — Returns TRUE if A's bounding box is strictly to the right of B's.

Synopsis

boolean **>>**(geometry A , geometry B);

Description

The >> operator returns TRUE if the bounding box of geometry A is strictly to the right of the bounding box of geometry B.

Note
This operand will make use of any indexes that may be available on the geometries.

Examples

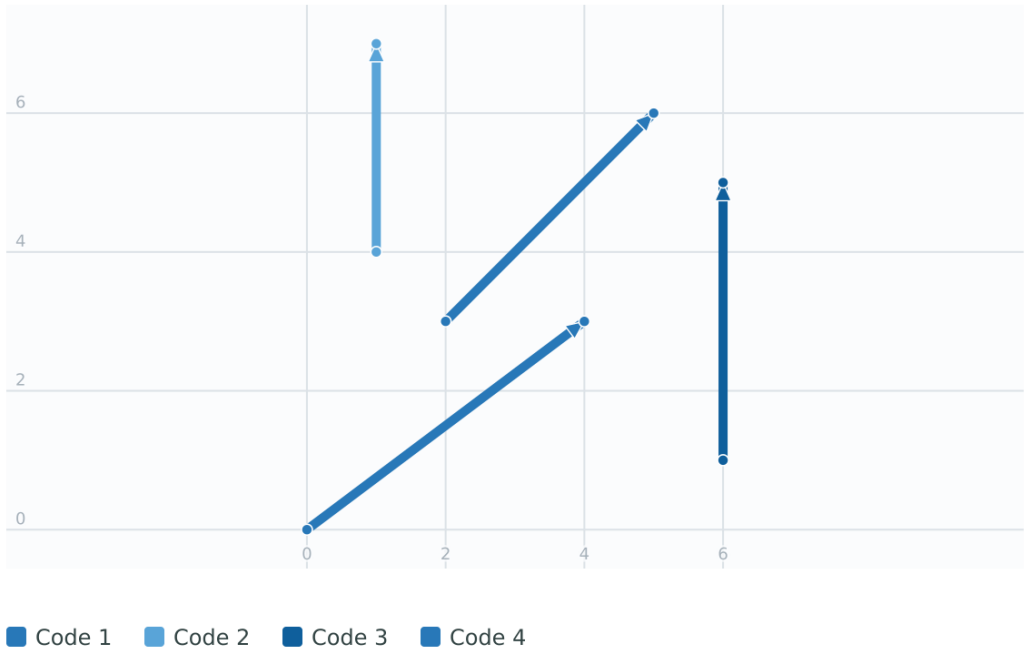
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 >> tbl2.column2 AS right
FROM
  ( VALUES
    (1, 'LINESTRING (2 3,5 6)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (1 4,1 7)'::geometry),
    (3, 'LINESTRING (6 1,6 5)'::geometry),
    (4, 'LINESTRING (0 0,4 3)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | right
---------+---------+-------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



See Also

<<, |>>, <<|

7.10.1.16 @

@ — Returns TRUE if A's bounding box is contained by B's.

Synopsis

boolean @(geometry A , geometry B);

Description

The @ operator returns TRUE if the bounding box of geometry A is completely contained by the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

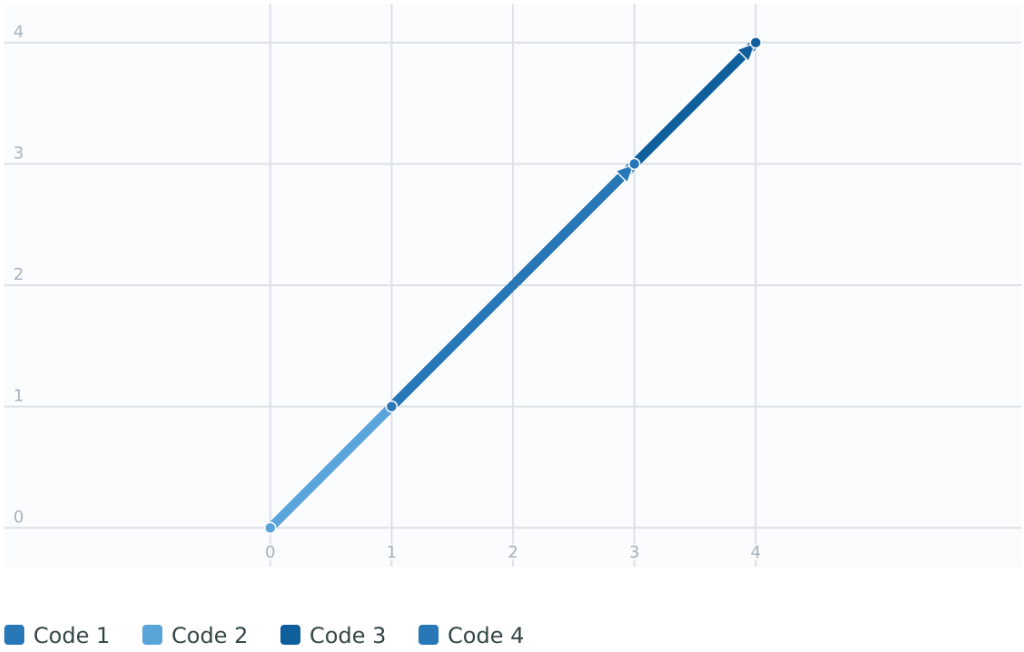
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 @ tbl2.column2 AS contained
FROM
  ( VALUES
    (1, 'LINESTRING (1 1, 3 3)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0, 4 4)'::geometry),
    (3, 'LINESTRING (2 2, 4 4)'::geometry),
    (4, 'LINESTRING (1 1, 3 3)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | contained
---------+---------+-----------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | t
(3 rows)
```

Figure



See Also

~, &&

7.10.1.17 @(geometry,box2df)

@(geometry,box2df) — Returns `TRUE` if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).

Synopsis

boolean @(geometry A , box2df B);

Description

The `@` operator returns `TRUE` if the A geometry's 2D bounding box is contained the 2D bounding box B, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_Buffer('POINT(2 2)'::geometry, 1) @ ST_MakeBox2D(ST_Point(0, 0), ST_Point(5, 5))  ↩
    AS is_contained;
```

Output

```
t
```

See Also

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.18 @(box2df,geometry)

@(box2df,geometry) — Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.

Synopsis

boolean @(box2df A , geometry B);

Description

The @ operator returns TRUE if the 2D bounding box A is contained into the B geometry's 2D bounding box, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_MakeBox2D(ST_Point(2, 2), ST_Point(3, 3)) @ ST_Buffer('POINT(1 1)'::geometry, 10)  ↩
    AS is_contained;
```

Output

```
t
```

See Also

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,box2df)

7.10.1.19 @(box2df,box2df)

@(box2df,box2df) — Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.

Synopsis

boolean @(box2df A , box2df B);

Description

The @ operator returns TRUE if the 2D bounding box A is contained into the 2D bounding box B, using float precision. This means that if A (or B) is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_MakeBox2D(ST_Point(2, 2), ST_Point(3, 3)) @ ST_MakeBox2D(ST_Point(0, 0), ST_Point ↩
    (5, 5)) AS is_contained;
```

Output

```
t
```

See Also

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry)

7.10.1.20 |&>

|&> — Returns TRUE if A's bounding box overlaps or is above B's.

Synopsis

boolean **|&>**(geometry A , geometry B);

Description

The `|&>` operator returns `TRUE` if the bounding box of geometry A overlaps or is above the bounding box of geometry B, or more accurately, overlaps or is NOT below the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

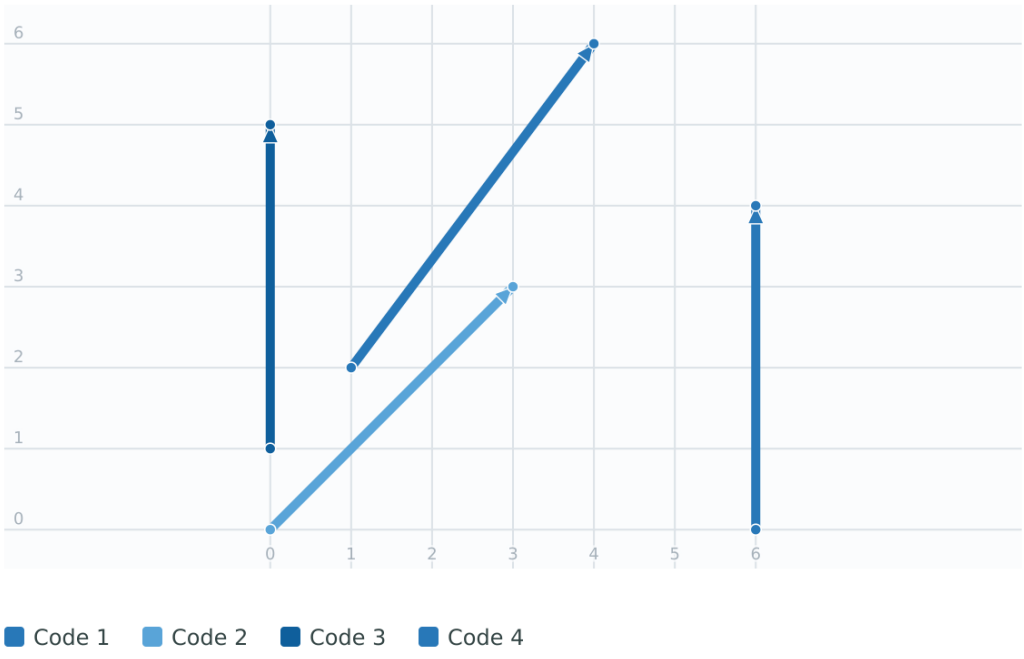
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 |&> tbl2.column2 AS overabove
FROM
  ( VALUES
    (1, 'LINESTRING(6 0,6 4)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0,3 3)'::geometry),
    (3, 'LINESTRING(0 1,0 5)'::geometry),
    (4, 'LINESTRING(1 2,4 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overabove
---------+---------+-----------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



See Also

&&, &>, &<|, &<

7.10.1.21 |>>

|>> — Returns TRUE if A's bounding box is strictly above B's.

Synopsis

boolean **|>>**(geometry A , geometry B);

Description

The |>> operator returns TRUE if the bounding box of geometry A is strictly above the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

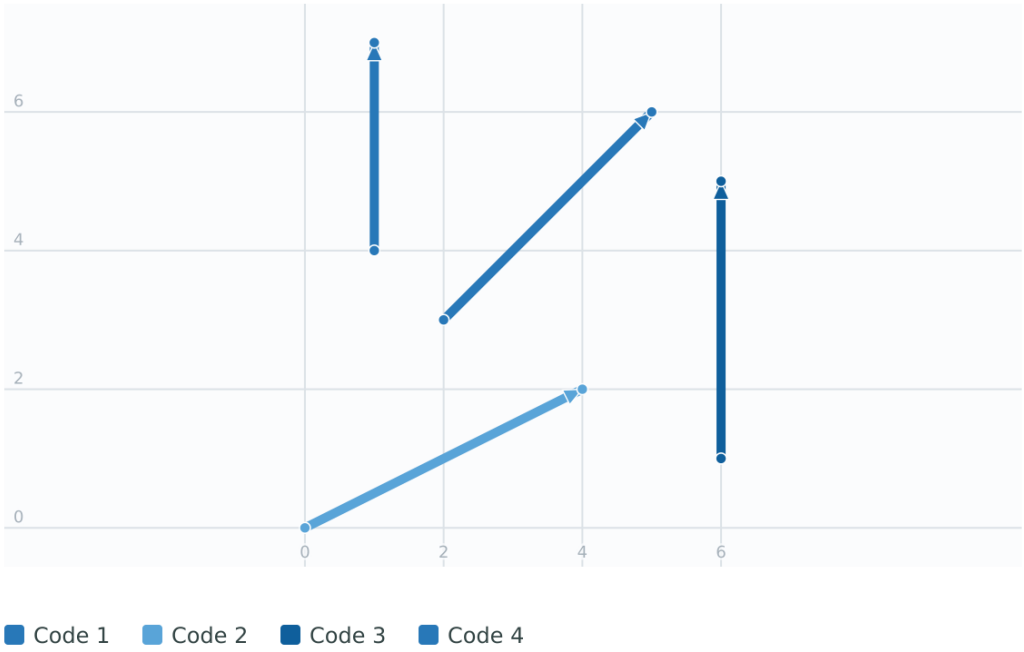
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 |>> tbl2.column2 AS above
FROM
  ( VALUES
    (1, 'LINESTRING (1 4,1 7)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0,4 2)'::geometry),
    (3, 'LINESTRING (6 1,6 5)'::geometry),
    (4, 'LINESTRING (2 3,5 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | above
---------+---------+-------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



See Also

<<, >>, <<|

7.10.1.22 ~

~ — Returns TRUE if A's bounding box contains B's.

Synopsis

boolean ~(geometry A , geometry B);

Description

The ~ operator returns TRUE if the bounding box of geometry A completely contains the bounding box of geometry B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

Code

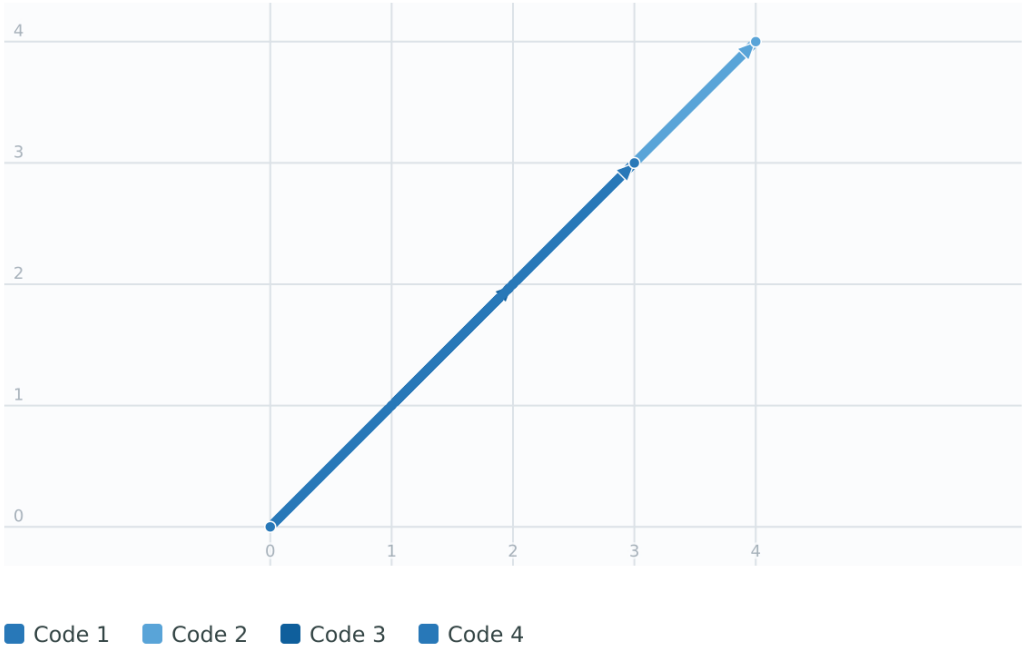
```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 ~ tbl2.column2 AS contains
FROM
  ( VALUES
    (1, 'LINESTRING (0 0,3 3)'::geometry)) AS tbl1,
  ( VALUES
```

```
    (2, 'LINESTRING (0 0,4 4)'::geometry),
    (3, 'LINESTRING (1 1,2 2)'::geometry),
    (4, 'LINESTRING (0 0,3 3)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | contains
---------+---------+----------
       1 |       2 | f
       1 |       3 | t
       1 |       4 | t
(3 rows)
```

Figure



See Also

@, &&

7.10.1.23 ~(geometry,box2df)

~(geometry,box2df) — Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).

Synopsis

boolean ~(geometry A , box2df B);

Description

The ~ operator returns TRUE if the 2D bounding box of a geometry A contains the 2D bounding box B, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_Buffer('POINT(1 1)'::geometry, 10) ~ ST_MakeBox2D(ST_Point(0, 0), ST_Point(2, 2))  ↩
    AS contains;
```

Output

```
t
```

See Also

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.24 ~(box2df,geometry)

~(box2df,geometry) — Returns `TRUE` if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.

Synopsis

boolean ~(box2df A , geometry B);

Description

The ~ operator returns `TRUE` if the 2D bounding box A contains the B geometry's bounding box, using float precision. This means that if A is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_MakeBox2D(ST_Point(0, 0), ST_Point(5, 5)) ~ ST_Buffer('POINT(2 2)'::geometry, 1)  ↩
    AS contains;
```

Output

```
t
```

See Also

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.25 ~(box2df,box2df)

~(box2df,box2df) — Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).

Synopsis

boolean ~(box2df A , box2df B);

Description

The ~ operator returns TRUE if the 2D bounding box A contains the 2D bounding box B, using float precision. This means that if A is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



Note
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

Examples

Code

```
SELECT ST_MakeBox2D(ST_Point(0, 0), ST_Point(5, 5)) ~ ST_MakeBox2D(ST_Point(2, 2), ST_Point ↩
    (3, 3)) AS contains;
```

Output

```
t
```

See Also

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.26 ~=

~= — Returns TRUE if A's bounding box is the same as B's.

Synopsis

boolean **~=**(geometry A , geometry B);

Description

The ~= operator returns TRUE if the bounding box of geometry/geography A is the same as the bounding box of geometry/geography B.



Note
This operand will make use of any indexes that may be available on the geometries.

Availability: 1.5.0 changed behavior


This function supports Polyhedral surfaces.



Warning
This operator has changed behavior in PostGIS 1.5 from testing for actual geometric equality to only checking for bounding box equality. To complicate things it also depends on if you have done a hard or soft upgrade which behavior your database has. To find out which behavior your database has you can run the query below. To check for true equality use ST_OrderingEquals or ST_Equals.

Examples

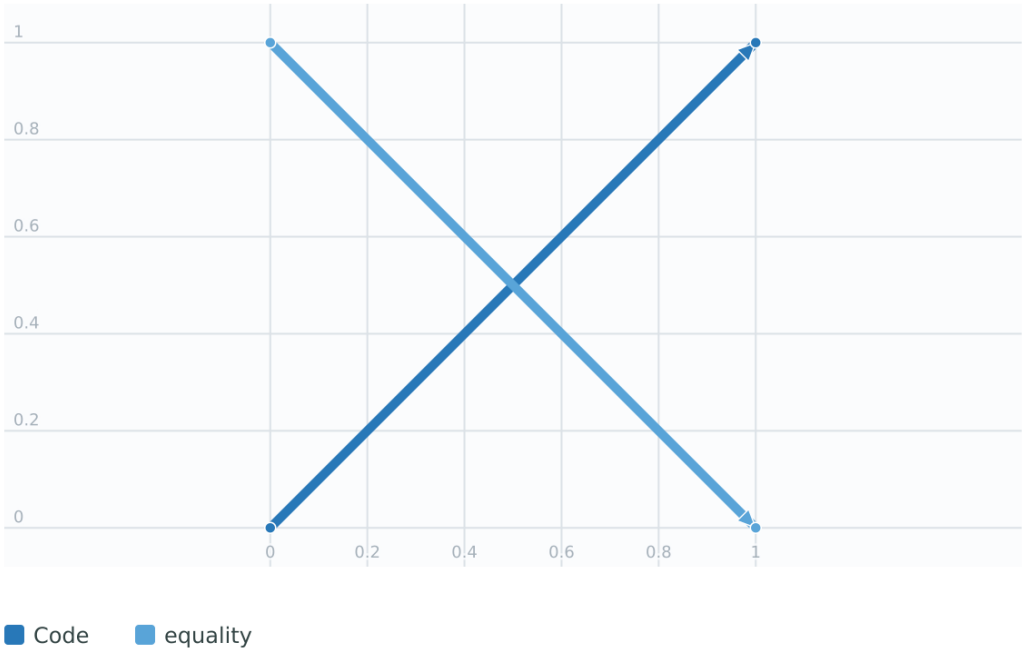
Code

```
SELECT 'LINESTRING(0 0,1 1)'::geometry ~= 'LINESTRING(0 1,1 0)'::geometry AS equality;
```

Output

```
t
```

Figure



See Also

ST_Equals, ST_OrderingEquals, =

7.10.2 Distance Operators

7.10.2.1 <->

<-> — Returns the 2D distance between A and B.

Synopsis

double precision **<->**(geometry A , geometry B);
double precision **<->**(geography A , geography B);

Description

The <-> operator returns the 2D distance between two geometries. Used in the "ORDER BY" clause provides index-assisted nearest-neighbor result sets with true KNN distance search for geometries, and distance on the sphere for geographies.



Note
This operand will make use of 2D GiST indexes that may be available on the geometries. It is different from other operators that use spatial indexes in that the spatial index is only used when the operator is in the ORDER BY clause.



Note
Index only kicks in if one of the geometries is a constant (not in a subquery/cte). e.g. 'SRID=3005;POINT(1011102 450541)'::geometry instead of a.geom

Refer to PostGIS workshop: Nearest-Neighbor Searching for a detailed example.

Enhanced: 2.2.0 -- True KNN ("K nearest neighbor") behavior for geometry and geography. Note for geography KNN is based on sphere rather than spheroid.

Changed: 2.2.0 -- Old hybrid-syntax workarounds may be slower once true KNN is available. See examples below.

Availability: 2.0.0 -- Weak KNN provides nearest neighbors based on geometry centroid distances instead of true distances. Exact results for points, inexact for all other types.

Examples

Code

```
SELECT ST_Distance(geom, 'SRID=3005;POINT(1011102 450541)'::geometry) as d, edabbr, vaabbr
FROM va2005
ORDER BY d limit 10;
```

Output

```
        d          | edabbr | vaabbr
-------------------+--------+--------
                 0 | ALQ    | 128
 5541.57712511724  | ALQ    | 129A
 5579.67450712005  | ALQ    | 001
  6083.4207708641  | ALQ    | 131
  7691.2205404848  | ALQ    | 003
 7900.75451037313  | ALQ    | 122
 8694.20710669982  | ALQ    | 129B
 9564.24289057111  | ALQ    | 130
  12089.665931705  | ALQ    | 127
 18472.5531479404  | ALQ    | 002
(10 rows)
```

Then the KNN raw answer:

Code

```
SELECT ST_Distance(geom, 'SRID=3005;POINT(1011102 450541)'::geometry) as d,edabbr, vaabbr
FROM va2005
ORDER BY geom <-> 'SRID=3005;POINT(1011102 450541)'::geometry limit 10;
```

Output

```
        d          | edabbr | vaabbr
-------------------+--------+--------
                 0 | ALQ    | 128
 5541.57712511724  | ALQ    | 129A
 5579.67450712005  | ALQ    | 001
  6083.4207708641  | ALQ    | 131
  7691.2205404848  | ALQ    | 003
 7900.75451037313  | ALQ    | 122
 8694.20710669982  | ALQ    | 129B
 9564.24289057111  | ALQ    | 130
  12089.665931705  | ALQ    | 127
 18472.5531479404  | ALQ    | 002
(10 rows)
```

If you run "EXPLAIN ANALYZE" on the two queries you would see a performance improvement for the second.

When a plan cannot produce exact distances directly, use a hybrid query to find the true nearest neighbors. First a CTE query using the index-assisted KNN, then an exact query to get correct ordering:

Code

```
WITH index_query AS (
  SELECT ST_Distance(geom, 'SRID=3005;POINT(1011102 450541)'::geometry) as d,edabbr, vaabbr
    FROM va2005
  ORDER BY geom <-> 'SRID=3005;POINT(1011102 450541)'::geometry LIMIT 100)
  SELECT *
    FROM index_query
  ORDER BY d limit 10;
```

Output

```
        d          | edabbr | vaabbr
-------------------+--------+--------
                 0 | ALQ    | 128
 5541.57712511724 | ALQ    | 129A
 5579.67450712005 | ALQ    | 001
  6083.4207708641 | ALQ    | 131
  7691.2205404848 | ALQ    | 003
 7900.75451037313 | ALQ    | 122
 8694.20710669982 | ALQ    | 129B
 9564.24289057111 | ALQ    | 130
  12089.665931705 | ALQ    | 127
 18472.5531479404 | ALQ    | 002
(10 rows)
```

See Also

ST_DWithin, ST_Distance, <#>

7.10.2.2 |=|

|=| — Returns the distance between A and B trajectories at their closest point of approach.

Synopsis

double precision **|=|**(geometry A , geometry B);

Description

The |=| operator returns the 3D distance between two trajectories (See ST_IsValidTrajectory). This is the same as ST_DistanceCPA but as an operator it can be used for doing nearest neighbor searches using an N-dimensional index.



Note
This operand will make use of ND GiST indexes that may be available on the geometries. It is different from other operators that use spatial indexes in that the spatial index is only used when the operator is in the ORDER BY clause.



Note
Index only kicks in if one of the geometries is a constant (not in a subquery/cte). e.g. 'SRID=3005;LINESTRINGM(0 0 0,0 0 1)'::geometry instead of a.geom

Availability: 2.2.0.

Examples

This example orders stored trajectories by their distance from a literal query trajectory.

Code

```
\set qt 'ST_AddMeasure(ST_MakeLine(ST_MakePointM(-350, 300, 0), ST_MakePointM(-410, 490, 0)) ↩
    , 10, 20)'

SELECT track_id, dist FROM (
  SELECT track_id,
    ST_DistanceCPA(tr, :qt) dist
  FROM trajectories
  ORDER BY tr |=| :qt
  LIMIT 5
) foo;
```

Output

```
 track_id        dist
----------+-------------------
      395 | 0.576496831518066
      380 |  5.06797130410151
      390 |  7.72262293958322
      385 |   9.8004461358071
      405 |  10.9534397988433
(5 rows)
```

See Also

ST_DistanceCPA, ST_ClosestPointOfApproach, ST_IsValidTrajectory

7.10.2.3 <#>

<#> — Returns the 2D distance between A and B bounding boxes.

Synopsis

double precision **<#>**(geometry A , geometry B);

Description

The <#> operator returns distance between two floating point bounding boxes, possibly reading them from a spatial index. Useful for doing nearest neighbor **approximate** distance ordering.



Note
This operand will make use of any indexes that may be available on the geometries. It is different from other operators that use spatial indexes in that the spatial index is only used when the operator is in the ORDER BY clause.



Note
Index only kicks in if one of the geometries is a constant e.g. ORDER BY (ST_GeomFromText('POINT(1 2)') <#> geom) instead of g1.geom <#>.

Availability: 2.0.0.

Examples

Code

```
SELECT *
FROM (
SELECT b.tlid, b.mtfcc,
    b.geom <#> ST_GeomFromText('LINESTRING(746149 2948672,745954 2948576,
        745787 2948499,745740 2948468,745712 2948438,
        745690 2948384,745677 2948319)', 2249) As b_dist,
        ST_Distance(b.geom, ST_GeomFromText('LINESTRING(746149 2948672,745954 2948576,
        745787 2948499,745740 2948468,745712 2948438,
        745690 2948384,745677 2948319)', 2249)) As act_dist
    FROM bos_roads As b
    ORDER BY b_dist, b.tlid
    LIMIT 100) As foo
    ORDER BY act_dist, tlid LIMIT 10;
```

Output

```
   tlid    | mtfcc |      b_dist       |     act_dist
-----------+-------+-------------------+------------------
  85732027 | S1400 |                 0 |                0
  85732029 | S1400 |                 0 |                0
  85732031 | S1400 |                 0 |                0
  85734335 | S1400 |                 0 |                0
  85736037 | S1400 |                 0 |                0
 624683742 | S1400 |                 0 | 128.528874268666
  85719343 | S1400 | 260.839270432962 | 260.839270432962
  85741826 | S1400 | 164.759294123275 | 260.839270432962
  85732032 | S1400 |           277.75 | 311.830282365264
  85735592 | S1400 |           222.25 | 311.830282365264
(10 rows)
```

See Also

ST_DWithin, ST_Distance, <->

7.10.2.4 <<->>

<<->> — Returns the n-D distance between the A and B geometries or bounding boxes

Synopsis

double precision **<<->>**(geometry A , geometry B);

Description

The <<->> operator returns the n-D (euclidean) distance between the centroids of the bounding boxes of two geometries. Useful for doing nearest neighbor **approximate** distance ordering.



Note
This operand will make use of n-D GiST indexes that may be available on the geometries. It is different from other operators that use spatial indexes in that the spatial index is only used when the operator is in the ORDER BY clause.



Note
Index only kicks in if one of the geometries is a constant (not in a subquery/cte). e.g. 'SRID=3005;POINT(1011102 450541)'::geometry instead of a.geom

Availability: 2.2.0.

See Also

<->

7.11 Spatial Relationships

7.11.1 Topological Relationships

7.11.1.1 ST_3DIntersects

ST_3DIntersects — Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)

Synopsis

boolean **ST_3DIntersects**(geometry geomA , geometry geomB);

Description

Overlaps, Touches, Within all imply spatial intersection. If any of the aforementioned returns true, then the geometries also spatially intersect. Disjoint implies false for spatial intersection.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.



Note
Because of floating robustness failures, geometries don't always intersect as you'd expect them to after geometric processing. For example the closest point on a linestring to a geometry may not lie on the linestring. For these kind of issues where a distance of a centimeter you want to just consider as intersecting, use ST_3DDWithin.

Changed: 3.0.0 SFCGAL backend removed, GEOS backend supports TINs.

Availability: 2.0.0

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

 This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

Geometry Examples

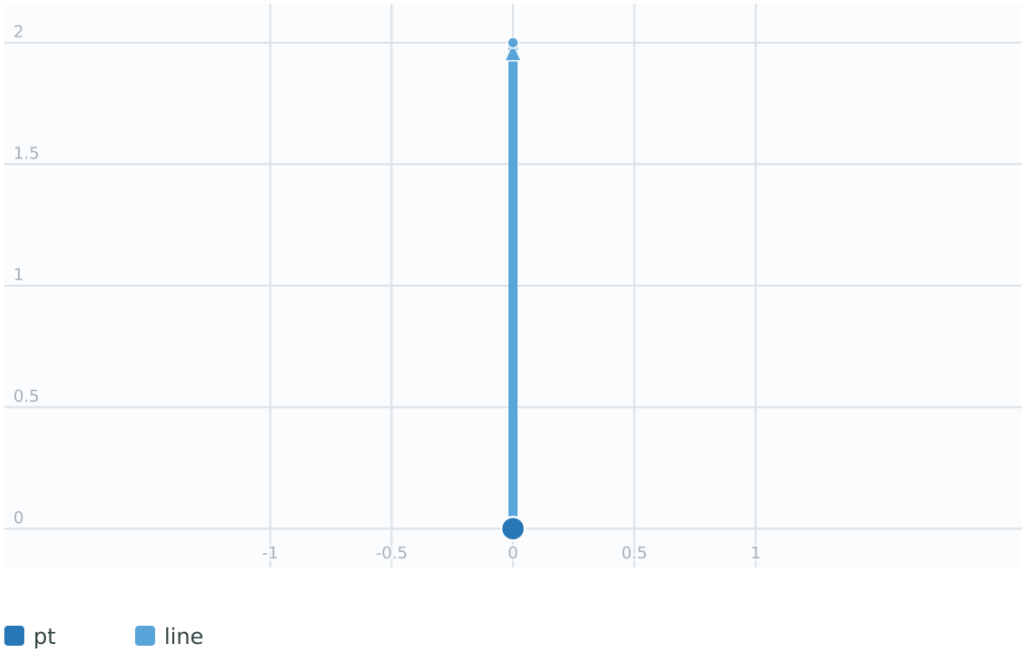
Code

```
SELECT ST_3DIntersects(pt, line), ST_Intersects(pt, line)
FROM (SELECT 'POINT(0 0 2)'::geometry As pt, 'LINESTRING (0 0 1,0 2 3)'::geometry As line)  ↩
    As foo;
```

Output

```
 st_3dintersects | st_intersects
-----------------+---------------
 f               | t
(1 row)
```

Figure



TIN Examples

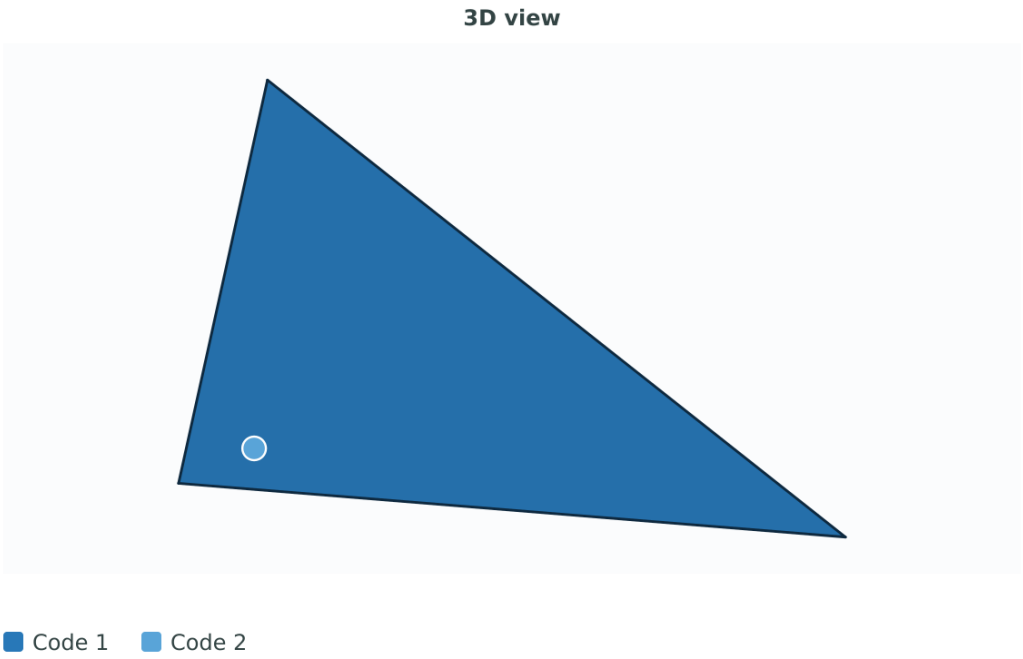
Code

```
SELECT ST_3DIntersects('TIN(((0 0 0,1 0 0,0 1 0,0 0 0)))'::geometry, 'POINT(.1 .1 0)':: ↩
    geometry);
```

Output

```
t
```

Figure



See Also

ST_3DDWithin, ST_Intersects

7.11.1.2 ST_Contains

ST_Contains — Tests if every point of B lies in A, and their interiors have a point in common

Synopsis

boolean **ST_Contains**(geometry geomA, geometry geomB);

Description

Returns TRUE if geometry A contains geometry B. A contains B if and only if all points of B lie inside (i.e. in the interior or boundary of) A (or equivalently, no points of B lie in the exterior of A), and the interiors of A and B have at least one point in common.

In mathematical terms: $ST\_Contains(A, B) \Leftrightarrow (A \bigcap B = B) \wedge (Int(A) \bigcap Int(B) \neq \varnothing)$

The contains relationship is reflexive: every geometry contains itself. (In contrast, in the ST_ContainsProperly predicate a geometry does *not* properly contain itself.) The relationship is antisymmetric: if `ST_Contains(A,B) = true` and `ST_Contains(B,A) = true`, then the two geometries must be topologically equal (`ST_Equals(A,B) = true`).

ST_Contains is the converse of ST_Within. So, `ST_Contains(A,B) = ST_Within(B,A)`.



Note
Because the interiors must have a common point, a subtlety of the definition is that polygons and lines do *not* contain lines and points lying fully in their boundary. For further details see Subtleties of OGC Covers, Contains, Within. The ST_Covers predicate provides a more inclusive relationship.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Contains`.

Performed by the GEOS module

Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



Important
Do not use this function with invalid geometries. You will get unexpected results.

NOTE: this is the "allowable" version that returns a boolean, not an integer.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3 - same as within(geometry B, geometry A)

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.31

Examples

`ST_Contains` returns `TRUE` in the following situations:

A LineString contains a MultiPoint whose points lie in its interior or boundary.

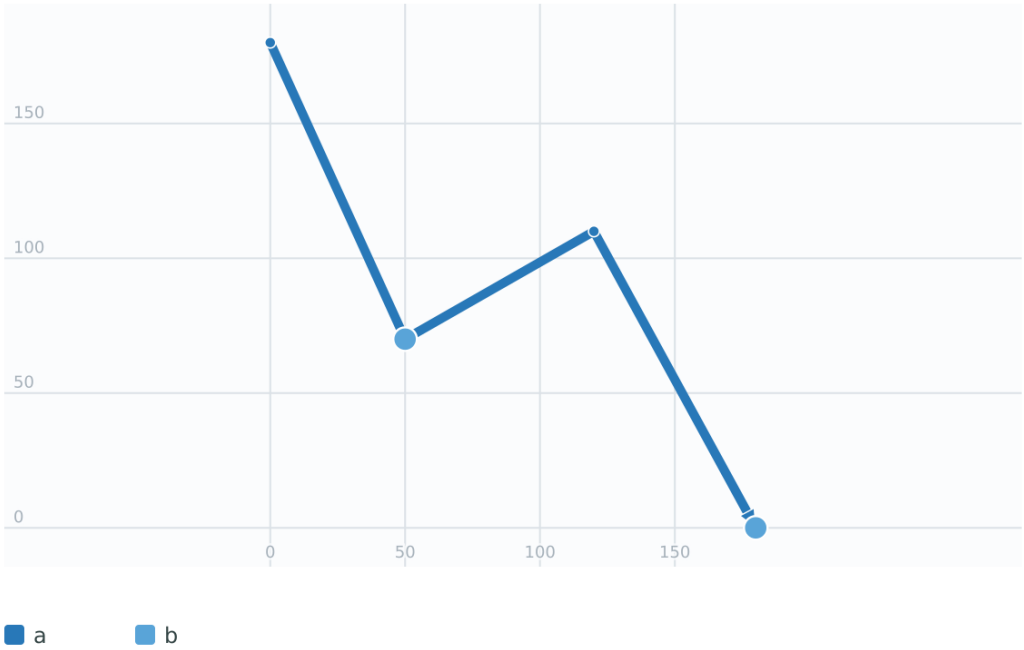
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'MULTIPOINT (60 80,190 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon contains a Point in its interior.

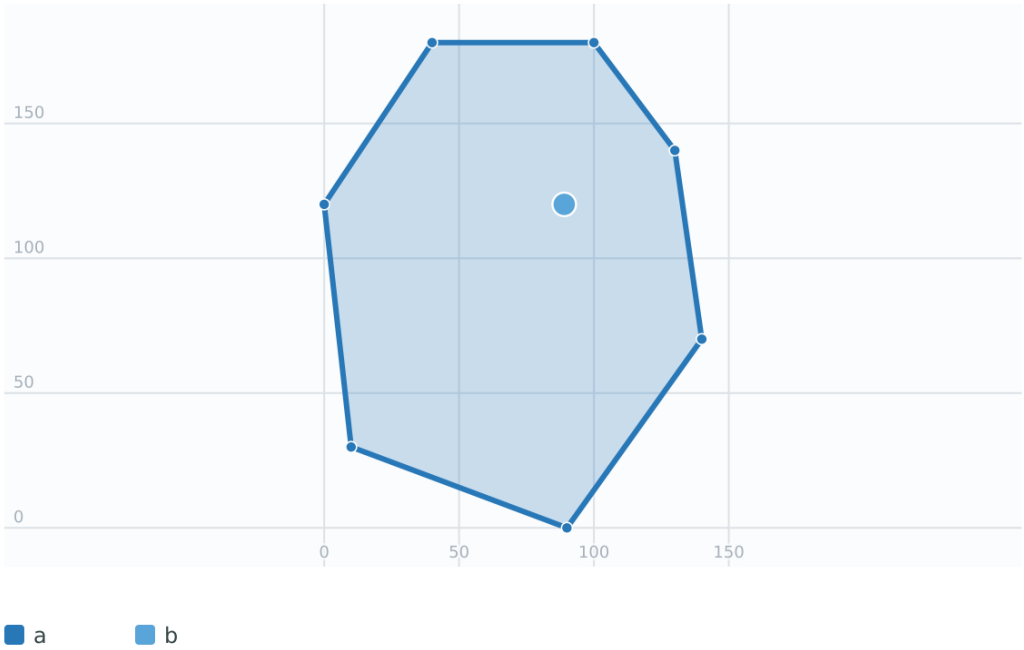
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'POINT (99 130)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon contains a LineString with interior points inside the Polygon.

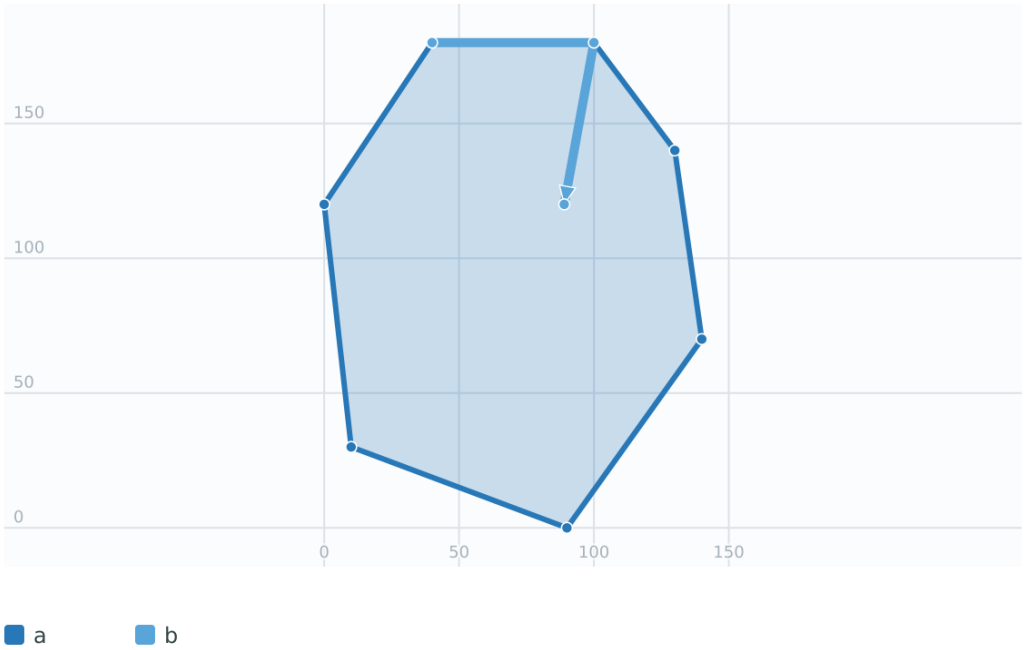
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'LINESTRING (50 190,110 190,99 130)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon contains another Polygon in its interior.

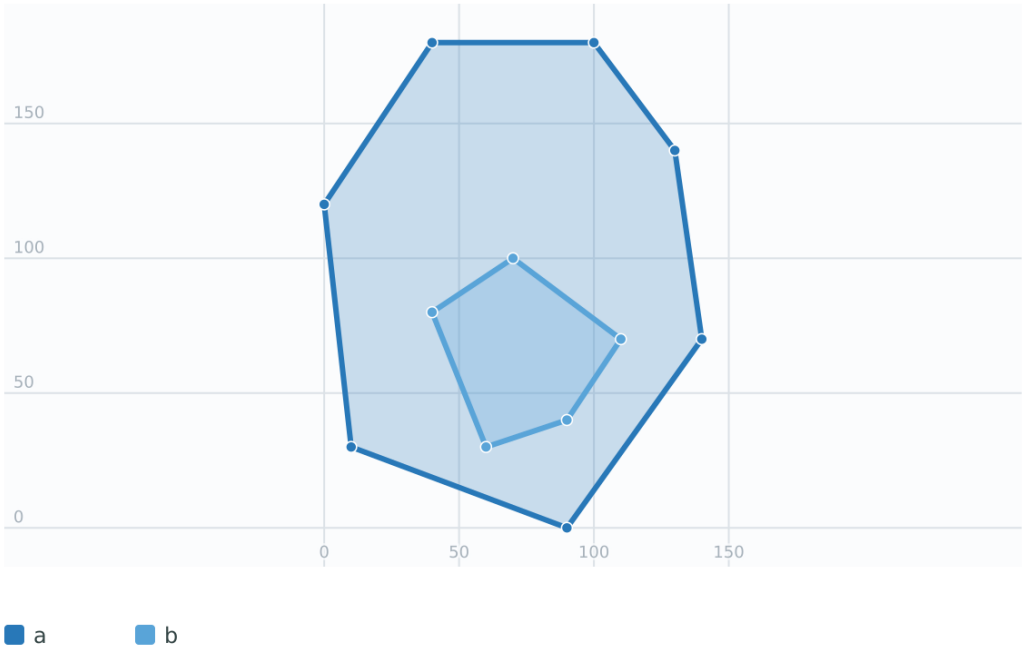
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'POLYGON ((70 40,100 50,120 80,80 110,50 90,70 40))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



`ST_Contains` returns `FALSE` in the following situations:

A Polygon does not contain a MultiPoint when one point lies in a hole.

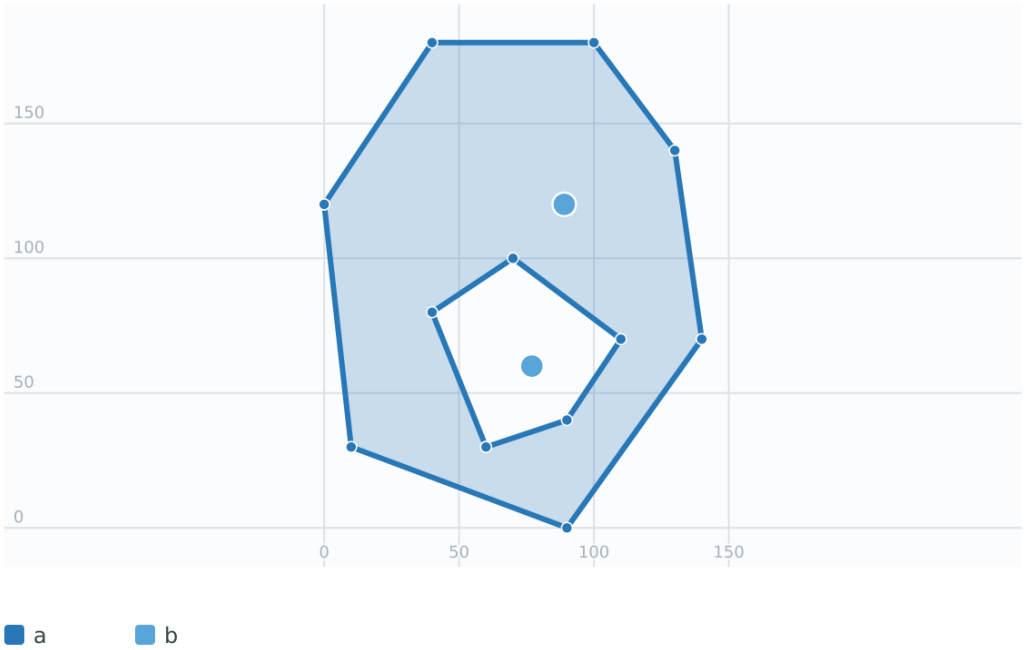
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130),(70 40,100 50,120 ↩
    80,80 110,50 90,70 40))'::geometry AS a,
  'MULTIPOINT (99 130,87 70)'::geometry AS b
) AS example;
```

Output

```
f
```

Figure



A Polygon does not contain a LineString extending outside it.

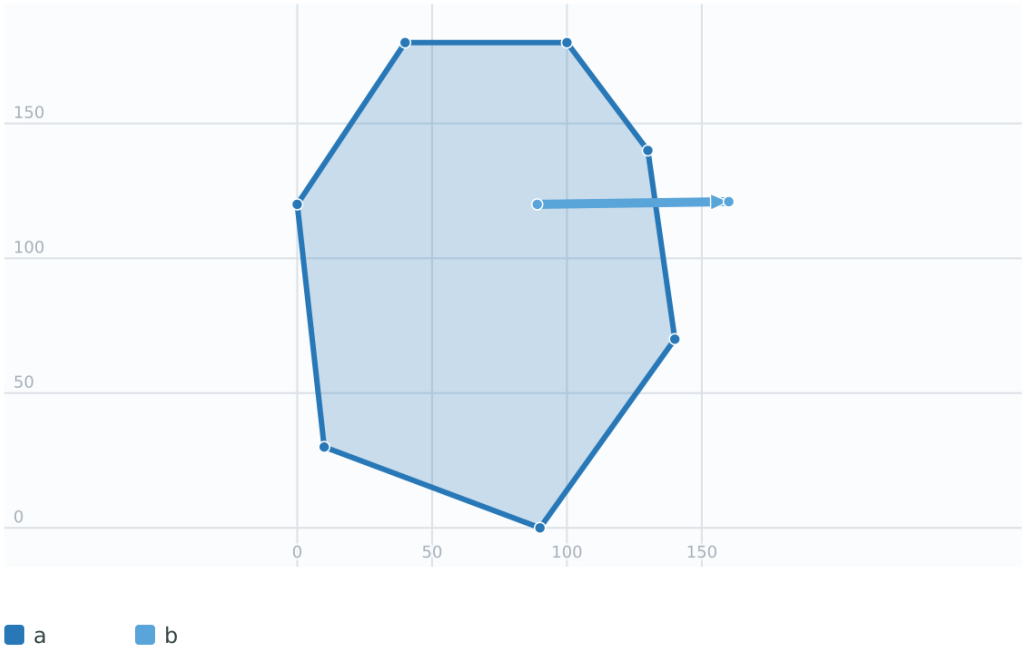
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'LINESTRING (99 130,170 131)'::geometry AS b
) AS example;
```

Output

```
f
```

Figure



Due to the interior intersection condition `ST_Contains` returns `FALSE` in the following situations (whereas `ST_Covers` returns `TRUE`):

A LineString does not contain a Point lying only on its boundary.

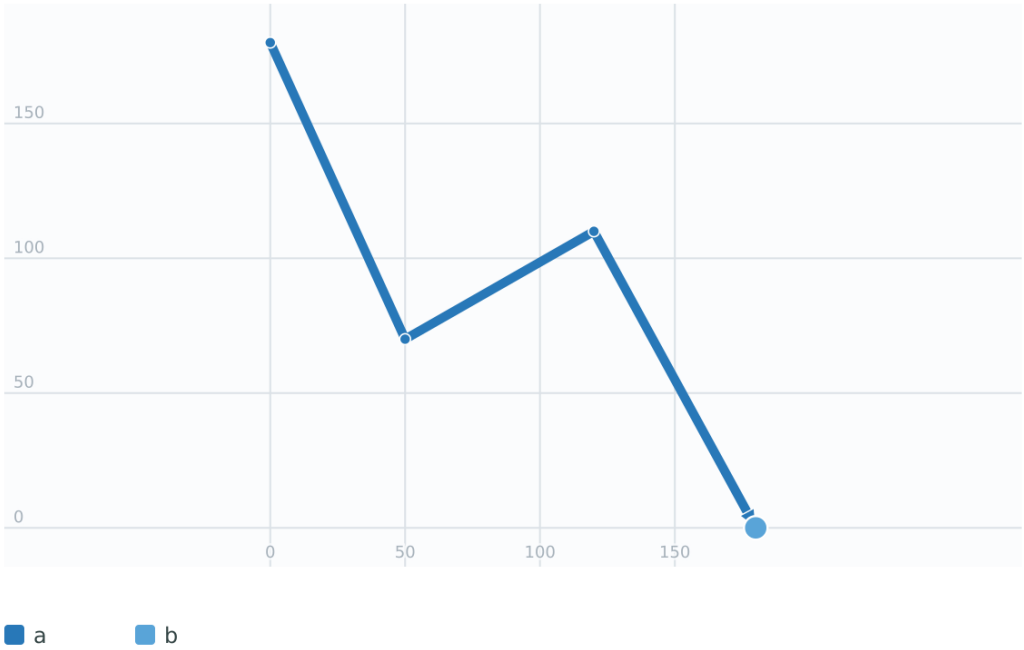
Code

```
SELECT ST_Contains(a, b), ST_Covers(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'POINT (190 10)'::geometry AS b
) AS example;
```

Output

```
f | t
```

Figure



A Polygon does not contain a LineString lying entirely on its boundary.

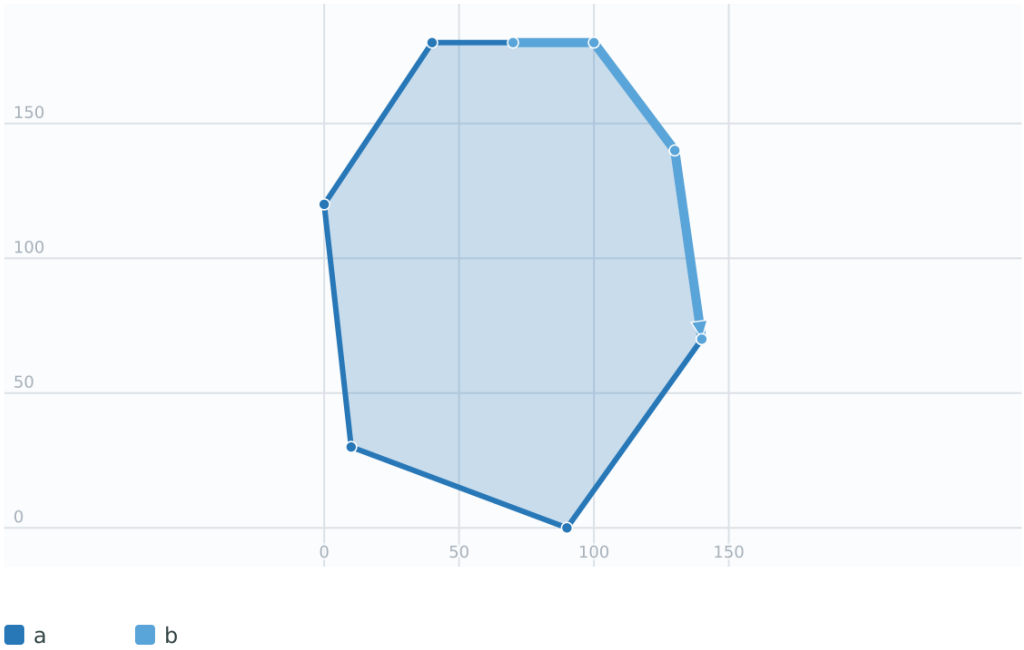
Code

```
SELECT ST_Contains(a, b), ST_Covers(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'LINESTRING (80 190,110 190,140 150,150 80)'::geometry AS b
) AS example;
```

Output

```
f | t
```

Figure



This example tests a circle within another circle.

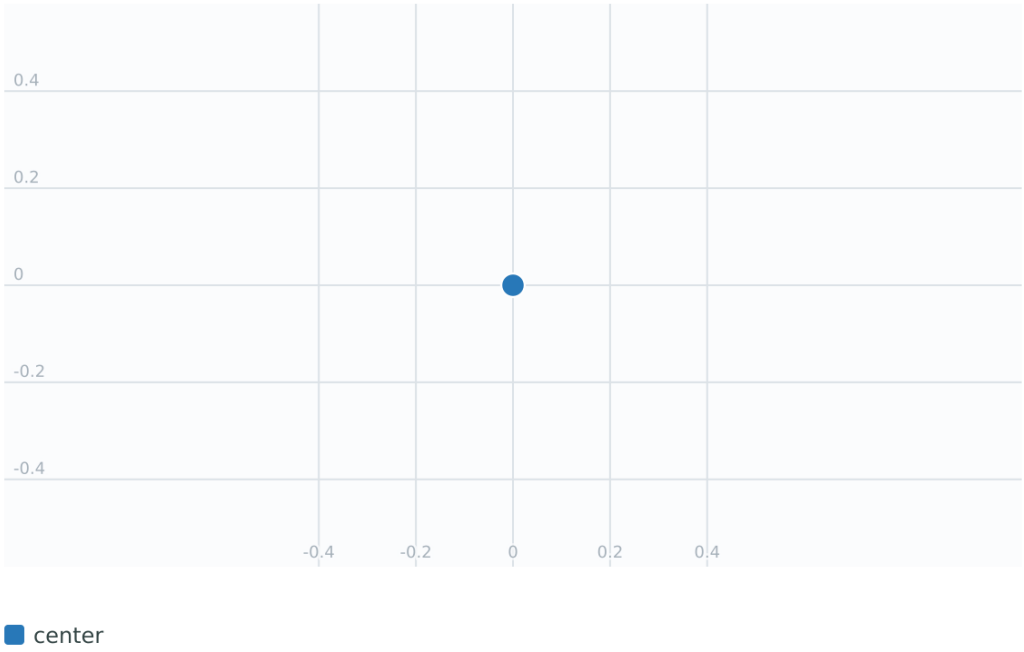
Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_Contains(smallc, bigc) AS smallcontainsbig,
  ST_Contains(bigc, smallc) AS bigcontainssmall,
  ST_Contains(bigc, ST_Union(smallc, bigc)) AS bigcontainsunion,
  ST_Equals(bigc, ST_Union(smallc, bigc)) AS bigisunion,
  ST_Covers(bigc, ST_ExteriorRing(bigc)) AS bigcoversexterior,
  ST_Contains(bigc, ST_ExteriorRing(bigc)) AS bigcontainsexterior
FROM circles;
```

Output

```
-[ RECORD 1 ]-----
smallcontainsbig   | f
bigcontainssmall   | t
bigcontainsunion   | t
bigisunion         | t
bigcoversexterior  | t
bigcontainsexterior | f
```

Figure



This example demonstrates the difference between contains and contains-properly predicates.

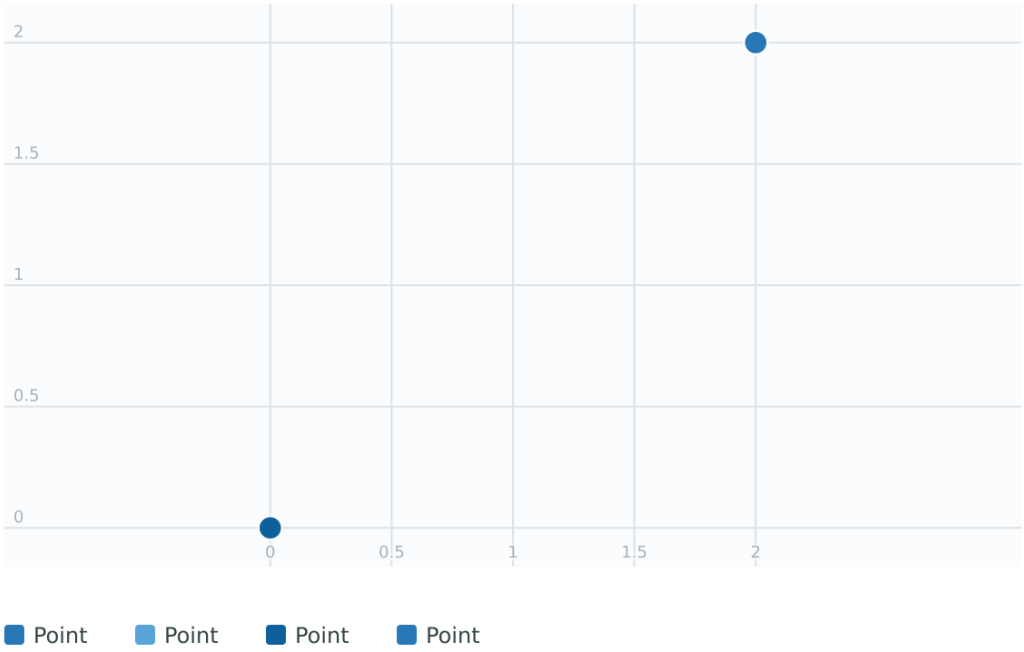
Code

```
SELECT
  ST_GeometryType(geom_a) AS geomtype,
  ST_Contains(geom_a, geom_a) AS acontainsa,
  ST_ContainsProperly(geom_a, geom_a) AS acontainspropa,
  ST_Contains(geom_a, ST_Boundary(geom_a)) AS acontainsba,
  ST_ContainsProperly(geom_a, ST_Boundary(geom_a)) AS acontainspropba
FROM (VALUES
    (ST_Buffer(ST_Point(1, 1), 5, 1)),
    (ST_MakeLine(ST_Point(1, 1), ST_Point(-1, -1))),
    (ST_Point(1, 1))
  ) AS foo(geom_a);
```

Output

```
  geomtype    | acontainsa | acontainspropa | acontainsba | acontainspropba
--------------+------------+----------------+-------------+-----------------
ST_Polygon    | t          | f              | f           | f
ST_LineString | t          | f              | f           | f
ST_Point      | t          | t              | f           | f
```

Figure



See Also

ST_Boundary, ST_ContainsProperly, ST_Covers, ST_CoveredBy, ST_Equals, ST_Within

7.11.1.3 ST_ContainsProperly

ST_ContainsProperly — Tests if every point of B lies in the interior of A

Synopsis

boolean **ST_ContainsProperly**(geometry geomA, geometry geomB);

Description

Returns `true` if every point of B lies in the interior of A (or equivalently, no point of B lies in the the boundary or exterior of A).

In mathematical terms: *ST_ContainsProperly(A, B)* $\Leftrightarrow$ *Int(A)* $\bigcap$ *B = B*

A contains B properly if the DE-9IM Intersection Matrix for the two geometries matches [T**FF*FF*]

A does not properly contain itself, but does contain itself.

A use for this predicate is computing the intersections of a set of geometries with a large polygonal geometry. Since intersection is a fairly slow operation, it can be more efficient to use containsProperly to filter out test geometries which lie fully inside the area. In these cases the intersection is known a priori to be exactly the original test geometry.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_ContainsProperly`.



Note
The advantage of this predicate over ST_Contains and ST_Intersects is that it can be computed more efficiently, with no need to compute topology at individual points.

Performed by the GEOS module.

Availability: 1.4.0

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



Important
Do not use this function with invalid geometries. You will get unexpected results.

Examples

This example tests a circle within another circle.

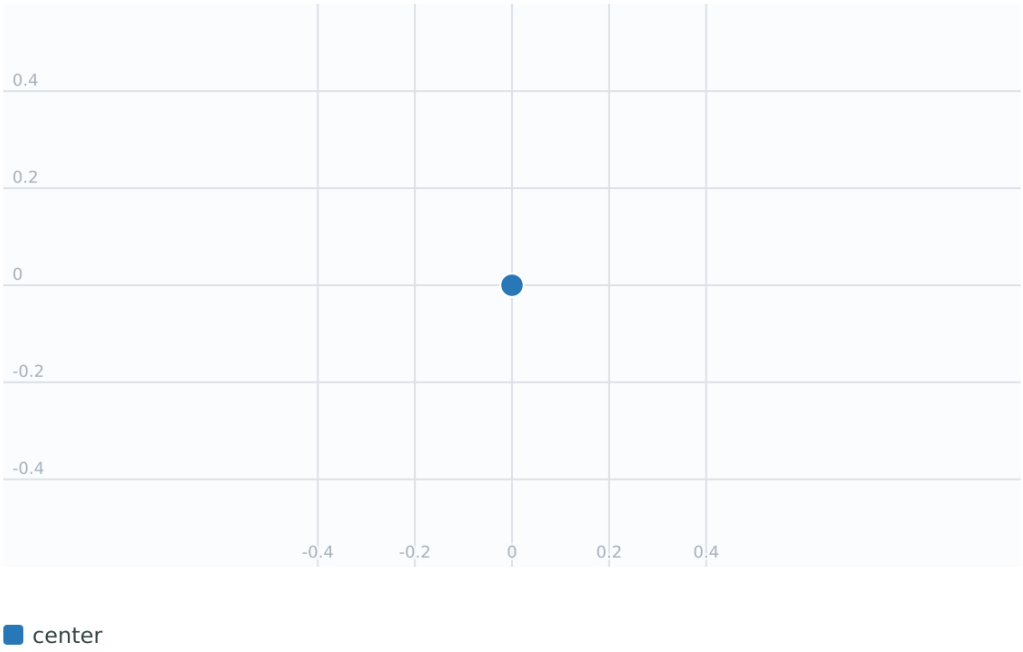
Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_ContainsProperly(smallc, bigc) AS smallcontainspropbig,
  ST_ContainsProperly(bigc, smallc) AS bigcontainspropsmall,
  ST_ContainsProperly(bigc, ST_Union(smallc, bigc)) AS bigcontainspropunion,
  ST_Equals(bigc, ST_Union(smallc, bigc)) AS bigisunion,
  ST_Covers(bigc, ST_ExteriorRing(bigc)) AS bigcoversexterior,
  ST_ContainsProperly(bigc, ST_ExteriorRing(bigc)) AS bigcontainsexterior
FROM circles;
```

Output

```
-[ RECORD 1 ]-----
smallcontainspropbig   | f
bigcontainspropsmall   | t
bigcontainspropunion   | f
bigisunion             | t
bigcoversexterior      | t
bigcontainsexterior    | f
```

Figure



This example demonstrates the difference between contains and contains-properly predicates.

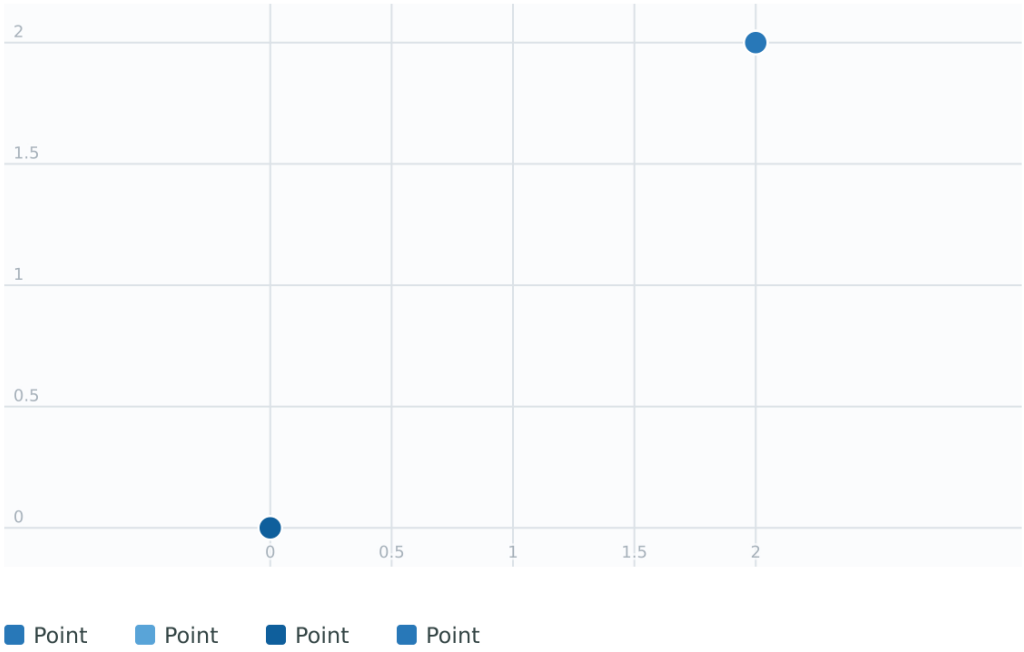
Code

```
WITH input(geom) AS (VALUES
  (ST_Buffer(ST_Point(1, 1), 5, 1)),
  (ST_MakeLine(ST_Point(1, 1), ST_Point(-1, -1))),
  (ST_Point(1, 1))
)
SELECT ST_GeometryType(geom) AS geomtype,
       ST_Contains(geom, geom) AS acontainsa,
       ST_ContainsProperly(geom, geom) AS acontainspropa,
       ST_Contains(geom, ST_Boundary(geom)) AS acontainsba,
       ST_ContainsProperly(geom, ST_Boundary(geom)) AS acontainspropba
FROM input;
```

Output

```
  geomtype    | acontainsa | acontainspropa | acontainsba | acontainspropba
--------------+------------+----------------+-------------+-----------------
ST_Polygon    | t          | f              | f           | f
ST_LineString | t          | f              | f           | f
ST_Point      | t          | t              | f           | f
```

Figure



See Also

ST_GeometryType, ST_Boundary, ST_Contains, ST_Covers, ST_CoveredBy, ST_Equals, ST_Relate, ST_Within

7.11.1.4 ST_CoveredBy

ST_CoveredBy — Tests if every point of A lies in B

Synopsis

boolean **ST_CoveredBy**(geometry geomA, geometry geomB);
boolean **ST_CoveredBy**(geography geogA, geography geogB);

Description

Returns `true` if every point in Geometry/Geography A lies inside (i.e. intersects the interior or boundary of) Geometry/Geography B. Equivalently, tests that no point of A lies outside (in the exterior of) B.

In mathematical terms: *ST_CoveredBy(A, B)* $\Leftrightarrow A \bigcap B = A$

ST_CoveredBy is the converse of ST_Covers. So, `ST_CoveredBy(A,B) = ST_Covers(B,A)`.

Generally this function should be used instead of ST_Within, since it has a simpler definition which does not have the quirk that "boundaries are not within their geometry".



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_CoveredBy`.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



Important
Do not use this function with invalid geometries. You will get unexpected results.

Performed by the GEOS module

Availability: 1.2.2

NOTE: this is the "allowable" version that returns a boolean, not an integer.

Not an OGC standard, but Oracle has it too.

Examples

This example tests a circle covered by another circle.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_CoveredBy(smallc, smallc) AS smallinsmall,
  ST_CoveredBy(smallc, bigc) AS smallcoveredbybig,
  ST_CoveredBy(ST_ExteriorRing(bigc), bigc) AS exteriorcoveredbybig,
  ST_Within(ST_ExteriorRing(bigc), bigc) AS exteriorwithinbig
FROM circles;
```

Output

```
 smallinsmall | smallcoveredbybig | exteriorcoveredbybig | exteriorwithinbig
--------------+-------------------+----------------------+------------------
 t            | t                 | t                    | f
(1 row)
```

Figure



See Also

ST_Contains, ST_Covers, ST_ExteriorRing, ST_Within

7.11.1.5 ST_Covers

ST_Covers — Tests if every point of B lies in A

Synopsis

boolean **ST_Covers**(geometry geomA, geometry geomB);
boolean **ST_Covers**(geography geogpolyA, geography geogpointB);

Description

Returns `true` if every point in Geometry/Geography B lies inside (i.e. intersects the interior or boundary of) Geometry/Geography A. Equivalently, tests that no point of B lies outside (in the exterior of) A.

In mathematical terms: *ST_Covers(A, B)* $\Leftrightarrow A \bigcap B = B$

ST_Covers is the converse of ST_CoveredBy. So, `ST_Covers(A,B) = ST_CoveredBy(B,A)`.

Generally this function should be used instead of ST_Contains, since it has a simpler definition which does not have the quirk that "geometries do not contain their boundary".



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Covers`.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



Important
Do not use this function with invalid geometries. You will get unexpected results.

Performed by the GEOS module

Enhanced: 2.4.0 Support for polygon in polygon and line in polygon added for geography type

Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Availability: 1.5 - support for geography was introduced.

Availability: 1.2.2

NOTE: this is the "allowable" version that returns a boolean, not an integer.

Not an OGC standard, but Oracle has it too.

Examples

Geometry example

This example tests a circle covering another circle.

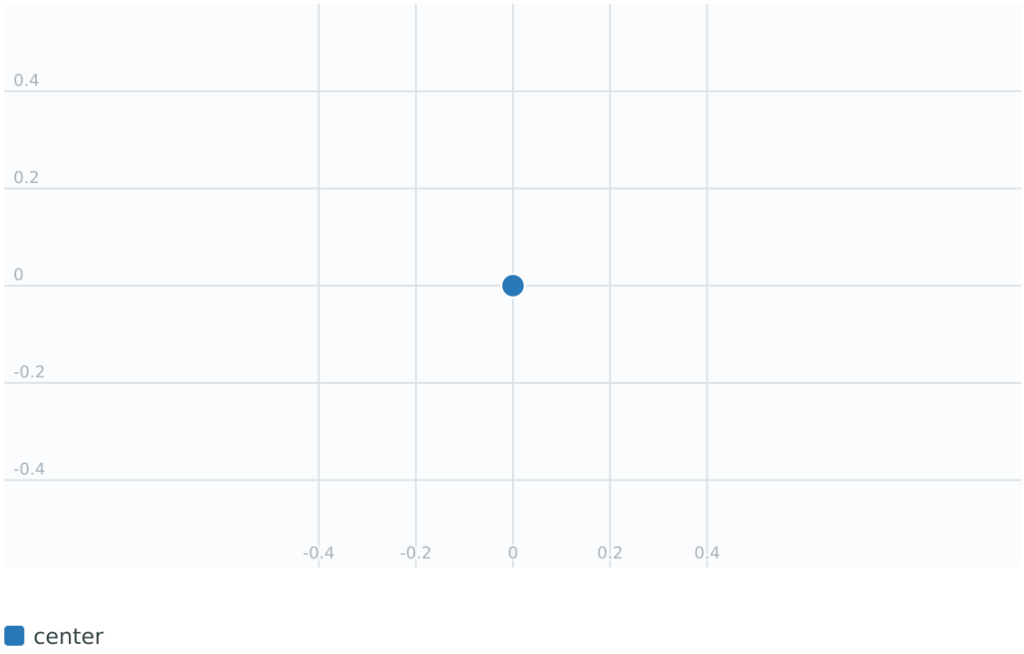
Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_Covers(smallc, smallc) AS smallinsmall,
  ST_Covers(smallc, bigc) AS smallcoversbig,
  ST_Covers(bigc, ST_ExteriorRing(bigc)) AS bigcoversexterior,
  ST_Contains(bigc, ST_ExteriorRing(bigc)) AS bigcontainsexterior
FROM circles;
```

Output

```
 smallinsmall | smallcoversbig | bigcoversexterior | bigcontainsexterior
--------------+----------------+-------------------+---------------------
 t            | f              | t                 | f
(1 row)
```

Figure



Geography example

This example compares a point with a 300-meter buffer against a point and against a 10-meter buffer around that point.

Code

```
SELECT ST_Covers(geog_poly, geog_pt) As poly_covers_pt,
  ST_Covers(ST_Buffer(geog_pt, 10), geog_pt) As buff_10m_covers_cent
  FROM (SELECT ST_Buffer('SRID=4326;POINT(-99.327 31.4821)'::geography, 300) As geog_poly,
        'SRID=4326;POINT(-99.33 31.483)'::geography As geog_pt ) As foo;
```

Output

```
 poly_covers_pt | buff_10m_covers_cent
----------------+---------------------
 f              | t
```

See Also

ST_Contains, ST_CoveredBy, ST_Within

7.11.1.6 ST_Crosses

ST_Crosses — Tests if two geometries have some, but not all, interior points in common

Synopsis

boolean **ST_Crosses**(geometry g1, geometry g2);

Description

Compares two geometry objects and returns `true` if their intersection "spatially crosses"; that is, the geometries have some, but not all interior points in common. The intersection of the interiors of the geometries must be non-empty and must have dimension less than the maximum dimension of the two input geometries, and the intersection of the two geometries must not equal either geometry. Otherwise, it returns `false`. The crosses relation is symmetric and irreflexive.

In mathematical terms: $ST\_Crosses(A, B) \Leftrightarrow (dim(\ Int(A) \bigcap Int(B)\ ) < max(\ dim(\ Int(A)\ ), dim(\ Int(B)\ )\ )) \wedge (A \bigcap B \neq A) \wedge (A \bigcap B \neq B)$

Geometries cross if their DE-9IM Intersection Matrix matches:

- `T*T******` for Point/Line, Point/Area, and Line/Area situations
- `T*****T**` for Line/Point, Area/Point, and Area/Line situations
- `0********` for Line/Line situations
- the result is `false` for Point/Point and Area/Area situations



Note
The OpenGIS Simple Features Specification defines this predicate only for Point/Line, Point/Area, Line/Line, and Line/Area situations. JTS / GEOS extends the definition to apply to Line/Point, Area/Point and Area/Line situations as well. This makes the relation symmetric.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.13.3


This method implements the SQL/MM specification. SQL-MM 3: 5.1.29

Examples

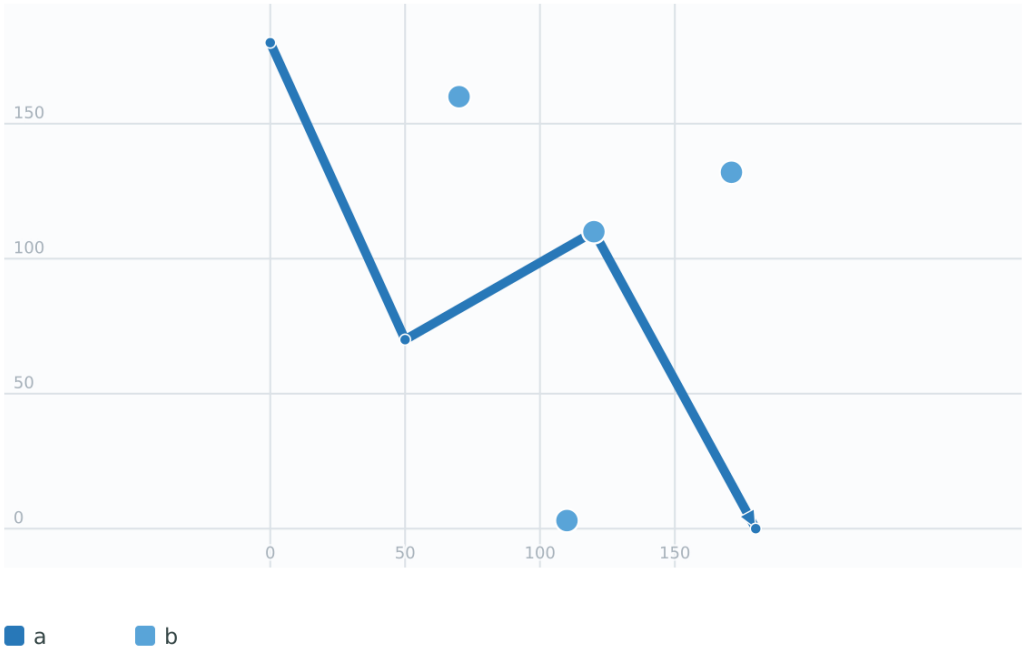
The following situations all return `true`.

A LineString crosses a MultiPoint when some, but not all, points lie on the line.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'MULTIPOINT (80 170,120 13,130 120,181 142)'::geometry AS b
) AS example;
```

Figure

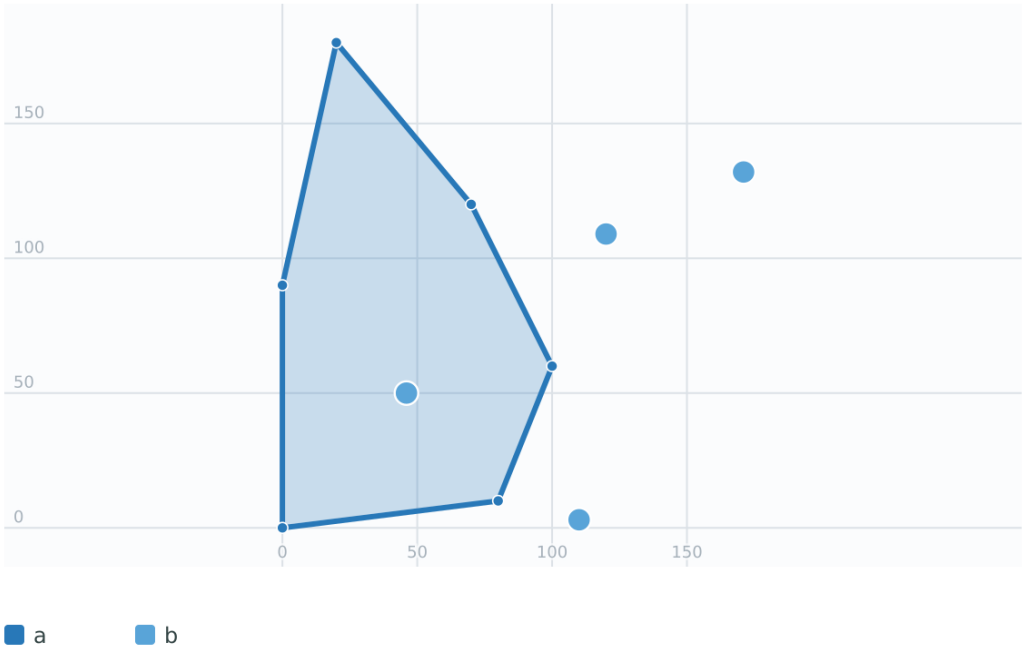


A Polygon crosses a MultiPoint when some, but not all, points lie in its interior.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'POLYGON ((30 190,10 100,10 10,90 20,110 70,80 130,30 190))'::geometry AS a,
  'MULTIPOINT (56 60,120 13,130 119,181 142)'::geometry AS b
) AS example;
```

Figure

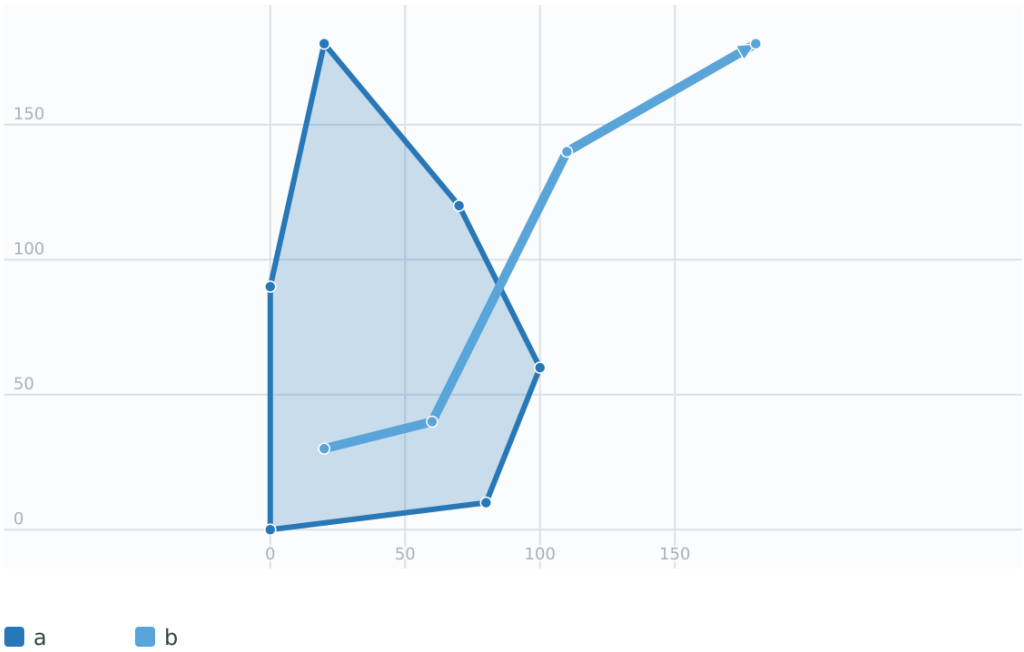


A LineString crosses a Polygon when part of the line lies inside and part outside.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'POLYGON ((30 190,10 100,10 10,90 20,110 70,80 130,30 190))'::geometry AS a,
  'LINESTRING (30 40,70 50,120 150,190 190)'::geometry AS b
) AS example;
```

Figure

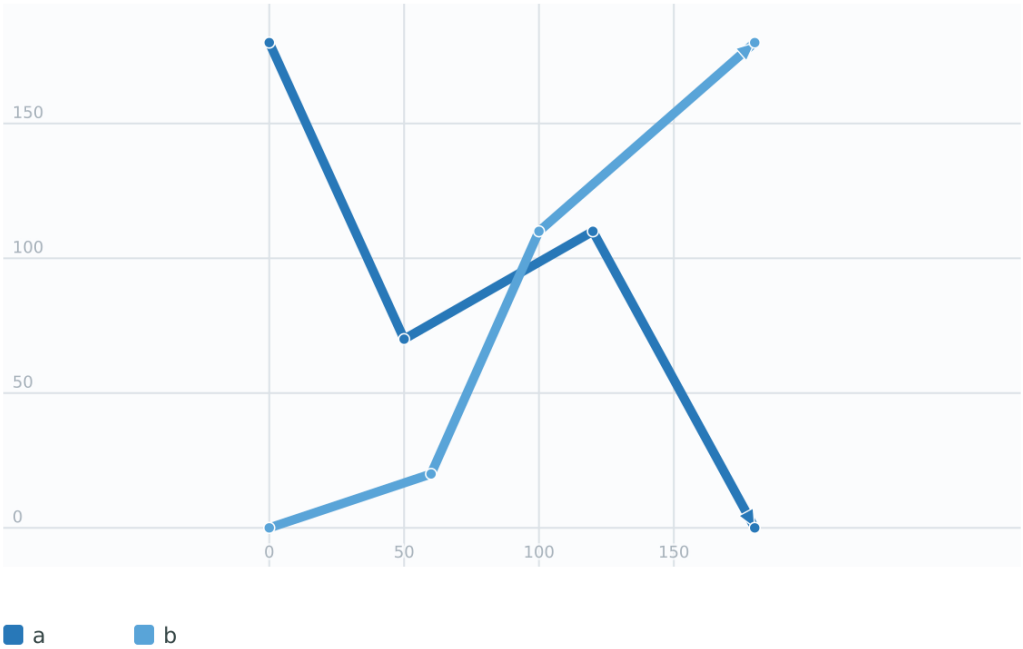


Two LineStrings cross when they share an interior point.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'LINESTRING (10 10,70 30,110 120,190 190)'::geometry AS b
) AS example;
```

Figure



Consider a situation where a user has two tables: a table of roads and a table of highways.

Code

```
CREATE TABLE roads (
  id serial NOT NULL,
  geom geometry,
  CONSTRAINT roads_pkey PRIMARY KEY ( ↩
    road_id)
);
```

Code

```
CREATE TABLE highways (
  id serial NOT NULL,
  the_gem geometry,
  CONSTRAINT roads_pkey PRIMARY KEY ( ↩
    road_id)
);
```

To determine a list of roads that cross a highway, use a query similar to:

Code

```
SELECT roads.id
FROM roads, highways
WHERE ST_Crosses(roads.geom, highways.geom);
```

See Also

ST_Contains, ST_Overlaps

7.11.1.7 ST_Disjoint

ST_Disjoint — Tests if two geometries have no points in common

Synopsis

boolean **ST_Disjoint**(geometry A , geometry B);

Description

Returns `true` if two geometries are disjoint. Geometries are disjoint if they have no point in common.

If any other spatial relationship is true for a pair of geometries, they are not disjoint. Disjoint implies that ST_Intersects is false.

In mathematical terms: $ST\_Disjoint(A, B) \Leftrightarrow A \bigcap B = \varnothing$

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

Performed by the GEOS module



Note
This function call does not use indexes. A negated ST_Intersects predicate can be used as a more performant alternative that uses indexes: `ST_Disjoint(A,B) = NOT ST_Intersects(A,B)`



Note
NOTE: this is the "allowable" version that returns a boolean, not an integer.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 //s2.1.13.3 - a.Relate(b, 'FF*FF****')



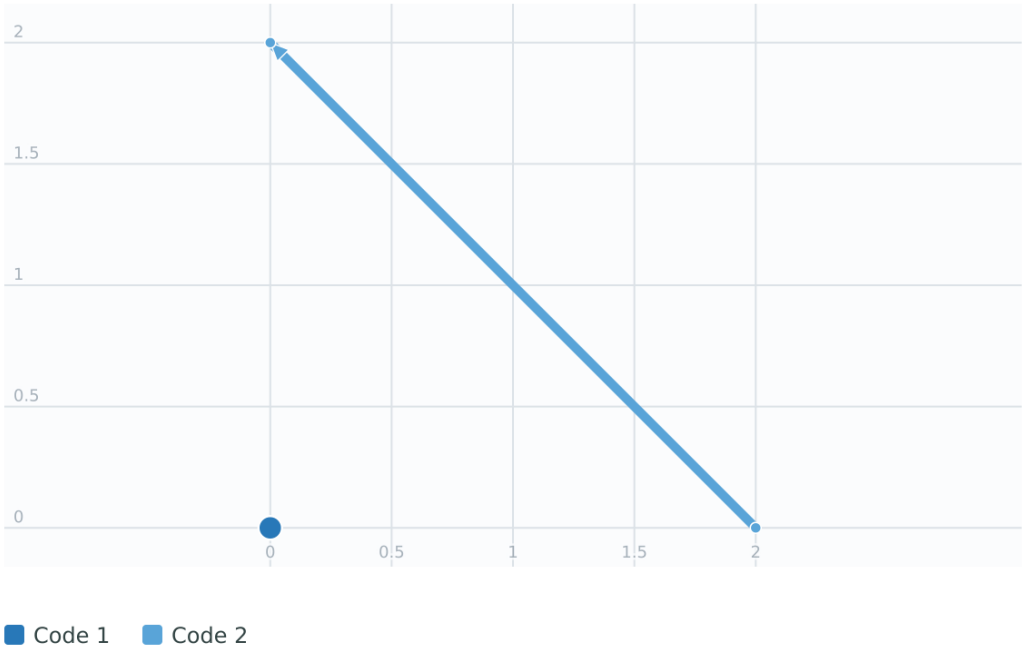
This method implements the SQL/MM specification. SQL-MM 3: 5.1.26

Examples

Code

```
SELECT ST_Disjoint('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

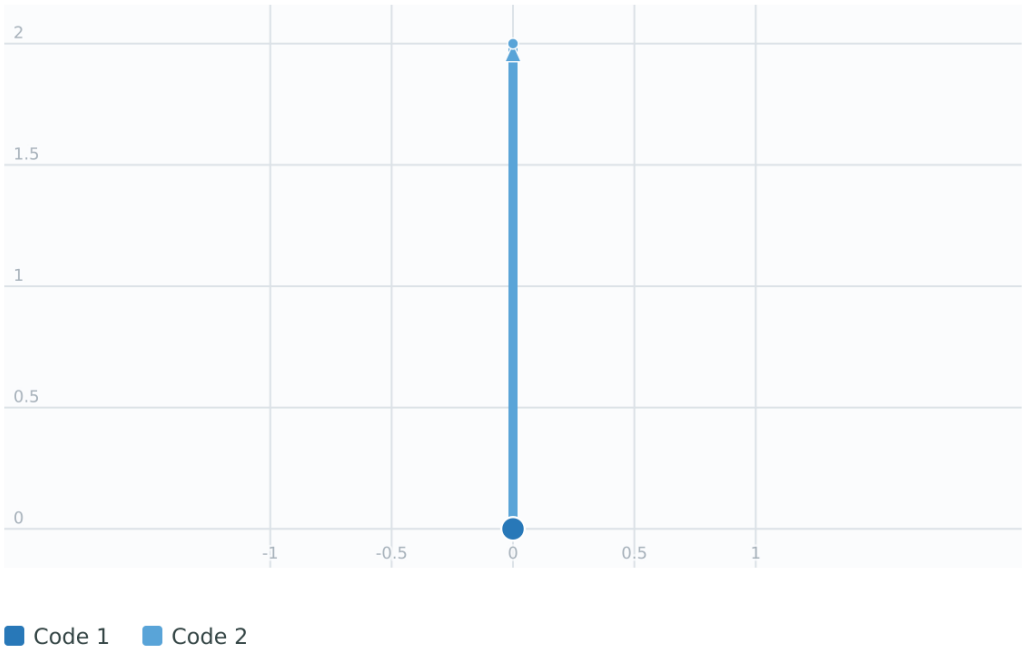
Figure



Code

```
SELECT ST_Disjoint('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Figure



See Also

ST_Intersects

7.11.1.8 ST_Equals

ST_Equals — Tests if two geometries include the same set of points

Synopsis

boolean **ST_Equals**(geometry A, geometry B);

Description

Returns `true` if the given geometries are "topologically equal". Use this for a 'better' answer than '='. Topological equality means that the geometries have the same dimension, and their point-sets occupy the same space. This means that the order of vertices may be different in topologically equal geometries. To verify the order of points is consistent use ST_OrderingEquals (it must be noted ST_OrderingEquals is a little more stringent than simply verifying order of points are the same).

In mathematical terms: *ST_Equals(A, B)* $\Leftrightarrow$ *A = B*

The following relation holds: *ST_Equals(A, B)* $\Leftrightarrow$ *ST_Within(A,B)* $\wedge$ *ST_Within(B,A)*

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2

This method implements the SQL/MM specification. SQL-MM 3: 5.1.24

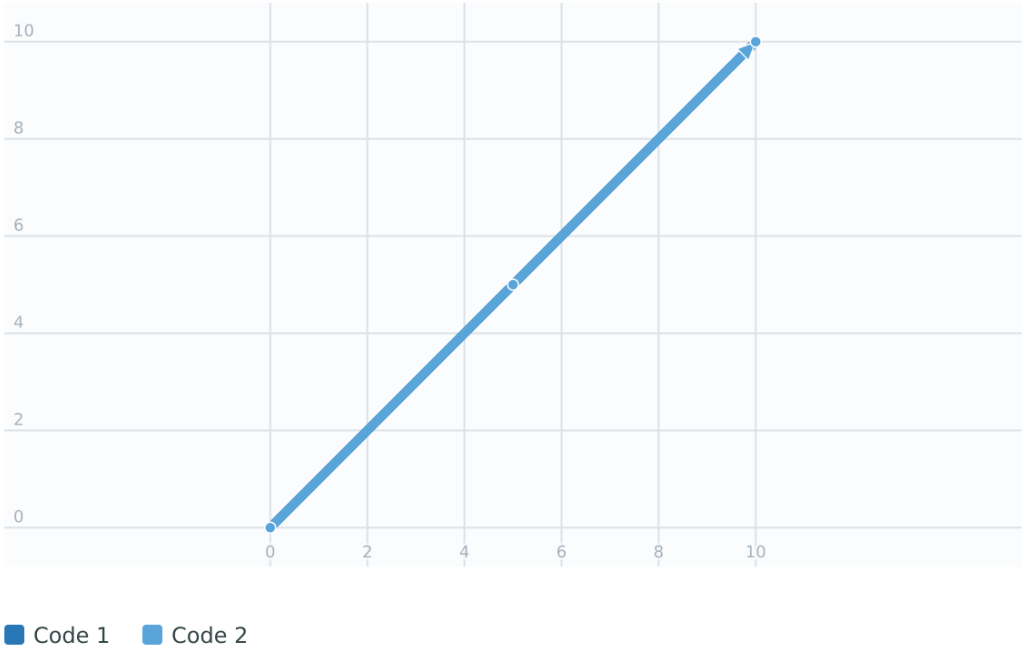
Changed: 2.2.0 Returns true even for invalid geometries if they are binary equal

Examples

Code

```
SELECT ST_Equals('LINESTRING(0 0,10 10)',
'LINESTRING(0 0,5 5,10 10)');
```

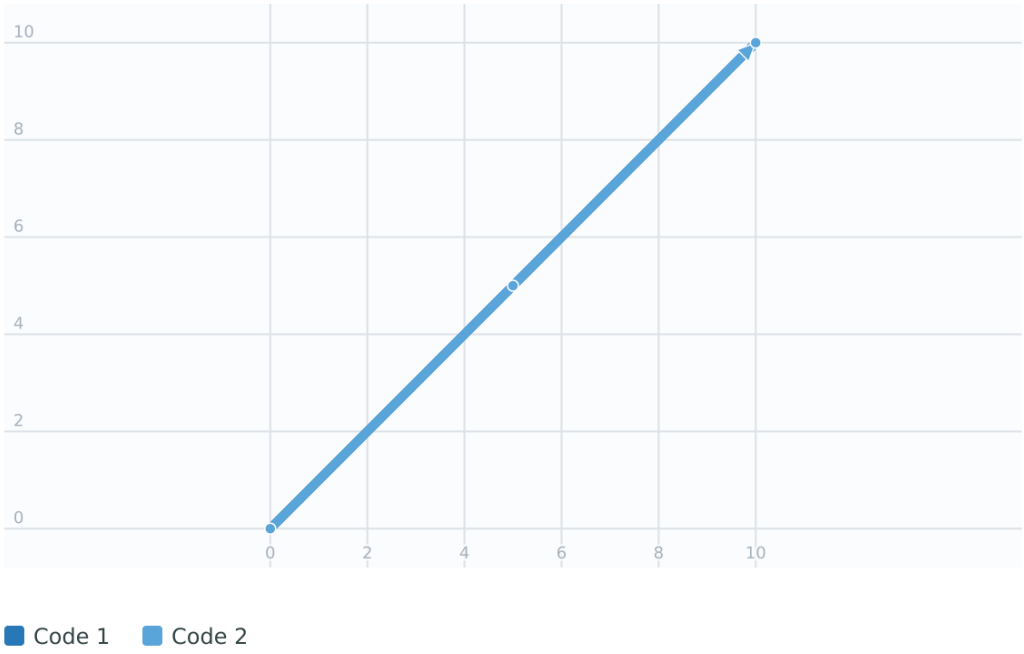
Figure



Code

```
SELECT ST_Equals(ST_Reverse('LINESTRING(0 0,10 10)'),
'LINESTRING(0 0,5 5,10 10)');
```

Figure



See Also

ST_IsValid, ST_OrderingEquals, ST_Reverse, ST_Within

7.11.1.9 ST_Intersects

ST_Intersects — Tests if two geometries intersect (they have at least one point in common)

Synopsis

boolean **ST_Intersects**(geometry geomA , geometry geomB);
boolean **ST_Intersects**(geography geogA , geography geogB);

Description

Returns `true` if two geometries intersect. Geometries intersect if they have any point in common.

For geography, a distance tolerance of 0.00001 meters is used (so points that are very close are considered to intersect).

In mathematical terms: *ST_Intersects(A, B)* $\Leftrightarrow A \bigcap B \neq \varnothing$

Geometries intersect if their DE-9IM Intersection Matrix matches one of:

- `T********`
- `*T*******`
- `***T*****`

- ****T****

Spatial intersection is implied by all the other spatial relationship tests, except ST_Disjoint, which tests that geometries do NOT intersect.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

Changed: 3.0.0 SFCGAL version removed and native support for 2D TINS added.

Enhanced: 2.5.0 Supports GEOMETRYCOLLECTION.

Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Performed by the GEOS module (for geometry), geography is native

Availability: 1.5 support for geography was introduced.



Note
For geography, this function has a distance tolerance of about 0.00001 meters and uses the sphere rather than spheroid calculation.



Note
NOTE: this is the "allowable" version that returns a boolean, not an integer.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 //s2.1.13.3 - ST_Intersects(g1, g2) --> Not (ST_Disjoint(g1, g2))

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.27

 This method supports Circular Strings and Curves.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Geometry Examples

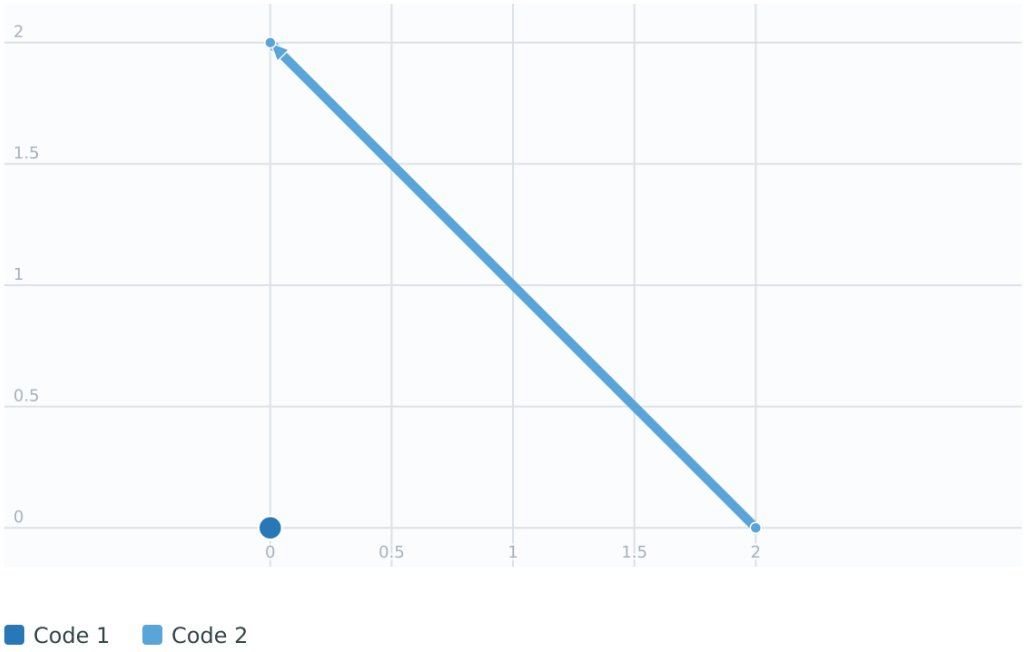
Code

```
SELECT ST_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
f
```

Figure



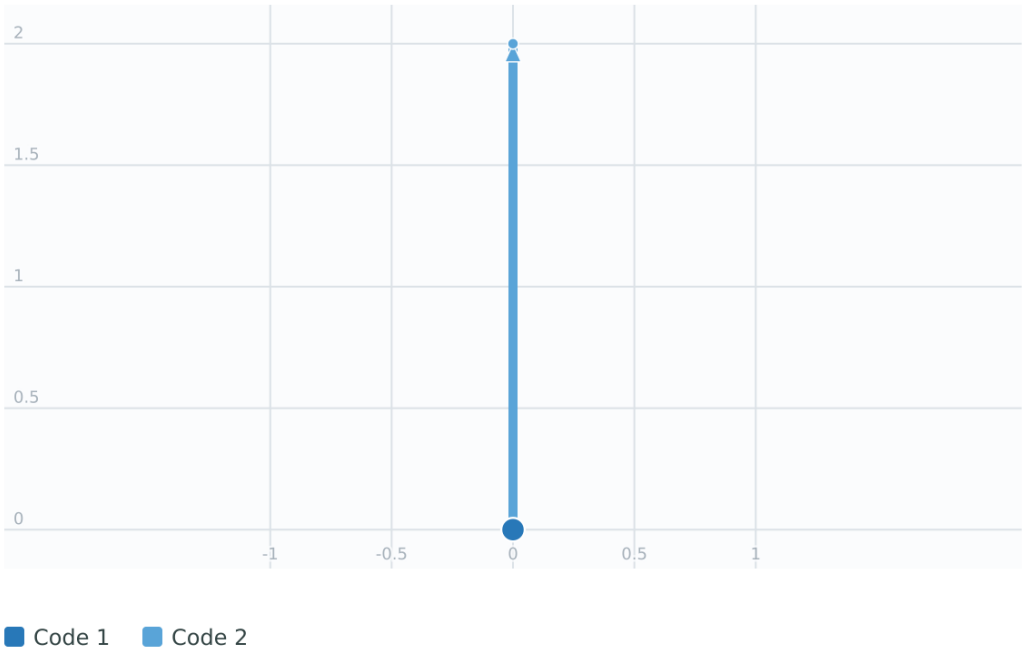
Code

```
SELECT ST_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Output

```
t
```

Figure



This example looks up rows in a table; for faster lookup, the geometry column should have a GiST index.

Code

```
SELECT id, name FROM cities WHERE ST_Intersects(geom, 'SRID=4326;POLYGON((28 53,27.707 52.293,27 52,26.293 52.293,26 53,26.293 53.707,27 54,27.707 53.707,28 53))');
```

Output

```
 id | name
----+-------
  2 | Minsk
(1 row)
```

Geography Examples

Code

```
SELECT ST_Intersects('SRID=4326;LINESTRING(-43.23456 72.4567,-43.23456 72.4568)'::geography,
'SRID=4326;POINT(-43.23456 72.4567772)'::geography
);
```

Output

```
t
```

See Also

&&, ST_3DIntersects, ST_Disjoint

7.11.1.10 ST_LineCrossingDirection

ST_LineCrossingDirection — Returns a number indicating the crossing behavior of two LineStrings

Synopsis

integer **ST_LineCrossingDirection**(geometry linestringA, geometry linestringB);

Description

Given two linestrings returns an integer between -3 and 3 indicating what kind of crossing behavior exists between them. 0 indicates no crossing. This is only supported for `LINESTRING`s.

The crossing number has the following meaning:

- 0: LINE NO CROSS
- -1: LINE CROSS LEFT
- 1: LINE CROSS RIGHT
- -2: LINE MULTICROSS END LEFT
- 2: LINE MULTICROSS END RIGHT
- -3: LINE MULTICROSS END SAME FIRST LEFT
- 3: LINE MULTICROSS END SAME FIRST RIGHT

Availability: 1.4

Examples

Example: LINE CROSS LEFT and LINE CROSS RIGHT

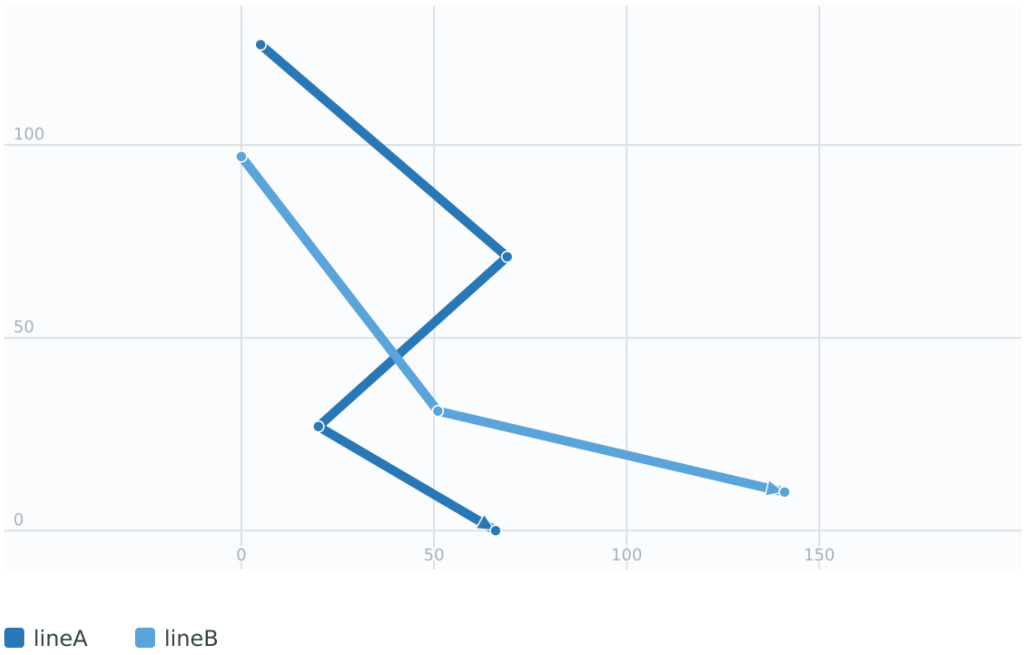
Code

```
SELECT ST_LineCrossingDirection(lineA, lineB) As A_cross_B,
       ST_LineCrossingDirection(lineB, lineA) As B_cross_A
FROM (SELECT
  'LINESTRING(25 169,89 114,40 70,86 43)'::geometry As lineA,
  'LINESTRING (20 140,71 74,161 53)'::geometry As lineB
  ) As foo;
```

Output

```
 A_cross_B | B_cross_A
-----------+-----------
        -1 |         1
```

Figure



Example: LINE MULTICROSS END SAME FIRST LEFT and LINE MULTICROSS END SAME FIRST RIGHT

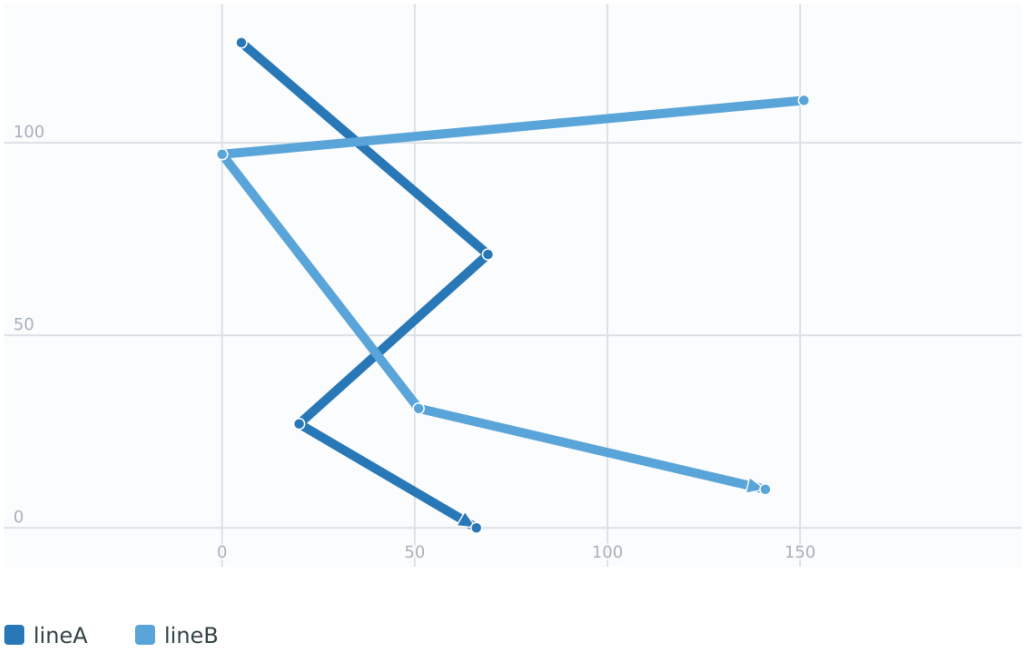
Code

```
SELECT ST_LineCrossingDirection(lineA, lineB) As A_cross_B,
       ST_LineCrossingDirection(lineB, lineA) As B_cross_A
FROM (SELECT
 'LINESTRING(25 169,89 114,40 70,86 43)'::geometry As lineA,
 'LINESTRING(171 154,20 140,71 74,161 53)'::geometry As lineB
  ) As foo;
```

Output

```
 A_cross_B | B_cross_A
-----------+-----------
         3 |        -3
```

Figure



Example: LINE MULTICROSS END LEFT and LINE MULTICROSS END RIGHT

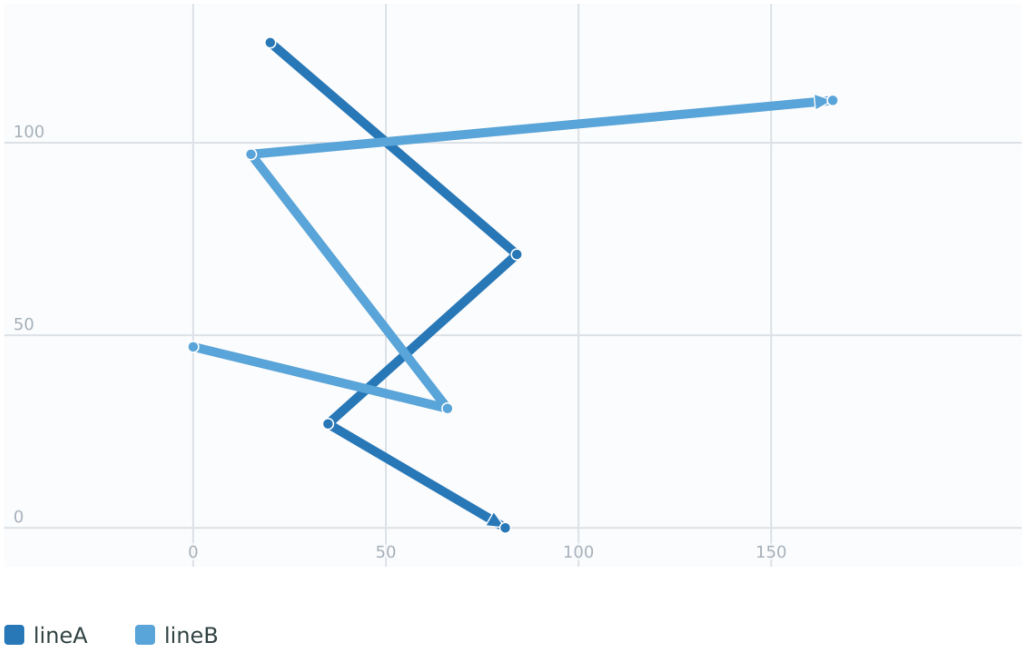
Code

```
SELECT ST_LineCrossingDirection(lineA, lineB) As A_cross_B,
       ST_LineCrossingDirection(lineB, lineA) As B_cross_A
FROM (SELECT
  'LINESTRING(25 169,89 114,40 70,86 43)'::geometry As lineA,
  'LINESTRING(5 90,71 74,20 140,171 154)'::geometry As lineB
  ) As foo;
```

Output

```
 A_cross_B | B_cross_A
-----------+-----------
        -2 |         2
```

Figure



Example: Finds all streets that cross

Code

```
SELECT s1.gid, s2.gid, ST_LineCrossingDirection(s1.geom, s2.geom)
  FROM streets s1 CROSS JOIN streets s2
        ON (s1.gid != s2.gid AND s1.geom && s2.geom )
WHERE ST_LineCrossingDirection(s1.geom, s2.geom) > 0;
```

See Also

ST_Crosses

7.11.1.11 ST_OrderingEquals

ST_OrderingEquals — Tests if two geometries represent the same geometry and have points in the same directional order

Synopsis

boolean **ST_OrderingEquals**(geometry A, geometry B);

Description

ST_OrderingEquals compares two geometries and returns t (TRUE) if the geometries are equal and the coordinates are in the same order; otherwise it returns f (FALSE).



Note
This function is implemented as per the ArcSDE SQL specification rather than SQL-MM. http://edndoc.esri.com/arcsde/9.1/sql_api/sqlapi3.htm#ST_OrderingEquals

This method implements the SQL/MM specification. SQL-MM 3: 5.1.43

Examples

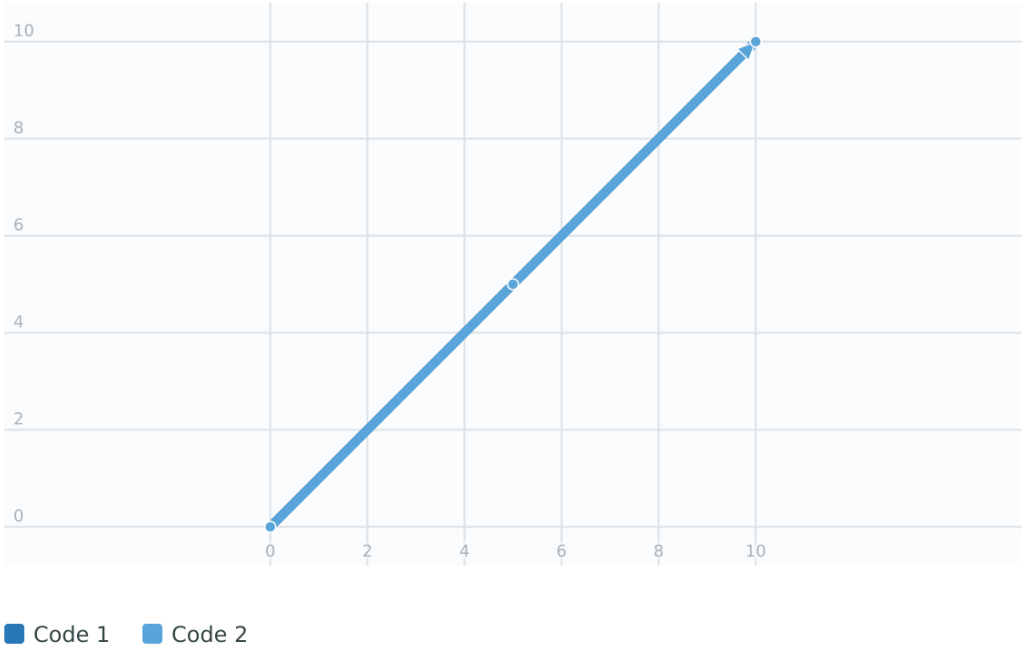
Code

```
SELECT ST_OrderingEquals('LINESTRING(0 0,10 10)',
'LINESTRING(0 0,5 5,10 10)');
```

Output

```
f
```

Figure



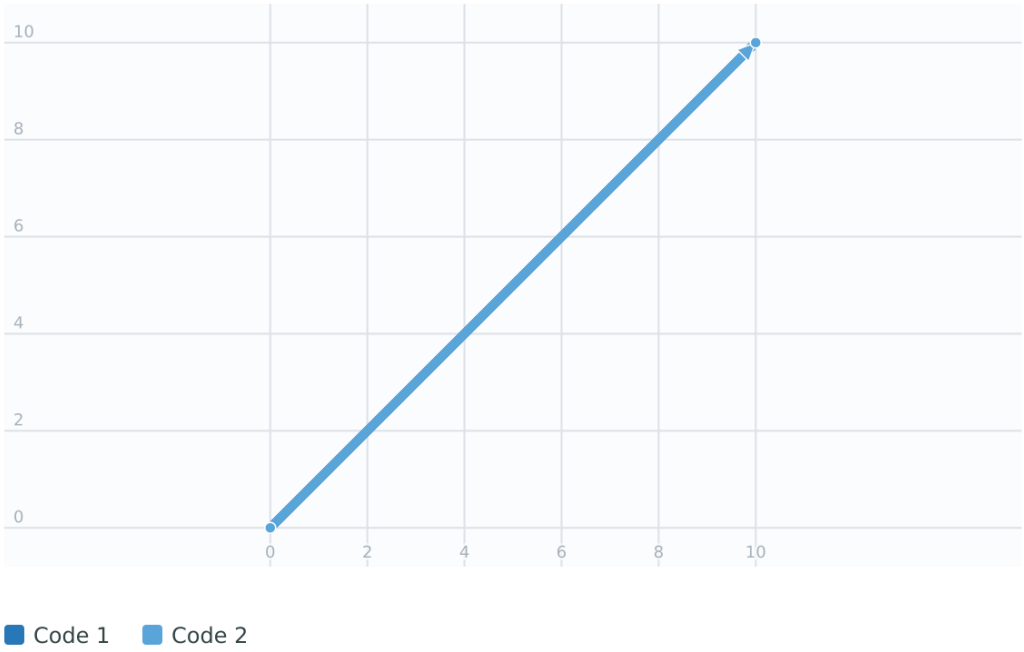
Code

```
SELECT ST_OrderingEquals('LINESTRING(0 0,10 10)',
'LINESTRING(0 0,10 10)');
```

Output

```
t
```

Figure



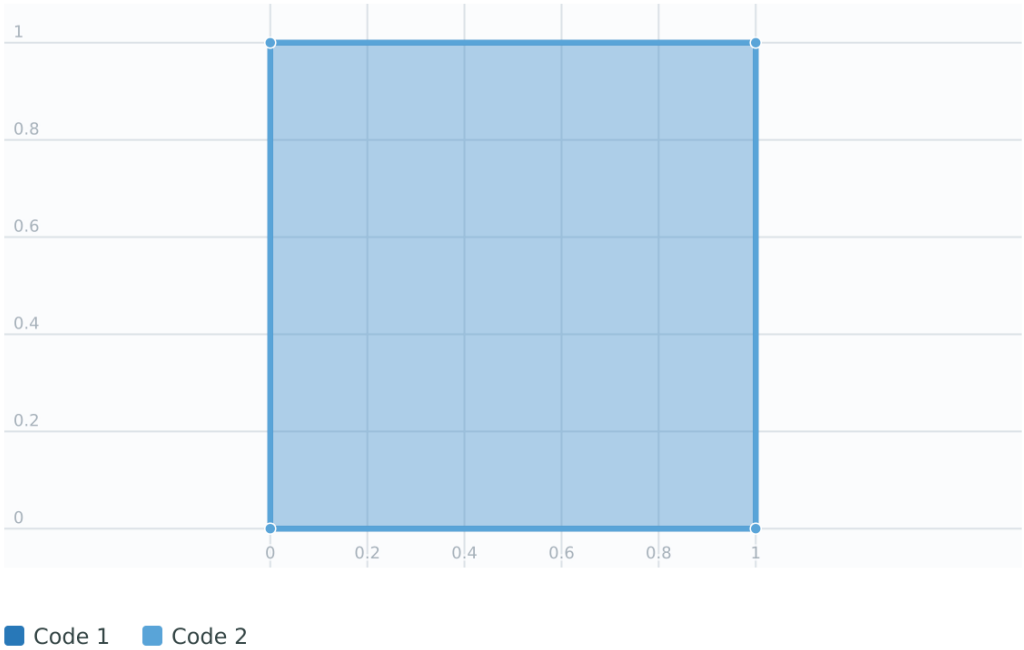
Code

```
SELECT ST_OrderingEquals('POLYGON((0 0,0 1,1 1,1 0,0 0))',
'POLYGON((0 0,1 0,1 1,0 1,0 0))');
```

Output

```
f
```

Figure



See Also

&&, ST_Equals, ST_Reverse

7.11.1.12 ST_Overlaps

ST_Overlaps — Tests if two geometries have the same dimension and intersect, but each has at least one point not in the other

Synopsis

boolean **ST_Overlaps**(geometry A, geometry B);

Description

Returns TRUE if geometry A and B "spatially overlap". Two geometries overlap if they have the same dimension, their interiors intersect in that dimension. and each has at least one point inside the other (or equivalently, neither one covers the other). The overlaps relation is symmetric and irreflexive.

In mathematical terms: $ST\_Overlaps(A, B) \Leftrightarrow (dim(A) = dim(B) = dim(Int(A) \bigcap Int(B))) \wedge (A \bigcap B \neq A) \wedge (A \bigcap B \neq B)$



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Overlaps`.

Performed by the GEOS module

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

NOTE: this is the "allowable" version that returns a boolean, not an integer.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3


This method implements the SQL/MM specification. SQL-MM 3: 5.1.32

Examples

`ST_Overlaps` returns `TRUE` in the following situations:

Two MultiPoints overlap when they share some, but not all, points.

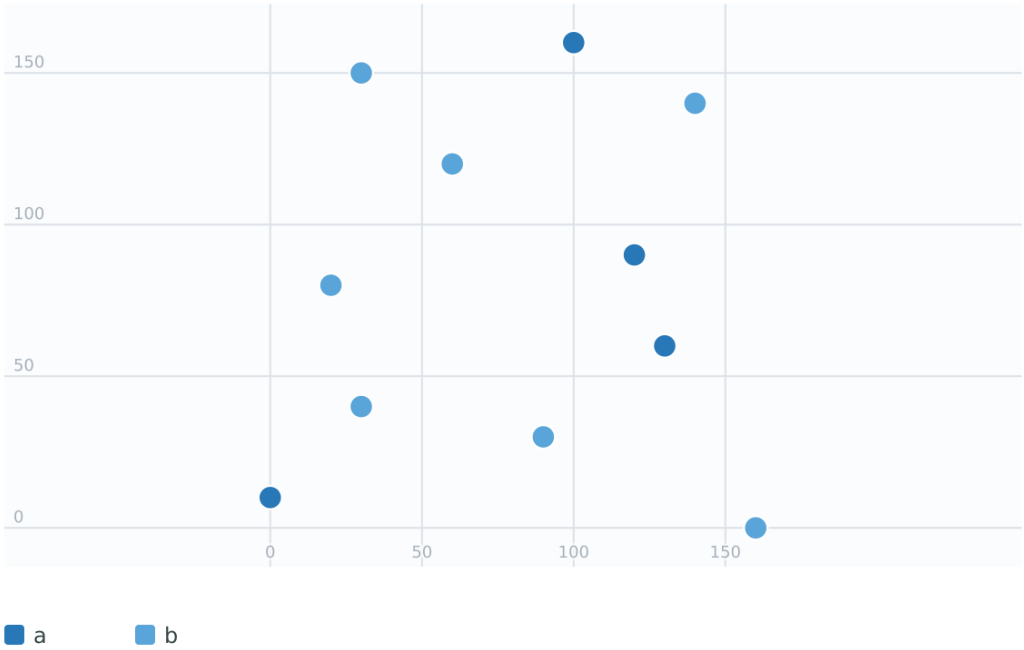
Code

```
SELECT ST_Overlaps(a, b)
FROM (SELECT
  'MULTIPOINT ((20 20),(40 90),(80 130),(140 100),(110 40),(120 170),(150 70))'::geometry AS ↩
    a,
  'MULTIPOINT ((180 10),(40 90),(80 130),(160 150),(110 40),(50 50),(50 160))'::geometry AS  ↩
    b
) AS example;
```

Output

```
t
```

Figure



Two LineStrings overlap along part of their paths.

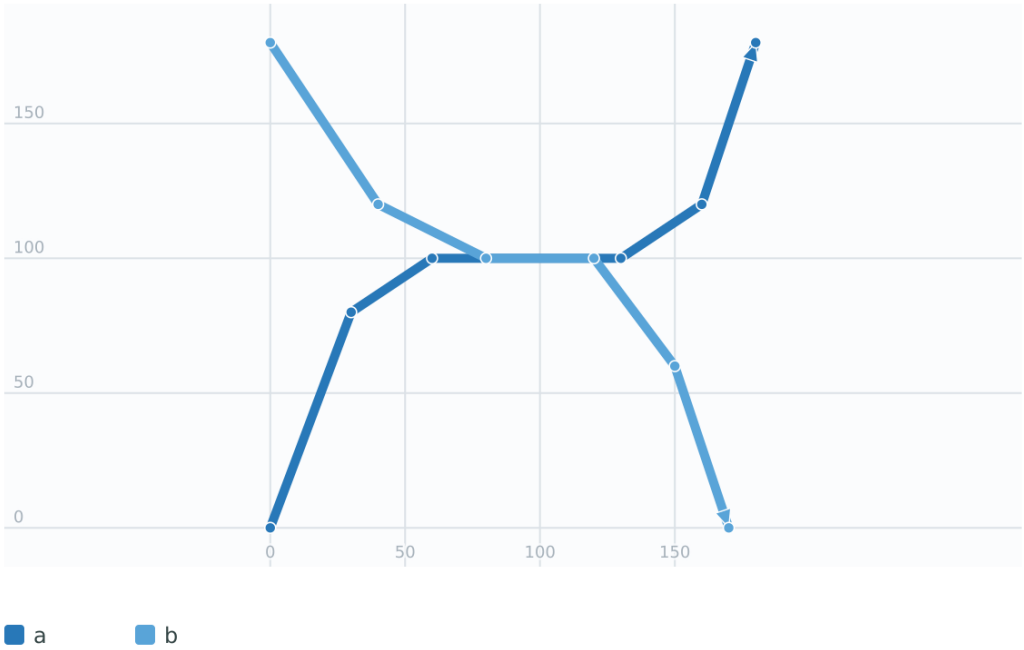
Code

```
SELECT ST_Overlaps(a, b)
FROM (SELECT
  'LINESTRING (10 10,40 90,70 110,140 110,170 130,190 190)'::geometry AS a,
  'LINESTRING (10 190,50 130,90 110,130 110,160 70,180 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two Polygons overlap where their interiors intersect and neither contains the other.

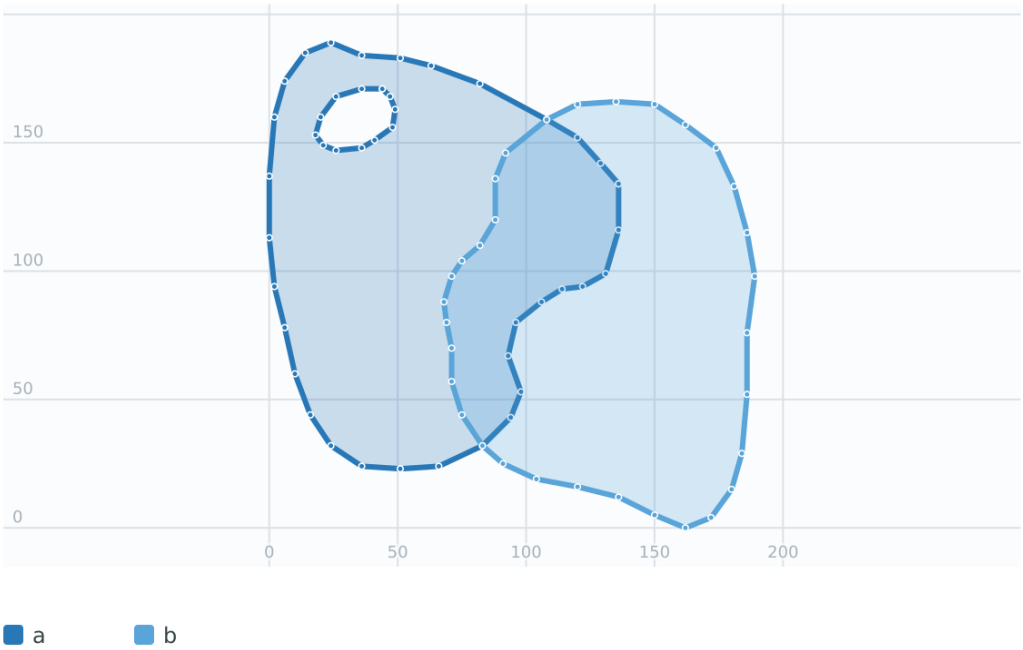
Code

```
SELECT ST_Overlaps(a, b)
FROM (SELECT
  'POLYGON ((140 140,140 122,135 105,126 100,118 99,110 94,100 86,97 73,102 59,98 49,87 ↩
    38,70 30,55 29,40 30,28 38,20 50,14 66,10 84,6 100,4 119,4 143,6 166,10 180,18 191,28 ↩
    195,40 190,55 189,67 186,86 179,112 165,124 158,133 148,140 140),(48 177,40 177,30 ↩
    174,24 166,22 159,25 155,30 153,40 154,45 157,52 162,53 169,51 174,48 177))'::geometry ↩
    AS a,
  'POLYGON ((75 63,79 50,87 38,95 31,108 25,124 22,140 18,154 11,166 6,176 10,184 21,188 ↩
    35,190 58,190 82,193 104,190 121,185 139,178 154,166 163,154 171,139 172,124 171,112 ↩
    165,96 152,92 142,92 126,86 116,79 110,75 104,72 94,73 86,75 76,75 63))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Point on a LineString is contained, but since it has lower dimension it does not overlap or cross.

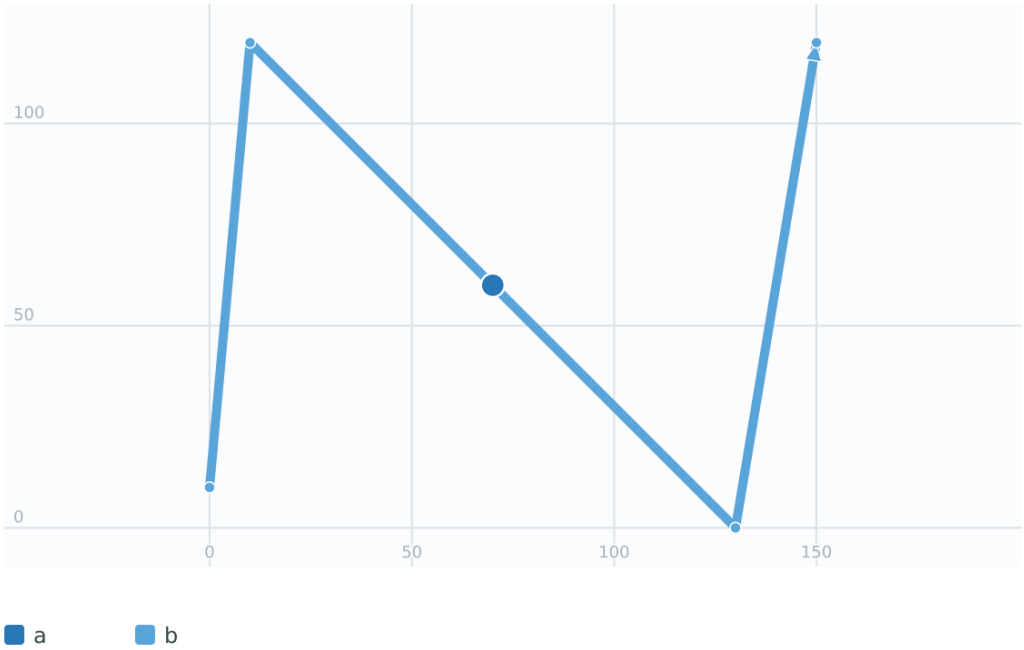
Code

```
SELECT ST_Overlaps(a, b) AS overlaps, ST_Crosses(a, b) AS crosses,
       ST_Intersects(a, b) AS intersects, ST_Contains(b, a) AS b_contains_a
FROM (SELECT 'POINT (100 100)'::geometry As a,
             'LINESTRING (30 50,40 160,160 40,180 160)'::geometry  AS b) AS t
```

Output

```
overlaps | crosses | intersects | b_contains_a
---------+---------+------------+--------------
f        | f       | t          | t
```

Figure



A LineString that partly covers a Polygon intersects and crosses, but does not overlap since it has different dimension.

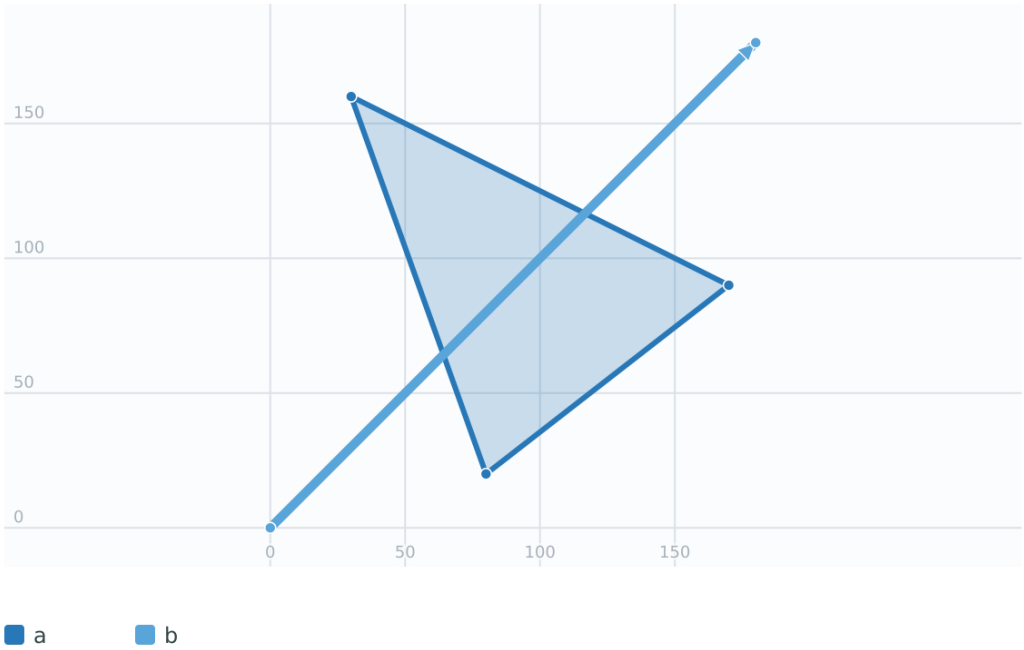
Code

```
SELECT ST_Overlaps(a, b) AS overlaps, ST_Crosses(a, b) AS crosses,
       ST_Intersects(a, b) AS intersects, ST_Contains(a, b) AS contains
FROM (SELECT 'POLYGON ((40 170,90 30,180 100,40 170))'::geometry AS a,
             'LINESTRING(10 10,190 190)'::geometry AS b) AS t;
```

Output

```
 overlap | crosses | intersects | contains
---------+---------+------------+--------------
 f       | t       | t          | f
```

Figure



Two Polygons that intersect but with neither contained by the other overlap, but do not cross because their intersection has the same dimension.

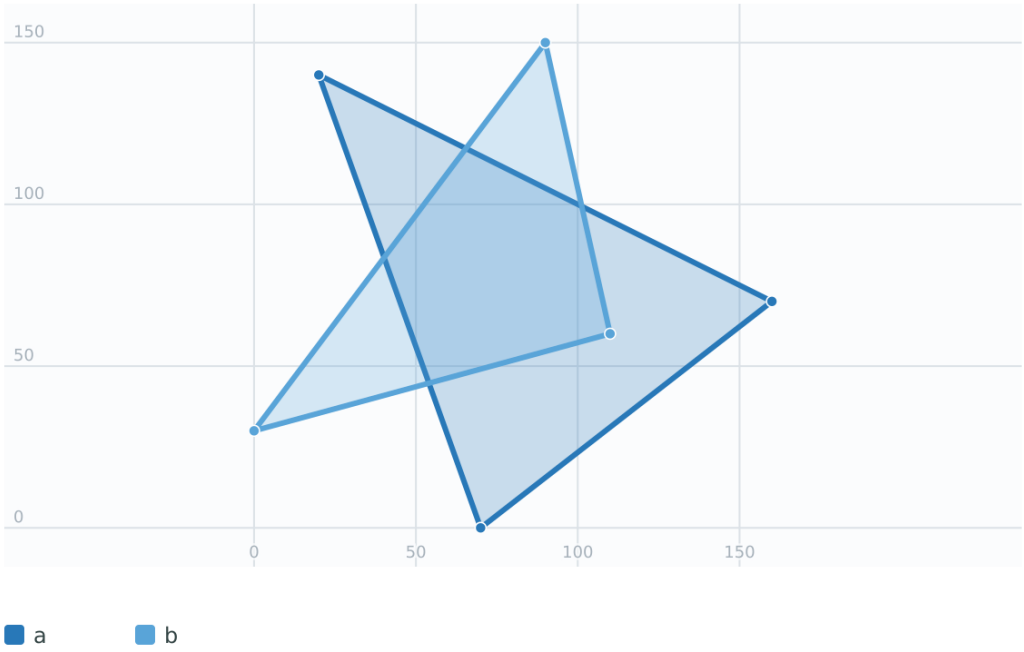
Code

```
SELECT ST_Overlaps(a, b) AS overlaps, ST_Crosses(a, b) AS crosses,
       ST_Intersects(a, b) AS intersects, ST_Contains(b, a) AS b_contains_a,
       ST_Dimension(a) AS dim_a, ST_Dimension(b) AS dim_b,
       ST_Dimension(ST_Intersection(a, b)) AS dim_int
FROM (SELECT 'POLYGON ((40 170,90 30,180 100,40 170))'::geometry AS a,
             'POLYGON ((110 180,20 60,130 90,110 180))'::geometry AS b) As t;
```

Output

```
 overlaps | crosses | intersects | b_contains_a | dim_a | dim_b | dim_int
----------+---------+------------+--------------+-------+-------+----------
 t        | f       | t          | f            |     2 |     2 |       2
```

Figure



See Also

ST_Contains, ST_Crosses, ST_Dimension, ST_Intersects

7.11.1.13 ST_Relate

ST_Relate — Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix

Synopsis

boolean **ST_Relate**(geometry geomA, geometry geomB, text intersectionMatrixPattern);
text **ST_Relate**(geometry geomA, geometry geomB);
text **ST_Relate**(geometry geomA, geometry geomB, integer boundaryNodeRule);

Description

These functions allow testing and evaluating the spatial (topological) relationship between two geometries, as defined by the Dimensionally Extended 9-Intersection Model (DE-9IM).

The DE-9IM is specified as a 9-element matrix indicating the dimension of the intersections between the Interior, Boundary and Exterior of two geometries. It is represented by a 9-character text string using the symbols 'F', '0', '1', '2' (e.g. `'FF1FF0102'`).

A specific kind of spatial relationship can be tested by matching the intersection matrix to an *intersection matrix pattern*. Patterns can include the additional symbols 'T' (meaning "intersection is non-empty") and '*' (meaning "any value"). Common spatial relationships are provided by the named functions ST_Contains, ST_ContainsProperly, ST_Covers, ST_CoveredBy, ST_Crosses, ST_Disjoint, ST_Equals, ST_Intersects, ST_Overlaps, ST_Touches, and ST_Within. Using an explicit pattern allows testing multiple conditions of intersects, crosses, etc in one step. It also allows testing spatial relationships which do not have a named spatial relationship function. For example, the relationship "Interior-Intersects" has the DE-9IM pattern `T********`, which is not evaluated by any named predicate.

For more information refer to Section 5.1.

Variant 1: Tests if two geometries are spatially related according to the given `intersectionMatrixPattern`.



Note
Unlike most of the named spatial relationship predicates, this does NOT automatically include an index call. The reason is that some relationships are true for geometries which do NOT intersect (e.g. Disjoint). If you are using a relationship pattern that requires intersection, then include the && index call.



Note
It is better to use a named relationship function if available, since they automatically use a spatial index where one exists. Also, they may implement performance optimizations which are not available with full relate evaluation.

Variant 2: Returns the DE-9IM matrix string for the spatial relationship between the two input geometries. The matrix string can be tested for matching a DE-9IM pattern using ST_RelateMatch.

Variant 3: Like variant 2, but allows specifying a **Boundary Node Rule**. A boundary node rule allows finer control over whether the endpoints of MultiLineStrings are considered to lie in the DE-9IM Interior or Boundary. The `boundaryNodeRule` values are:

- `1`: **OGC-Mod2** - line endpoints are in the Boundary if they occur an odd number of times. This is the rule defined by the OGC SFS standard, and is the default for `ST_Relate`.
- `2`: **Endpoint** - all endpoints are in the Boundary.
- `3`: **MultivalentEndpoint** - endpoints are in the Boundary if they occur more than once. In other words, the boundary is all the "attached" or "inner" endpoints (but not the "unattached/outer" ones).
- `4`: **MonovalentEndpoint** - endpoints are in the Boundary if they occur only once. In other words, the boundary is all the "unattached" or "outer" endpoints.

This function is not in the OGC spec, but is implied. see s2.1.13.2



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3



This method implements the SQL/MM specification. SQL-MM 3: 5.1.25

Performed by the GEOS module

Enhanced: 2.0.0 - added support for specifying boundary node rule.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

Examples

Using the boolean-valued function to test spatial relationships.

Code

```
SELECT ST_Relate('POINT(1 2)', ST_Buffer('POINT(1 2)', 2), '0FFFFF212');
```

Output

```
t
```

Code

```
SELECT ST_Relate('POINT(1 2)', ST_Buffer('POINT(1 2)', 2), '*FF*FF212');
```

Output

```
t
```

Testing a custom spatial relationship pattern as a query condition, with `&&` to enable using a spatial index.

Find compounds that properly intersect (not just touch) a poly (Interior Intersects)

Code

```
SELECT c.* , p.name As poly_name
    FROM polys AS p
    INNER JOIN compounds As c
          ON c.geom && p.geom
             AND ST_Relate(p.geom, c.geom, 'T********');
```

Computing the intersection matrix for spatial relationships.

Code

```
SELECT ST_Relate('POINT(1 2)',
ST_Buffer('POINT(1 2)', 2));
```

Output

```
0FFFFF212
```

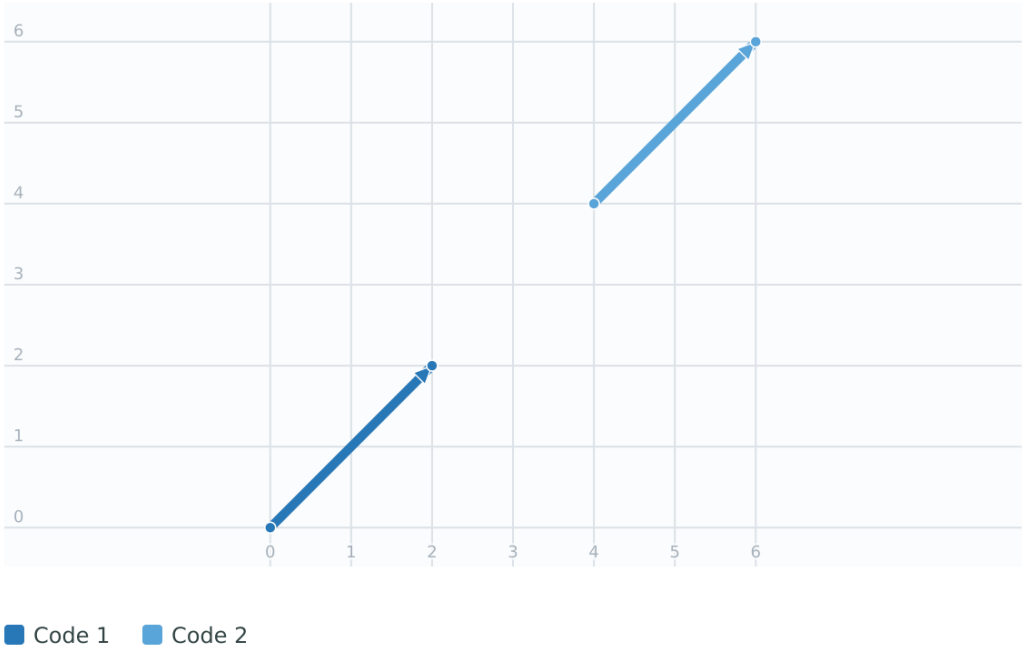
Code

```
SELECT ST_Relate('LINESTRING(1 2,3 4)',
'LINESTRING(5 6,7 8)' );
```

Output

```
FF1FF0102
```

Figure



Using different Boundary Node Rules to compute the spatial relationship between a LineString and a MultiLineString with a duplicate endpoint `(3 3)`:

- Using the **OGC-Mod2** rule (1) the duplicate endpoint is in the **interior** of the MultiLineString, so the DE-9IM matrix entry [aB:bI] is `0` and [aB:bB] is `F`.
- Using the **Endpoint** rule (2) the duplicate endpoint is in the **boundary** of the MultiLineString, so the DE-9IM matrix entry [aB:bI] is `F` and [aB:bB] is `0`.

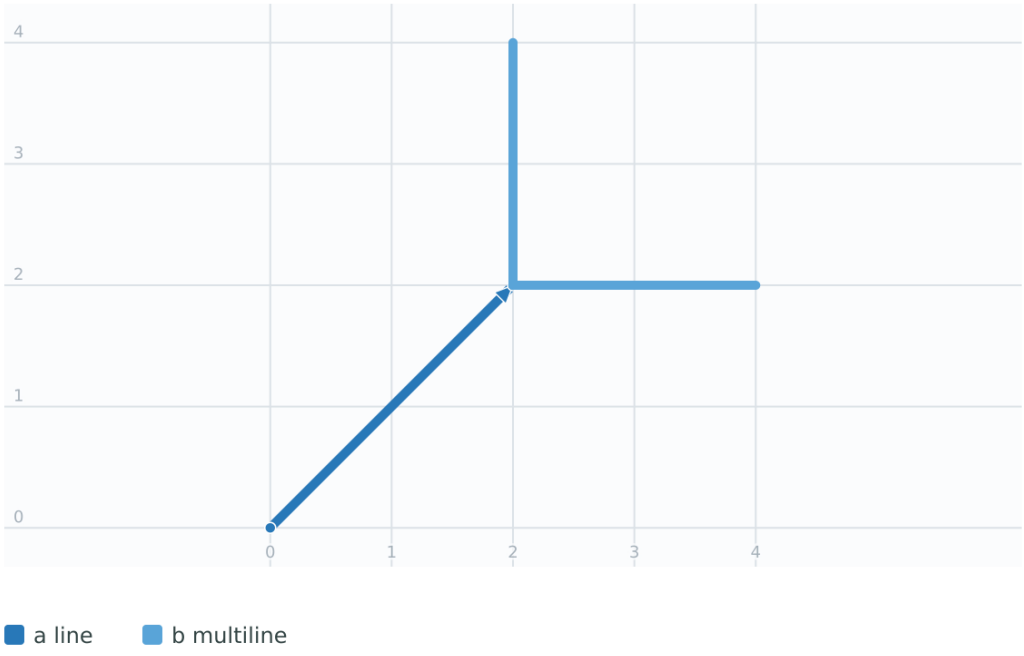
Code

```
WITH data AS (SELECT
  'LINESTRING(1 1,3 3)'::geometry AS a_line,
  'MULTILINESTRING((3 3,3 5),(3 3,5 3))'::geometry AS b_multiline
)
SELECT ST_Relate(a_line, b_multiline, 1) AS bnr_mod2,
       ST_Relate(a_line, b_multiline, 2) AS bnr_endpoint
    FROM data;
```

Output

```
 bnr_mod2  | bnr_endpoint
-----------+--------------
 FF10F0102 | FF1F00102
```

Figure



See Also

Section 5.1, ST_RelateMatch, ST_Contains, ST_ContainsProperly, ST_Covers, ST_CoveredBy, ST_Crosses, ST_Disjoint, ST_Equals, ST_Intersects, ST_Overlaps, ST_Touches, ST_Within

7.11.1.14 ST_RelateMatch

ST_RelateMatch — Tests if a DE-9IM Intersection Matrix matches an Intersection Matrix pattern

Synopsis

boolean **ST_RelateMatch**(text intersectionMatrix, text intersectionMatrixPattern);

Description

Tests if a Dimensionally Extended 9-Intersection Model (DE-9IM) `intersectionMatrix` value satisfies an `intersectionMatr`
Intersection matrix values can be computed by ST_Relate.

For more information refer to Section 5.1.

Performed by the GEOS module

Availability: 2.0.0

Examples

Code

```
SELECT ST_RelateMatch('101202FFF', 'TTTTTTFFF') ;
```

Output

```
t
```

Patterns for common spatial relationships matched against intersection matrix values, for a line in various positions relative to a polygon

Code

```
SELECT pat.name AS relationship, pat.val AS pattern,
       mat.name AS position, mat.val AS matrix,
       ST_RelateMatch(mat.val, pat.val) AS match
    FROM (VALUES ( 'Equality', 'T1FF1FFF1' ),
                 ( 'Overlaps', 'T*T***T**' ),
                 ( 'Within', 'T*F**F***' ),
                 ( 'Disjoint', 'FF*FF****' )) AS pat(name, val)
    CROSS JOIN
        (VALUES  ('non-intersecting', 'FF1FF0212'),
                 ('overlapping', '1010F0212'),
                 ('inside', '1FF0FF212')) AS mat(name, val);
```

Output

```
 relationship |  pattern  |     position      |  matrix   | match
--------------+-----------+-------------------+-----------+-------
 Equality     | T1FF1FFF1 | non-intersecting | FF1FF0212 | f
 Equality     | T1FF1FFF1 | overlapping       | 1010F0212 | f
 Equality     | T1FF1FFF1 | inside            | 1FF0FF212 | f
 Overlaps     | T*T***T** | non-intersecting | FF1FF0212 | f
 Overlaps     | T*T***T** | overlapping       | 1010F0212 | t
 Overlaps     | T*T***T** | inside            | 1FF0FF212 | f
 Within       | T*F**F*** | non-intersecting | FF1FF0212 | f
 Within       | T*F**F*** | overlapping       | 1010F0212 | f
 Within       | T*F**F*** | inside            | 1FF0FF212 | t
 Disjoint     | FF*FF**** | non-intersecting | FF1FF0212 | t
 Disjoint     | FF*FF**** | overlapping       | 1010F0212 | f
 Disjoint     | FF*FF**** | inside            | 1FF0FF212 | f
```

See Also

Section 5.1, ST_Relate

7.11.1.15 ST_Touches

ST_Touches — Tests if two geometries have at least one point in common, but their interiors do not intersect

Synopsis

boolean **ST_Touches**(geometry A, geometry B);

Description

Returns TRUE if A and B intersect, but their interiors do not intersect. Equivalently, A and B have at least one point in common, and the common points lie in at least one boundary. For Point/Point inputs the relationship is always FALSE, since points do not have a boundary.

In mathematical terms: $ST\_Touches(A, B) \Leftrightarrow (Int(A) \bigcap Int(B) = \varnothing) \wedge (A \bigcap B \neq \varnothing)$

This relationship holds if the DE-9IM Intersection Matrix for the two geometries matches one of:

- FT*******
- F**T*****
- F***T****



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid using an index, use _ST_Touches instead.

Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3


This method implements the SQL/MM specification. SQL-MM 3: 5.1.28

Examples

The ST_Touches predicate returns TRUE in the following examples.

Two Polygons touching along a boundary segment.

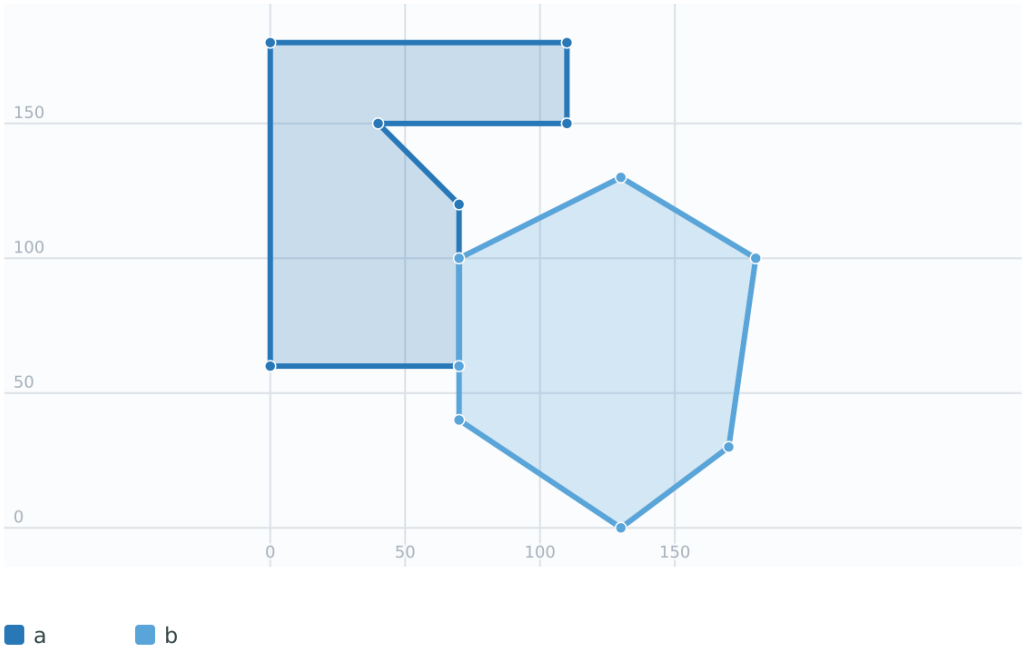
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190))'::geometry AS a,
  'POLYGON ((80 110,140 140,190 110,180 40,140 10,80 50,80 70,80 110))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two Polygons touching at a boundary point.

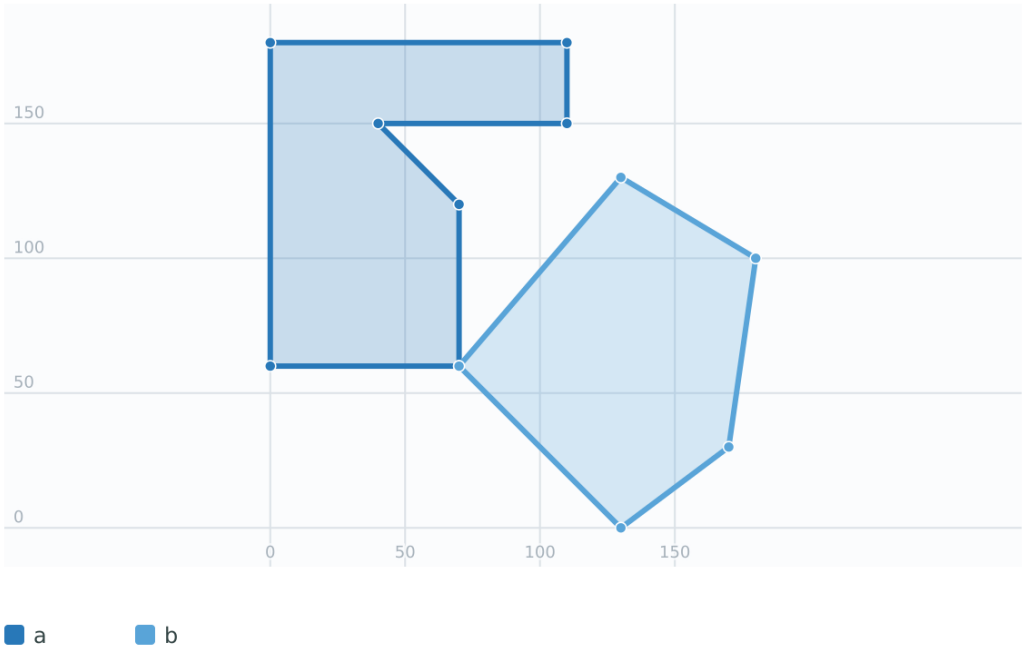
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190))'::geometry AS a,
  'POLYGON ((140 140,190 110,180 40,140 10,80 70,140 140))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon and a LineString touching at the line endpoint.

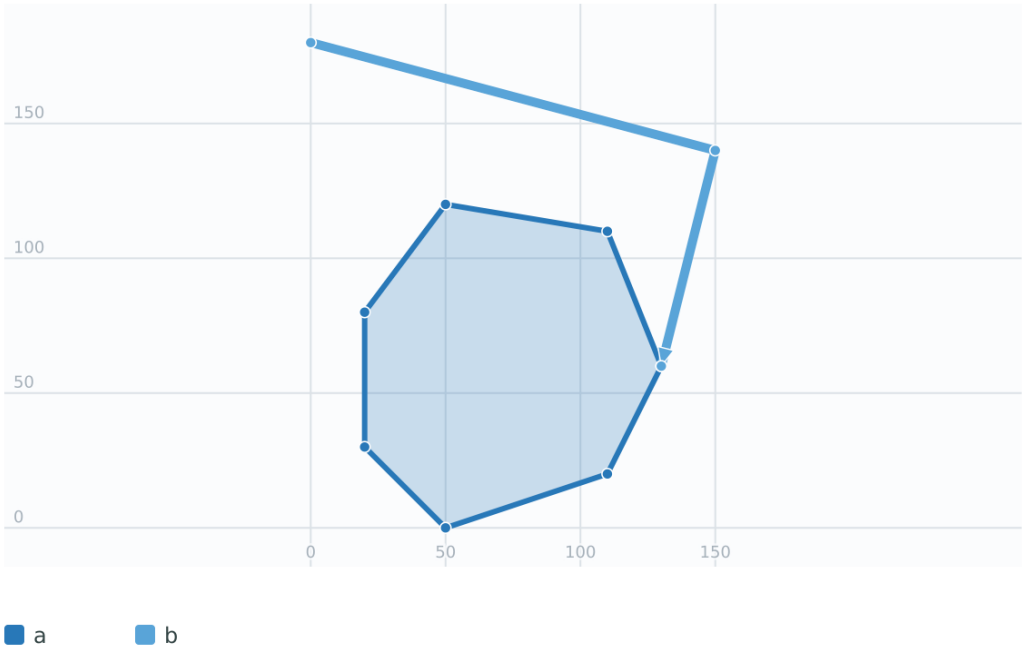
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((80 10,50 40,50 90,80 130,140 120,160 70,140 30,80 10))'::geometry AS a,
  'LINESTRING (30 190,180 150,160 70)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two LineStrings touching at their endpoints.

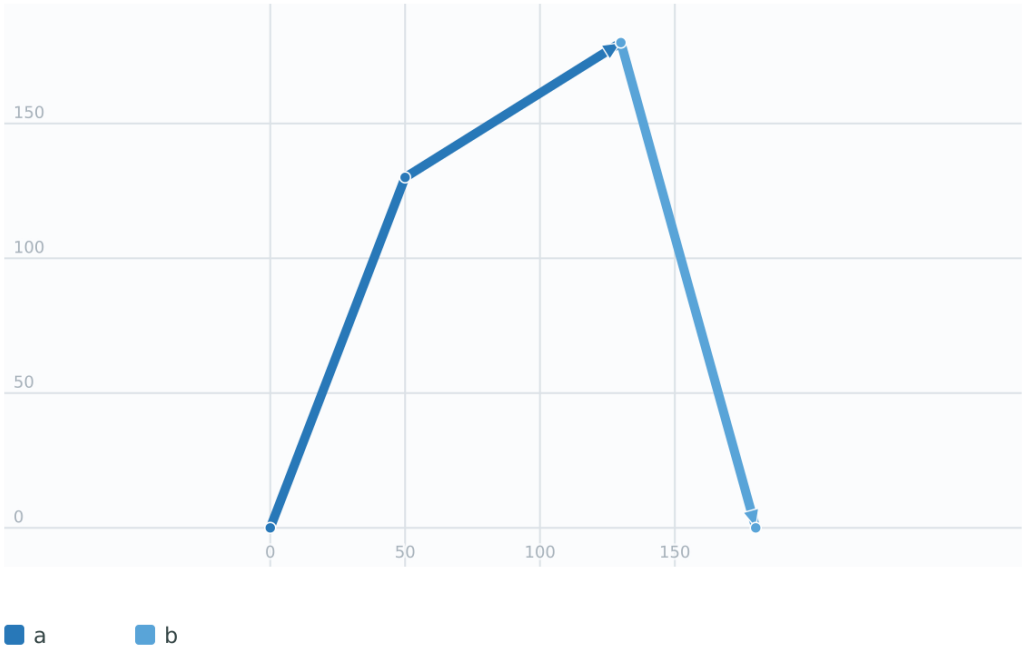
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'LINESTRING (10 10,60 140,140 190)'::geometry AS a,
  'LINESTRING (140 190,190 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two LineStrings touching where an endpoint meets the other line.

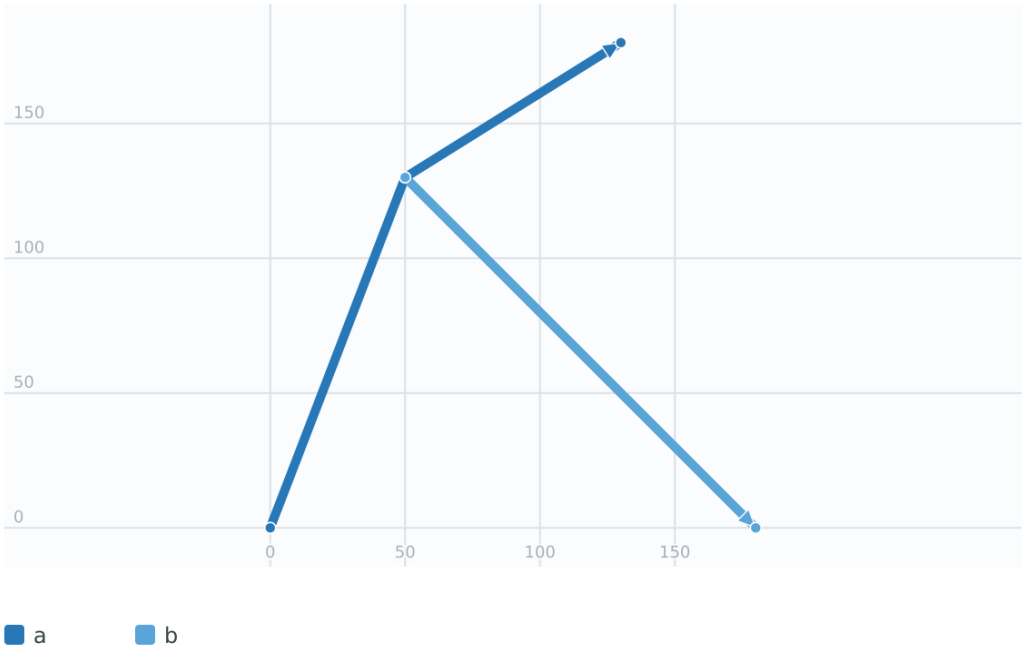
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'LINESTRING (10 10,60 140,140 190)'::geometry AS a,
  'LINESTRING (60 140,190 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Point touching a Polygon boundary.

Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((30 30,10 110,40 190,120 170,180 180,170 100,130 10,30 30))'::geometry AS a,
  'POINT (170 100)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



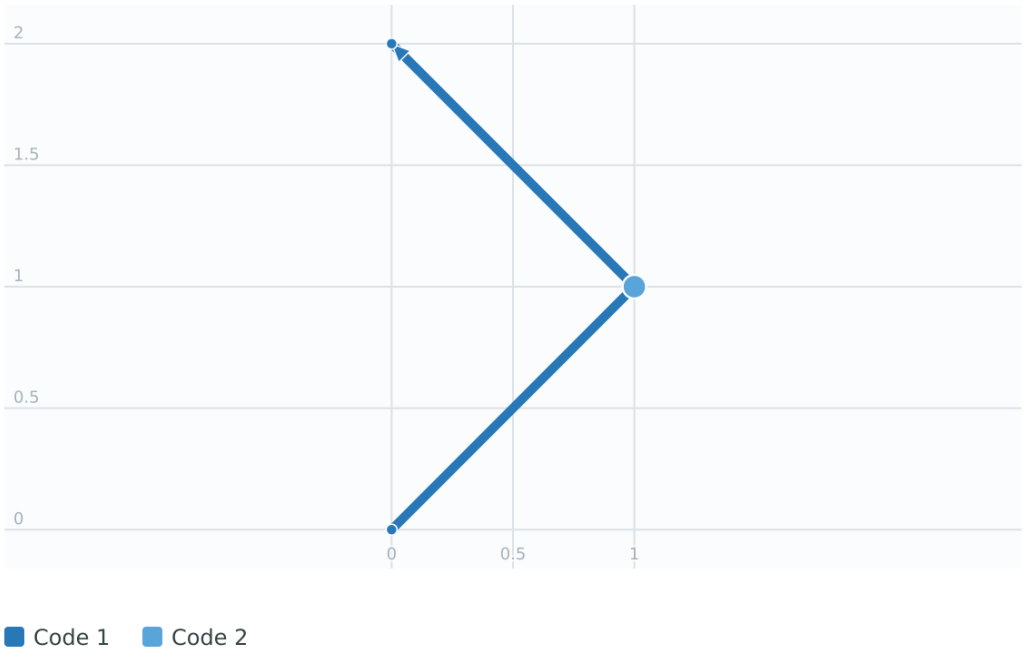
Code

```
SELECT ST_Touches('LINESTRING(0 0,1 1,0 2)'::geometry, 'POINT(1 1)'::geometry);
```

Output

```
f
```

Figure



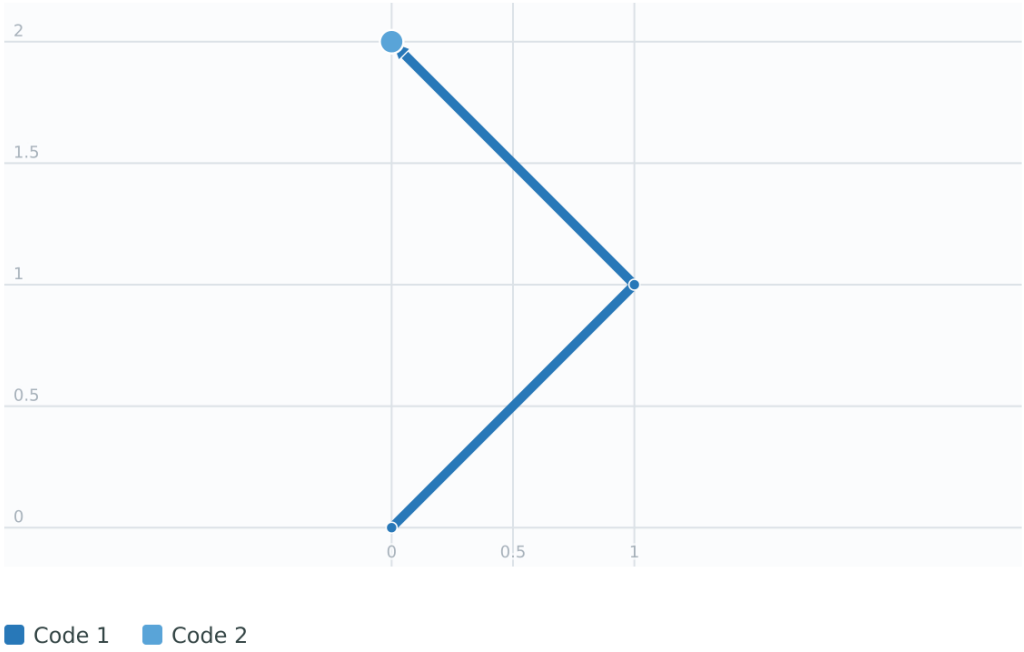
Code

```
SELECT ST_Touches('LINESTRING(0 0,1 1,0 2)'::geometry, 'POINT(0 2)'::geometry);
```

Output

```
t
```

Figure



7.11.1.16 ST_Within

ST_Within — Tests if every point of A lies in B, and their interiors have a point in common

Synopsis

boolean **ST_Within**(geometry A, geometry B);

Description

Returns TRUE if geometry A is within geometry B. A is within B if and only if all points of A lie inside (i.e. in the interior or boundary of) B (or equivalently, no points of A lie in the exterior of B), and the interiors of A and B have at least one point in common.

For this function to make sense, the source geometries must both be of the same coordinate projection, having the same SRID.

In mathematical terms: $ST\_Within(A, B) \Leftrightarrow (A \bigcap B = A) \wedge (Int(A) \bigcap Int(B) \neq \varnothing)$

The within relation is reflexive: every geometry is within itself. The relation is antisymmetric: if `ST_Within(A,B) = true` and `ST_Within(B,A) = true`, then the two geometries must be topologically equal (`ST_Equals(A,B) = true`).

ST_Within is the converse of ST_Contains. So, `ST_Within(A,B) = ST_Contains(B,A)`.



Note
Because the interiors must have a common point, a subtlety of the definition is that lines and points lying fully in the boundary of polygons or lines are *not* within the geometry. For further details see Subtleties of OGC Covers, Contains, Within. The ST_CoveredBy predicate provides a more inclusive relationship.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Within`.

Performed by the GEOS module

Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



Important
Do not use this function with invalid geometries. You will get unexpected results.

NOTE: this is the "allowable" version that returns a boolean, not an integer.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3 - a.Relate(b, 'T*F**F***')

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.30

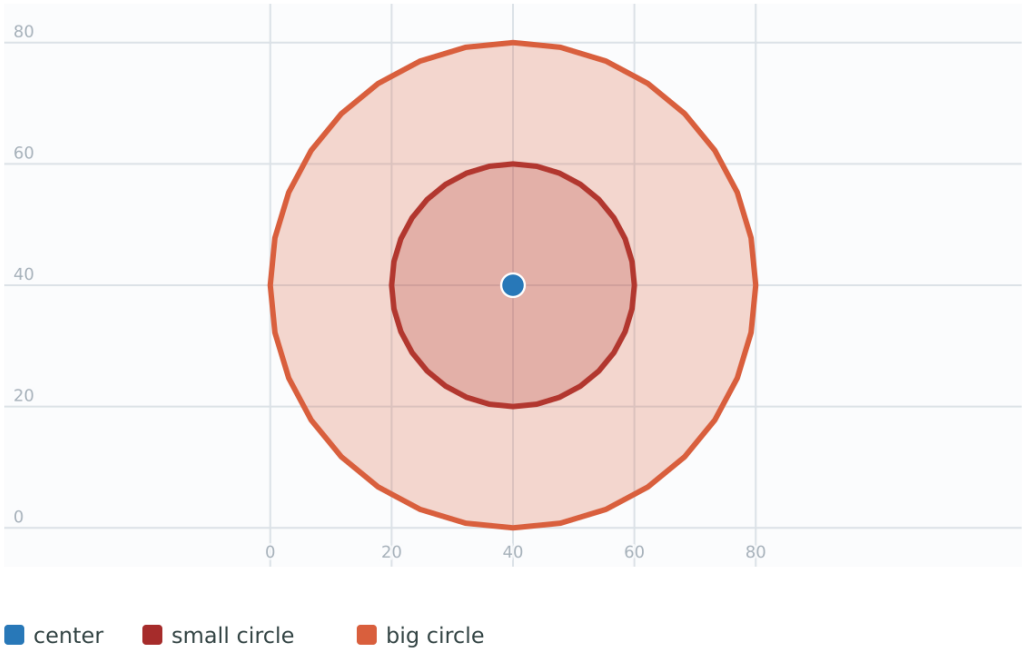
Examples

This example defines two concentric circles once in a CTE so the same inputs can be rendered and then reused in related predicate checks.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 20) AS smallc,
         ST_Buffer(center, 40) AS bigc
  FROM (SELECT 'POINT(50 50)'::geometry AS center) AS p
)
SELECT smallc AS small_circle,
       bigc AS big_circle
FROM circles;
```

Figure



Using the same circles, `ST_Within` is reflexive on each circle, false from the larger circle into the smaller one, and true for `ST_Union` only when the union does not extend beyond the larger circle.

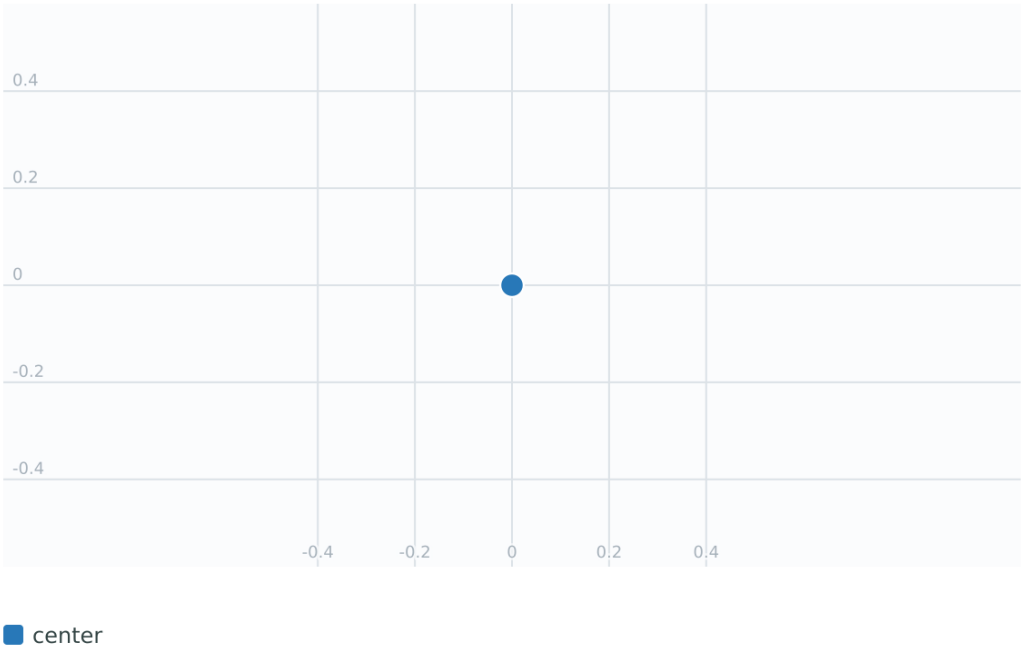
Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 20) AS smallc,
         ST_Buffer(center, 40) AS bigc
  FROM (SELECT 'POINT(50 50)'::geometry AS center) AS p
)
SELECT
  ST_Within(smallc, smallc) AS smallinsmall,
  ST_Within(smallc, bigc) AS smallinbig,
  ST_Within(bigc, smallc) AS biginsmall,
  ST_Within(ST_Union(smallc, bigc), bigc) AS unioninbig,
  ST_Within(bigc, ST_Union(smallc, bigc)) AS biginunion,
  ST_Equals(bigc, ST_Union(smallc, bigc)) AS bigisunion
FROM circles;
```

Output

```
t | t | f | t | t | t
```

Figure



See Also

ST_Contains, ST_CoveredBy, ST_Equals, ST_IsValid

7.11.2 Distance Relationships

7.11.2.1 ST_3DDWithin

ST_3DDWithin — Tests if two 3D geometries are within a given 3D distance

Synopsis

boolean **ST_3DDWithin**(geometry g1, geometry g2, double precision distance_of_srid);

Description

Returns true if the 3D distance between two geometry values is no larger than distance `distance_of_srid`. The distance is specified in units defined by the spatial reference system of the geometries. For this function to make sense the source geometries must be in the same coordinate system (have the same SRID).



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.


This function supports 3d and will not drop the z-index.


This function supports Polyhedral surfaces.

This method implements the SQL/MM specification. SQL-MM ?

Availability: 2.0.0

Examples

Compare the same transformed point and line in both 3D and 2D. The 3D test fails because the Z difference matters even though the 2D footprints are close enough.

Code

```
WITH sample AS (
  SELECT ST_Transform('SRID=4326;POINT(-72.1235 42.3521 4)'::geometry, 2163) AS input_point,
         ST_Transform(
           'SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)'::geometry,
           2163
         ) AS input_line
)
SELECT ST_3DDWithin(input_point, input_line, 126.8) AS within_dist_3d,
       ST_DWithin(input_point, input_line, 126.8) AS within_dist_2d,
       input_point AS input_point,
       input_line AS input_line
FROM sample;
```

Output

```
-[ RECORD 1 ↩
    ]--+----------------------------------------------------------------------------------------------------------------------------------------------------------------

within_dist_3d | f
within_dist_2d | t
input_point    | POINT Z (-11774128.9 5237160.2 4)
input_line     | LINESTRING Z (-11774339.5 5243556.8 15,-11773706.7 5215256.3 20)
```

See Also

ST_3DDFullyWithin, ST_DWithin, ST_DFullyWithin, ST_3DDistance, ST_Distance, ST_3DMaxDistance, ST_Transform

7.11.2.2 ST_3DDFullyWithin

ST_3DDFullyWithin — Tests if two 3D geometries are entirely within a given 3D distance

Synopsis

boolean **ST_3DDFullyWithin**(geometry g1, geometry g2, double precision distance);

Description

Returns true if the 3D geometries are fully within the specified distance of one another. The distance is specified in units defined by the spatial reference system of the geometries. For this function to make sense, the source geometries must both be of the same coordinate projection, having the same SRID.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

Availability: 2.0.0

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

Examples

This example compares distance-within predicates in 2D and 3D. The point is distance-within the line in 3D, but the full 3D line is not within the same distance of the point.

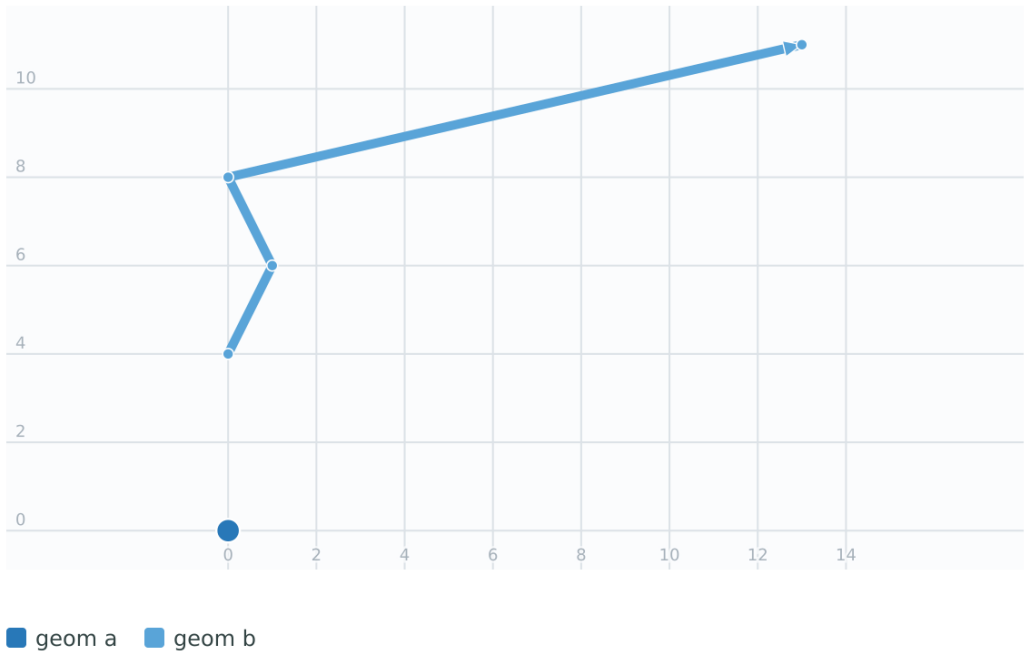
Code

```
WITH data AS (
  SELECT
    'POINT(1 1 2)'::geometry AS geom_a,
    'LINESTRING(1 5 2,2 7 20,1 9 100,14 12 3)'::geometry AS geom_b
)
SELECT
  ST_3DDFullyWithin(geom_a, geom_b, 10) AS d3dfullywithin10,
  ST_3DDWithin(geom_a, geom_b, 10) AS d3dwithin10,
  ST_DFullyWithin(geom_a, geom_b, 20) AS d2dfullywithin20,
  ST_3DDFullyWithin(geom_a, geom_b, 20) AS d3dfullywithin20
FROM data;
```

Output

```
 d3dfullywithin10 | d3dwithin10 | d2dfullywithin20 | d3dfullywithin20
------------------+-------------+------------------+------------------
 f                | t           | t                | f
```

Figure



See Also

ST_3DDWithin, ST_DWithin, ST_DFullyWithin, ST_3DMaxDistance

7.11.2.3 ST_DFullyWithin

ST_DFullyWithin — Tests if a geometry is entirely inside a distance of another

Synopsis

boolean **ST_DFullyWithin**(geometry g1, geometry g2, double precision distance);

Description

Returns true if `g2` is entirely within `distance` of `g1`. Visually, the condition is true if `g2` is contained within a `distance` buffer of `g1`. The distance is specified in units defined by the spatial reference system of the geometries.



Note
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

Availability: 1.5.0

Changed: 3.5.0 : the logic behind the function now uses a test of containment within a buffer, rather than the ST_MaxDistance algorithm. Results will differ from prior versions, but should be closer to user expectations.

Examples

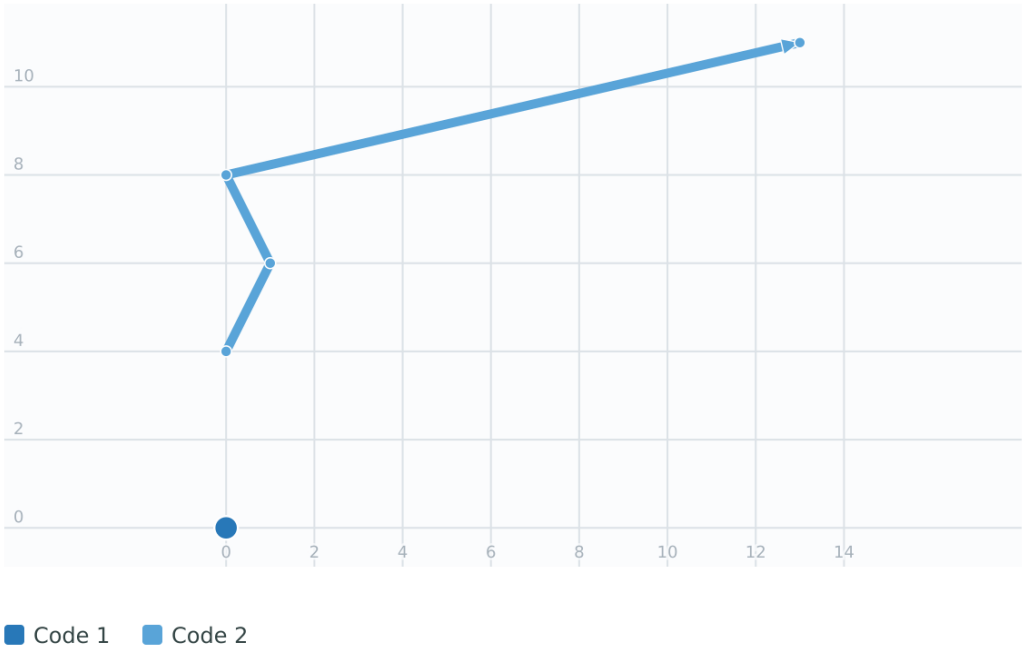
Code

```
SELECT
    ST_DFullyWithin(geom_a, geom_b, 10) AS DFullyWithin10,
    ST_DWithin(geom_a, geom_b, 10) AS DWithin10,
    ST_DFullyWithin(geom_a, geom_b, 20) AS DFullyWithin20
FROM (VALUES
    ('POINT(1 1)'::geometry, 'LINESTRING(1 5,2 7,1 9,14 12)'::geometry)
    ) AS v(geom_a, geom_b)
```

Output

```
 dfullywithin10 | dwithin10 | dfullywithin20
----------------+-----------+----------------
 f              | t         | t
```

Figure



See Also

ST_MaxDistance, ST_DWithin, ST_3DDWithin, ST_3DDFullyWithin

7.11.2.4 ST_DWithin

ST_DWithin — Tests if two geometries are within a given distance

Synopsis

boolean **ST_DWithin**(geometry g1, geometry g2, double precision distance_of_srid);
boolean **ST_DWithin**(geography gg1, geography gg2, double precision distance_meters, boolean use_spheroid = true);

Description

Returns true if the geometries are within a given distance

For geometry: The distance is specified in units defined by the spatial reference system of the geometries. For this function to make sense, the source geometries must be in the same coordinate system (have the same SRID).

For geography: units are in meters and distance measurement defaults to `use_spheroid = true`. For faster evaluation use `use_spheroid = false` to measure on the sphere.



Note
Use ST_3DDWithin for 3D geometries.



Note
This function call includes a bounding box comparison that makes use of any indexes that are available on the geometries.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

Availability: 1.5.0 support for geography was introduced

Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details.

Enhanced: 2.1.0 support for curved geometries was introduced.

Prior to 1.3, ST_Expand was commonly used in conjunction with && and ST_Distance to test for distance, and in pre-1.3.4 this function used that logic. From 1.3.4, ST_DWithin uses a faster short-circuit distance function.

Examples

This self-contained geometry example keeps the compared shapes visible while also returning the boolean results.

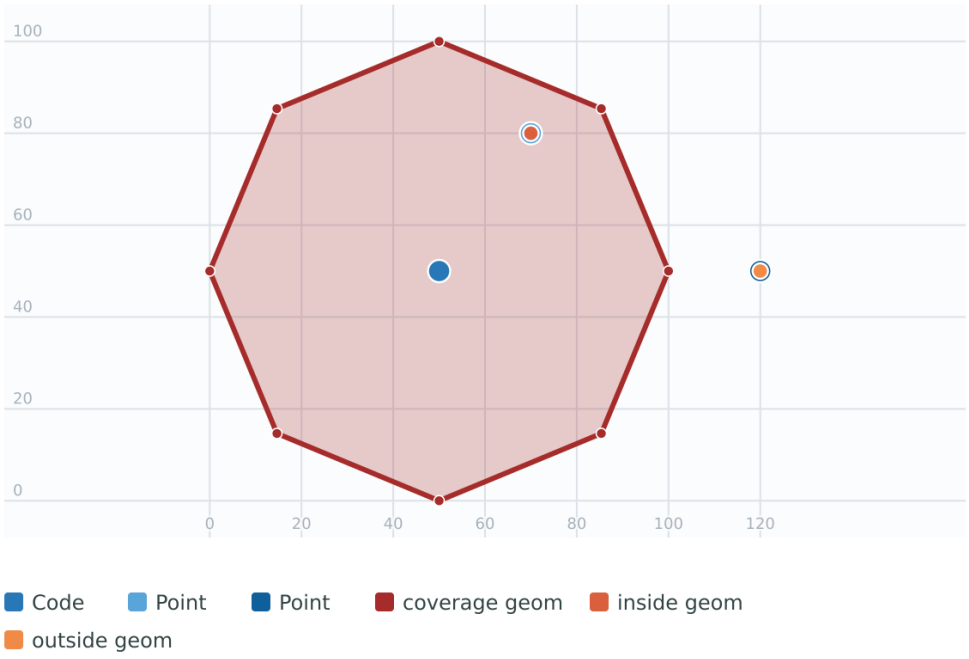
Code

```
WITH sample AS (
  SELECT ST_Buffer('POINT(100 90)'::geometry, 50, 'quad_segs=2') AS coverage,
         ST_Point(120, 120) AS inside_pt,
         ST_Point(170, 90) AS outside_pt
)
SELECT ST_DWithin(coverage, inside_pt, 0) AS inside_now,
       ST_DWithin(coverage, outside_pt, 10) AS outside_within_10,
       coverage AS coverage_geom,
       inside_pt AS inside_geom,
       outside_pt AS outside_geom
FROM sample;
```

Output

```
-[ RECORD 1 ]-----
inside_now        | t
outside_within_10 | f
coverage_geom     | POLYGON((150 90,135 55,100 40,65 55,50 90,65 125,100 140,135 125,150 90))
inside_geom       | POINT(120 120)
outside_geom      | POINT(170 90)
```

Figure



Find the nearest hospital to each school. That is within 3000 units of the school. We do an ST_DWithin search to utilize indexes to limit our search list. That the non-indexable ST_Distance needs to process. If the units of the spatial reference is meters then units would be meters.

Code

```
SELECT DISTINCT ON (s.gid) s.gid, s.school_name, s.geom, h.hospital_name
  FROM schools s
    LEFT JOIN hospitals h ON ST_DWithin(s.geom, h.geom, 3000)
  ORDER BY s.gid, ST_Distance(s.geom, h.geom);
```

The schools with no close hospitals. Find all schools with no hospital within 3000 units away from the school. Units are in units of spatial reference (for example meters, feet, or degrees).

Code

```
SELECT s.gid, s.school_name
  FROM schools s
    LEFT JOIN hospitals h ON ST_DWithin(s.geom, h.geom, 3000)
  WHERE h.gid IS NULL;
```

Find locations more than 3000 units away from their nearest hospital. Use `NOT EXISTS` so that a location is returned only when no nearby hospital exists.

Code

```
SELECT l.location_name
  FROM locations l
  WHERE NOT EXISTS (
    SELECT 1
    FROM hospitals h
    WHERE ST_DWithin(l.geom, h.geom, 3000)
  );
```

Find broadcasting towers that a receiver with limited range can receive. Data is geometry in Spherical Mercator (SRID=3857), ranges are approximate.

Code

```
CREATE INDEX ON broadcasting_towers USING gist (geom);
CREATE INDEX ON broadcasting_towers USING gist (ST_Expand(geom, sending_range));
```

Query towers that a 4-kilometer receiver in Minsk Hackerspace can get. Note: two conditions are required because the shorter `LEAST(b.sending_range, 4000)` will not use an index.

Code

```
SELECT b.tower_id, b.geom
  FROM broadcasting_towers b
  WHERE ST_DWithin(b.geom, 'SRID=3857;POINT(3072163.4 7159374.1)', 4000)
    AND ST_DWithin(b.geom, 'SRID=3857;POINT(3072163.4 7159374.1)', b.sending_range);
```

See Also

ST_Distance, ST_3DDWithin

7.11.2.5 ST_PointInsideCircle

ST_PointInsideCircle — Tests if a point geometry is inside a circle defined by a center and radius

Synopsis

boolean **ST_PointInsideCircle**(geometry a_point, float center_x, float center_y, float radius);

Description

Returns true if the geometry is a point and is inside the circle with center `center_x`,`center_y` and radius `radius`.



Warning
Does not use spatial indexes. Use ST_DWithin instead.

Availability: 1.2

Changed: 2.2.0 In prior versions this was called ST_Point_Inside_Circle

Examples

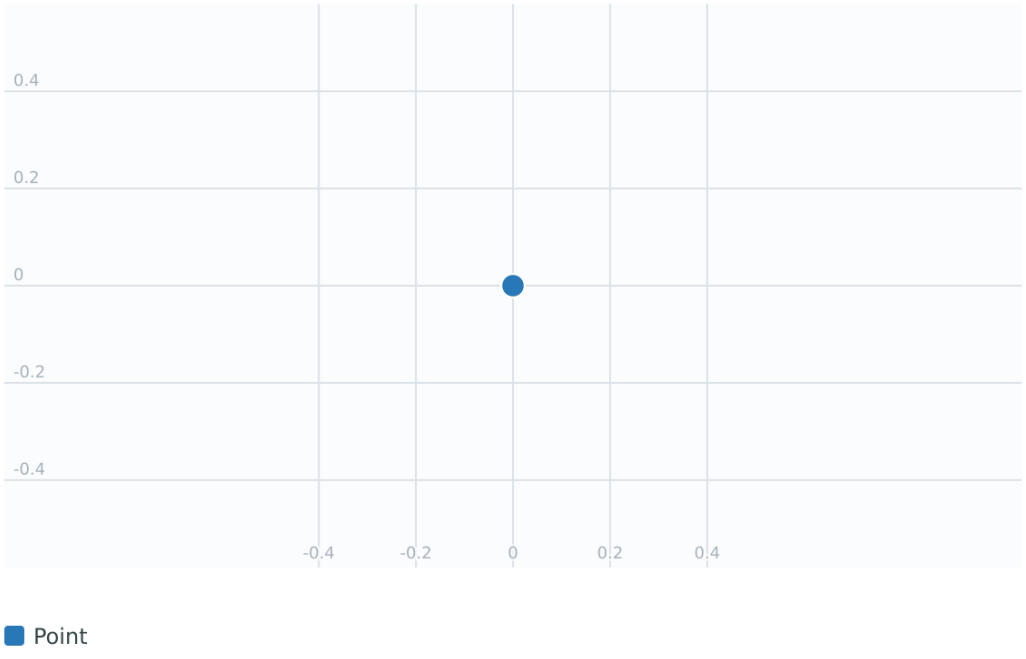
Code

```
SELECT ST_PointInsideCircle(ST_Point(1, 2), 0.5, 2, 3);
```

Output

```
t
```

Figure



See Also

ST_DWithin

7.12 Measurement Functions

7.12.1 ST_Area

ST_Area — Returns the area of a polygonal geometry.

Synopsis

float **ST_Area**(geometry g1);
float **ST_Area**(geography geog, boolean use_spheroid = true);

Description

Returns the area of a polygonal geometry. For geometry types a 2D Cartesian (planar) area is computed, with units specified by the SRID. For geography types by default area is determined on a spheroid with units in square meters. To compute the area using the faster but less accurate spherical model use `ST_Area(geog,false)`.

Enhanced: 2.0.0 - support for 2D polyhedral surfaces was introduced.

Enhanced: 2.2.0 - measurement on spheroid performed with GeographicLib for improved accuracy and robustness. Requires PROJ >= 4.9.0 to take advantage of the new feature.

Changed: 3.0.0 - does not depend on SFCGAL anymore.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

 This method implements the SQL/MM specification. SQL-MM 3: 8.1.2, 9.5.3

 This function supports Polyhedral surfaces.



Note
For polyhedral surfaces, only supports 2D polyhedral surfaces (not 2.5D). For 2.5D, may give a non-zero answer, but only for the faces that sit completely in XY plane.

Examples

Return area in square feet for a plot of Massachusetts land and multiply by conversion to get square meters. Note this is in square feet because EPSG:2249 is Massachusetts State Plane Feet

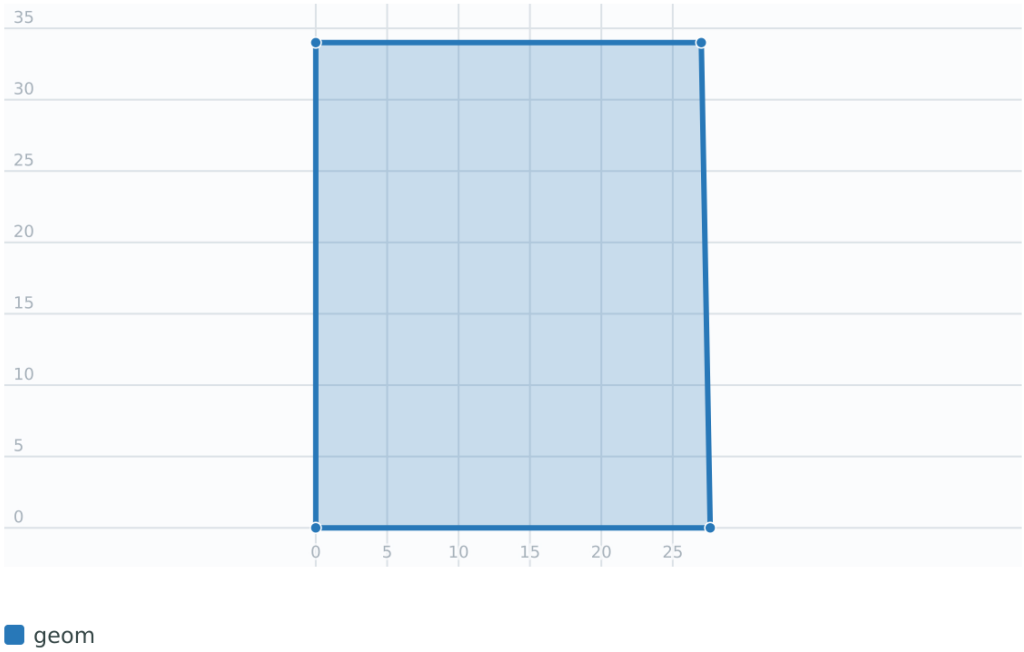
Code

```
SELECT ST_Area(geom) AS sqft,
       ST_Area(geom) * 0.3048 ^ 2 AS sqm
FROM (
  SELECT 'SRID=2249;POLYGON((743238 2967416,743238 2967450,
          743265 2967450,743265.625 2967416,743238 2967416))'::geometry AS geom
) AS subquery;
```

Output

```
 sqft   |     sqm
--------+-------------
928.625 | 86.27208552
```

Figure



Return area square feet and transform to Massachusetts state plane meters (EPSG:26986) to get square meters. Note this is in square feet because 2249 is Massachusetts State Plane Feet and transformed area is in square meters since EPSG:26986 is state plane Massachusetts meters.

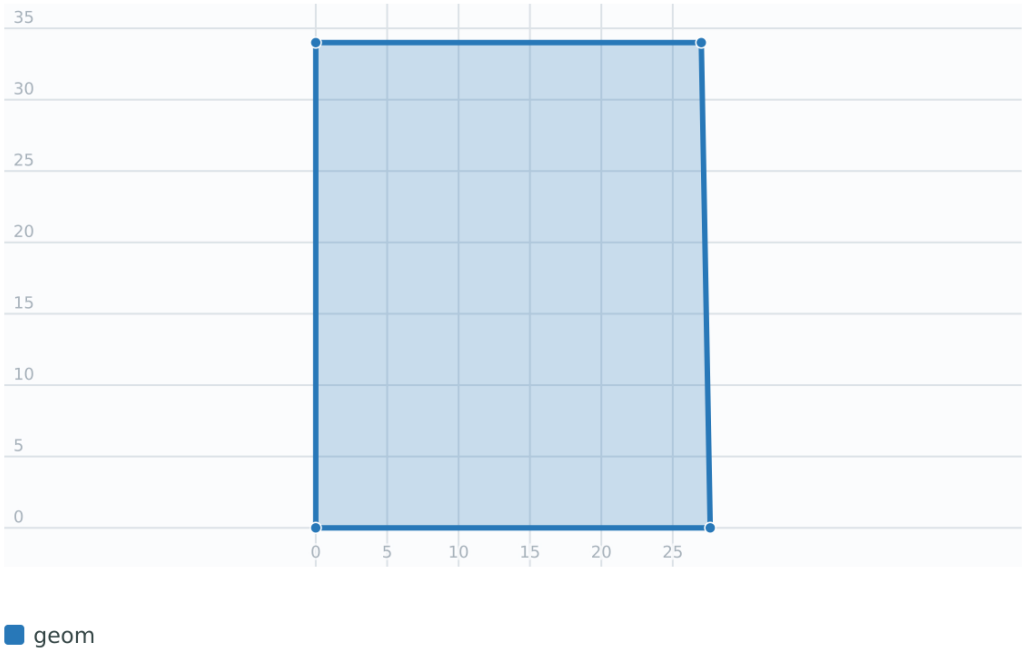
Code

```
SELECT ST_Area(geom) AS sqft,
       round(ST_Area(ST_Transform(geom, 26986))::numeric, 6) AS sqm
FROM (
  SELECT 'SRID=2249;POLYGON((743238 2967416,743238 2967450,
          743265 2967450,743265.625 2967416,743238 2967416))'::geometry AS geom
) AS subquery;
```

Output

```
 sqft    |    sqm
---------+-----------
928.625 | 86.272431
```

Figure



Return area square feet and square meters using geography data type. Note that we transform to our geometry to geography (before you can do that make sure your geometry is in WGS 84 long lat 4326). Geography always measures in meters. This is just for demonstration to compare. Normally your table will be stored in geography data type already.

Code

```
SELECT round((ST_Area(geog) / 0.3048 ^ 2)::numeric, 2) AS sqft_spheroid,
       round((ST_Area(geog, false) / 0.3048 ^ 2)::numeric, 2) AS sqft_sphere,
       round(ST_Area(geog)::numeric, 2) AS sqm_spheroid
FROM (
  SELECT ST_Transform(
           'SRID=2249;POLYGON((743238 2967416,743238 2967450,
           743265 2967450,743265.625 2967416,743238 2967416))'::geometry,
           4326
         )::geography AS geog
) AS subquery;
```

Output

```
 sqft_spheroid | sqft_sphere | sqm_spheroid
---------------+-------------+--------------
        928.68 |      926.61 |        86.28
```

If your data is in geography already:

Code

```
SELECT ST_Area(geog) / 0.3048 ^ 2 AS sqft,
         ST_Area(the_geog) AS sqm
  FROM somegeogtable;
```

See Also

CG_3DArea, ST_GeomFromText, ST_GeographyFromText, ST_SetSRID, ST_Transform

7.12.2 ST_Azimuth

ST_Azimuth — Returns the north-based azimuth of a line between two points.

Synopsis

float **ST_Azimuth**(geometry origin, geometry target);
float **ST_Azimuth**(geography origin, geography target);

Description

Returns the azimuth in radians of the target point from the origin point, or NULL if the two points are coincident. The azimuth angle is a positive clockwise angle referenced from the positive Y axis (geometry) or the North meridian (geography): North = 0; Northeast = $\pi/4$; East = $\pi/2$; Southeast = $3\pi/4$; South = π; Southwest $5\pi/4$; West = $3\pi/2$; Northwest = $7\pi/4$.

For the geography type, the azimuth solution is known as the inverse geodesic problem.

The azimuth is a mathematical concept defined as the angle between a reference vector and a point, with angular units in radians. The result value in radians can be converted to degrees using the PostgreSQL function `degrees()`.

For offsetting lines relative to their direction, use ST_OffsetCurve. For projecting a point by distance and bearing, use ST_Project.

Availability: 1.1.0

Enhanced: 2.0.0 support for geography was introduced.

Enhanced: 2.2.0 measurement on spheroid performed with GeographicLib for improved accuracy and robustness. Requires PROJ >= 4.9.0 to take advantage of the new feature.

Examples

Geometry Azimuth in degrees

Code

```
SELECT degrees(ST_Azimuth(ST_Point(25, 45), ST_Point(75, 100))) AS degA_B,
       degrees(ST_Azimuth(ST_Point(75, 100), ST_Point(25, 45) )) AS degB_A;
```

Output

```
      dega_b       |      degb_a
-------------------+-------------------
42.27368900609371 | 222.27368900609372
```

Compass direction labels can be computed from the azimuth.

Code

```
WITH directions(deg) AS (VALUES
  (0), (45), (90), (135), (180), (225), (270), (315)
),
bearings AS (
  SELECT deg,
         degrees(ST_Azimuth(ST_Point(0, 0),
                            ST_MakePoint(sin(radians(deg)),
                                         cos(radians(deg))))) AS azimuth
  FROM directions
)
SELECT deg,
       round(azimuth::numeric, 1) AS azimuth,
       (ARRAY['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW', 'N'])
         [floor((azimuth + 22.5) / 45)::int + 1] AS direction
FROM bearings
ORDER BY deg;
```

Output

```
 deg | azimuth | direction
-----+---------+-----------
   0 |     0.0 | N
  45 |    45.0 | NE
  90 |    90.0 | E
 135 |   135.0 | SE
 180 |   180.0 | S
 225 |   225.0 | SW
 270 |   270.0 | W
 315 |   315.0 | NW
```

Azimuth from Point(25, 45) to Point(75, 100), shown against the positive Y axis (north).

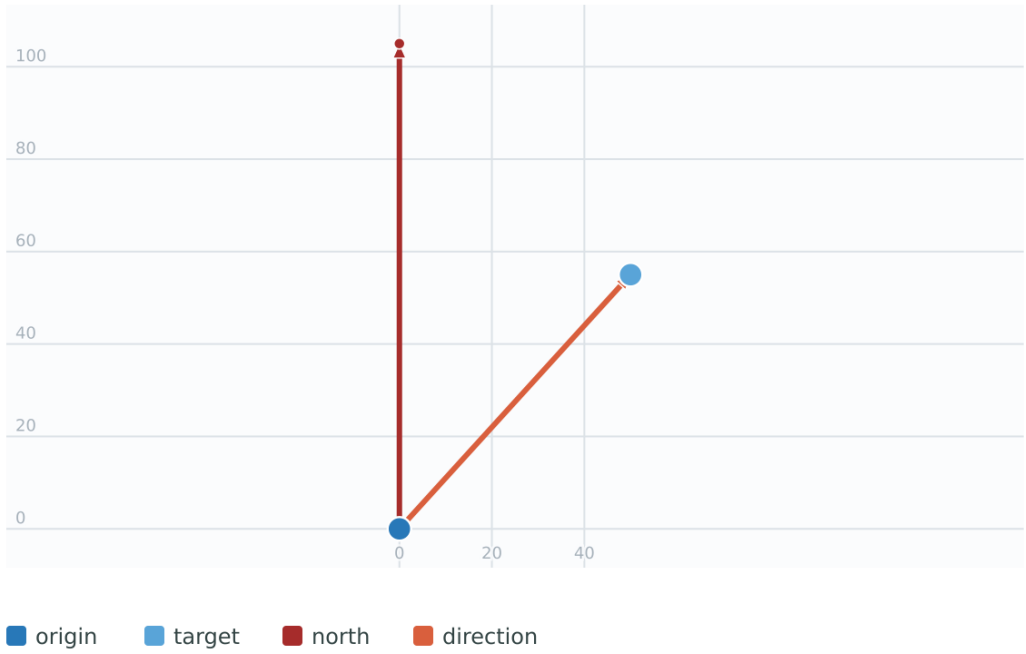
Code

```
WITH points AS (
  SELECT 'POINT(25 45)'::geometry AS origin,
         'POINT(75 100)'::geometry AS target
)
SELECT round(degrees(ST_Azimuth(origin, target))::numeric, 6) AS degrees,
       ST_MakeLine(origin, ST_Translate(origin, 0, 105)) AS north,
       ST_MakeLine(origin, target) AS direction
FROM points;
```

Output

```
42.273689 | LINESTRING(25 45,25 150) | LINESTRING(25 45,75 100)
```

Figure



Reverse azimuth from Point(75, 100) to Point(25, 45).

Code

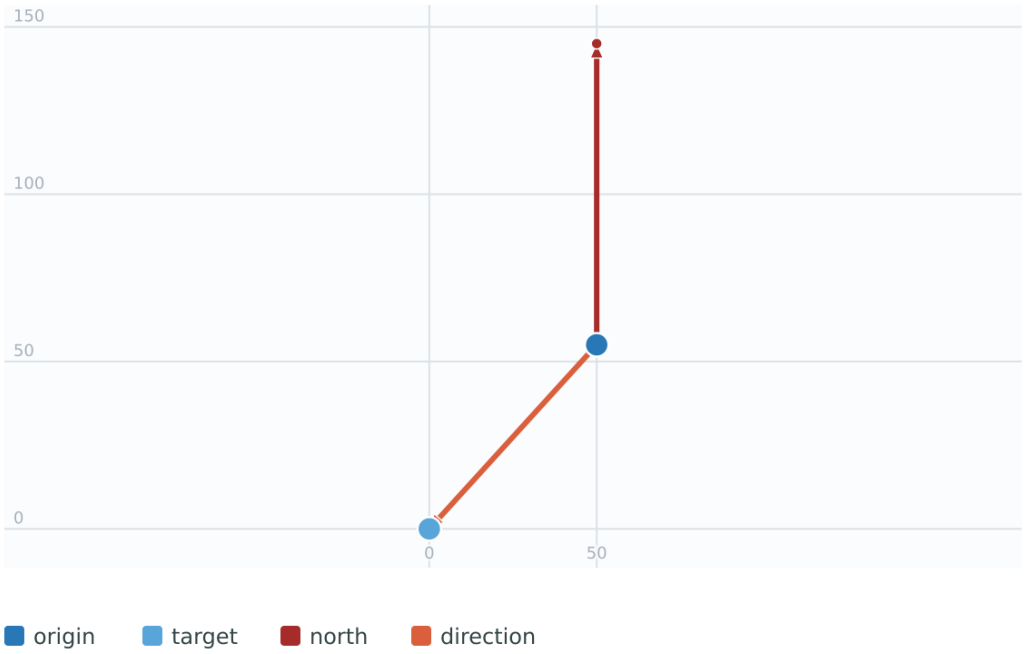
```
WITH points AS (
  SELECT 'POINT(75 100)'::geometry AS origin,
         'POINT(25 45)'::geometry AS target
)
SELECT round(degrees(ST_Azimuth(origin, target))::numeric, 6) AS degrees,
```

```
       ST_MakeLine(origin, ST_Translate(origin, 0, 90)) AS north,
       ST_MakeLine(origin, target) AS direction
FROM points;
```

Output

```
222.273689 | LINESTRING(75 100,75 190) | LINESTRING(75 100,25 45)
```

Figure



See Also

ST_Angle, ST_Point, ST_Translate, ST_Project, PostgreSQL Math Functions

7.12.3 ST_Angle

ST_Angle — Returns the angle between two vectors defined by 3 or 4 points, or 2 lines.

Synopsis

float **ST_Angle**(geometry point1, geometry point2, geometry point3, geometry point4);
float **ST_Angle**(geometry line1, geometry line2);

Description

Computes the clockwise angle between two vectors.

Variant 1: computes the angle enclosed by the points P1-P2-P3. If a 4th point provided computes the angle points P1-P2 and P3-P4

Variant 2: computes the angle between two vectors S1-E1 and S2-E2, defined by the start and end points of the input lines

The result is a positive angle between 0 and 2π radians. The radian result can be converted to degrees using the PostgreSQL function `degrees()`.

Note that `ST_Angle(P1,P2,P3) = ST_Angle(P2,P1,P2,P3)`.

Availability: 2.5.0

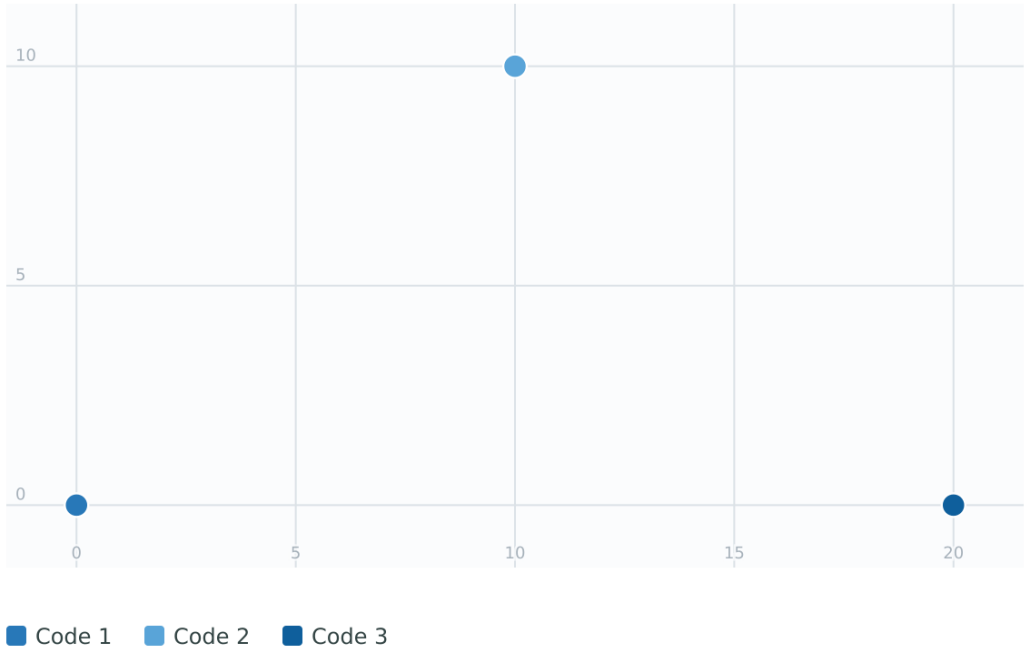
Examples

Angle between three points

Code

```
SELECT degrees(ST_Angle('POINT(0 0)', 'POINT(10 10)', 'POINT(20 0)'));
```

Figure

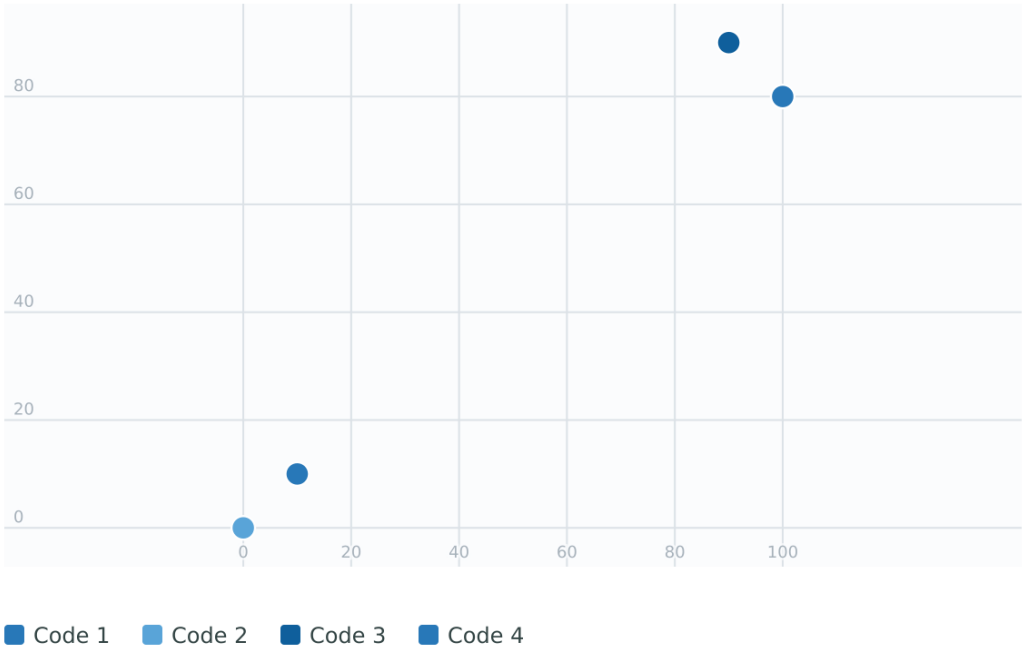


Angle between vectors defined by four points

Code

```
SELECT degrees(ST_Angle('POINT (10 10)', 'POINT (0 0)', 'POINT(90 90)', 'POINT (100 80)'));
```

Figure

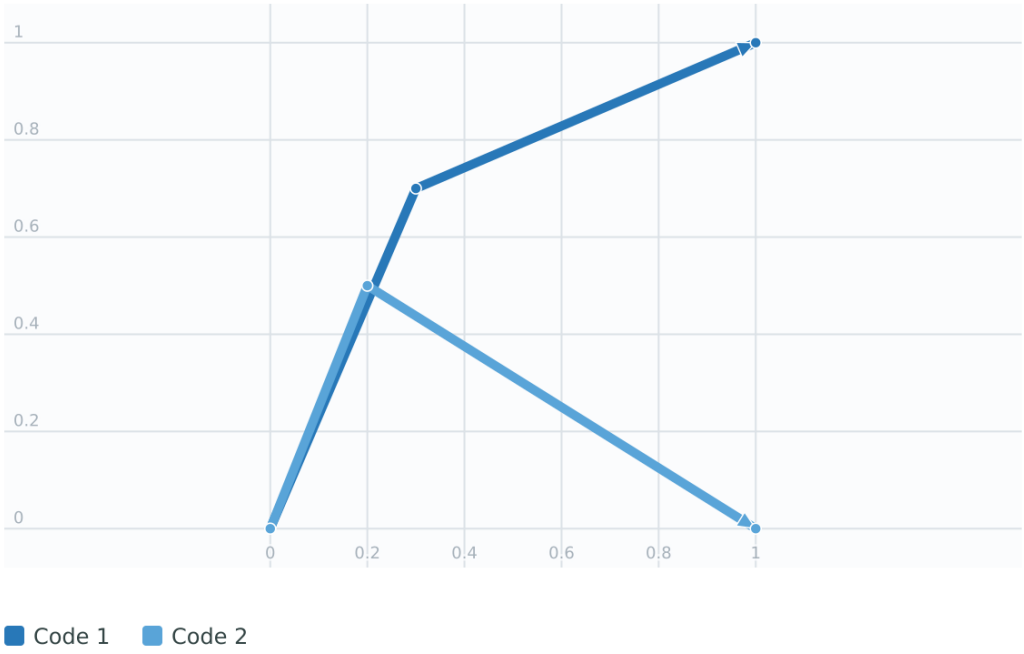


Angle between vectors defined by the start and end points of lines

Code

```
SELECT degrees(ST_Angle('LINESTRING(0 0,0.3 0.7,1 1)', 'LINESTRING(0 0,0.2 0.5,1 0)'));
```

Figure



See Also

ST_Azimuth

7.12.4 ST_ClosestPoint

ST_ClosestPoint — Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.

Synopsis

geometry **ST_ClosestPoint**(geometry geom1, geometry geom2);
geography **ST_ClosestPoint**(geography geom1, geography geom2, boolean use_spheroid = true);

Description

Returns the 2-dimensional point on `geom1` that is closest to `geom2`. This is the first point of the shortest line between the geometries (as computed by ST_ShortestLine).



Note
If you have a 3D Geometry, you may prefer to use ST_3DClosestPoint.

Enhanced: 3.4.0 - Support for geography.

Availability: 1.5.0

Examples

The closest point for a Point and a LineString is the point itself. The closest point for a LineString and a Point is a point on the line.

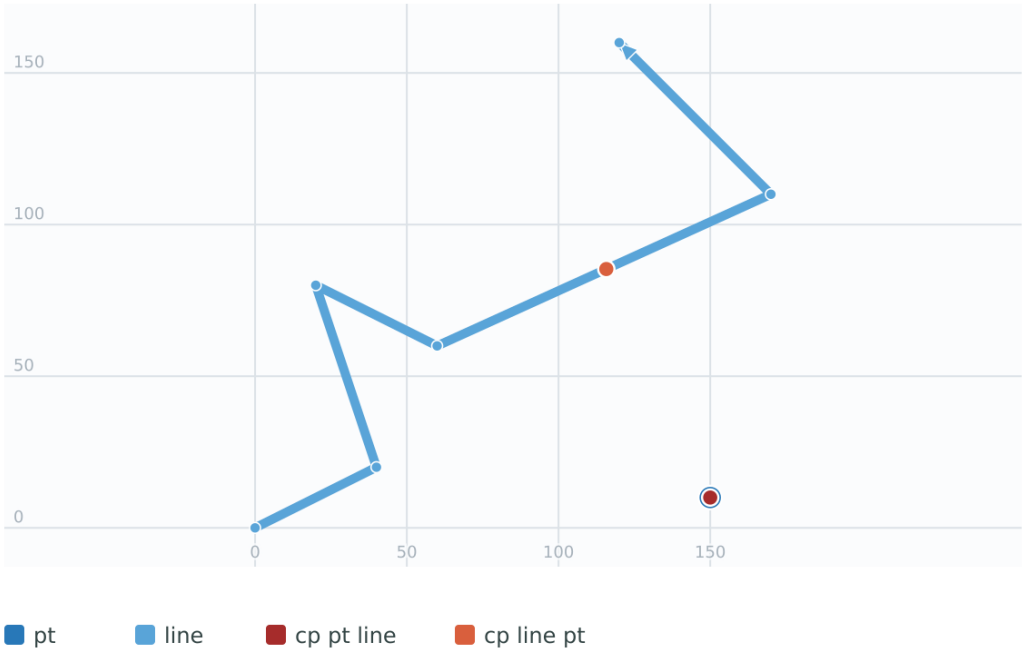
Code

```
SELECT ST_ClosestPoint(pt, line) AS cp_pt_line, ST_ClosestPoint(line, pt) AS cp_line_pt
    FROM (SELECT 'POINT (160 40)'::geometry AS pt,
                 'LINESTRING (10 30,50 50,30 110,70 90,180 140,130 190)'::geometry AS line ) ↩
     AS t;
```

Output

```
   cp_pt_line    |               cp_line_pt
-----------------+-----------------------------------------
 POINT(160 40)   | POINT(125.75342465753425 115.34246575342466)
```

Figure

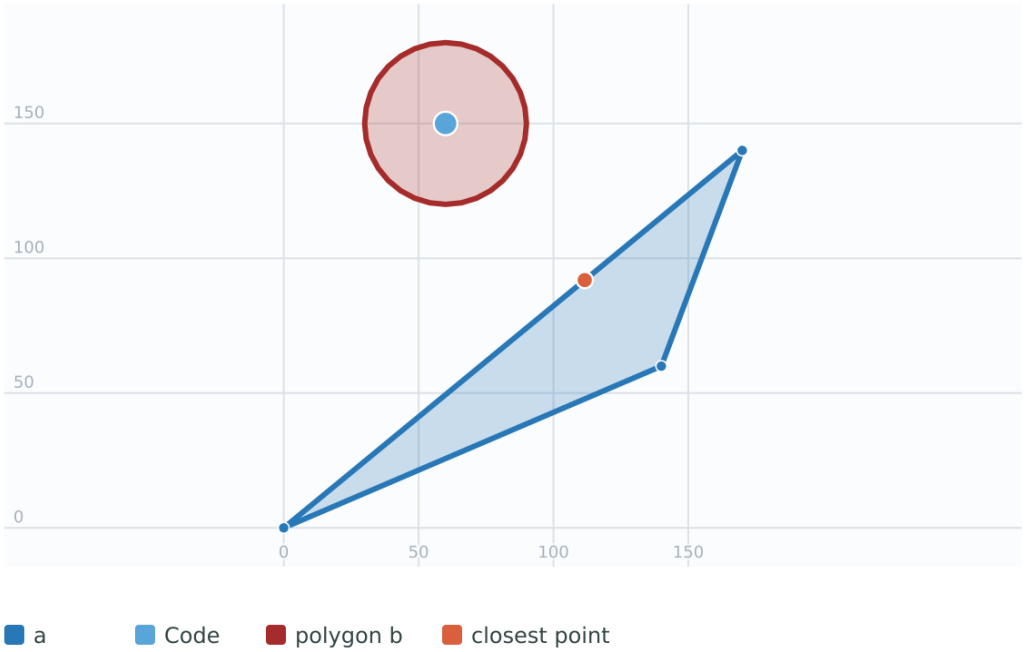


Find the closest point on polygon A to polygon B.

Code

```
WITH source AS (
  SELECT 'POLYGON ((190 150,20 10,160 70,190 150))'::geometry AS a,
         ST_Buffer('POINT(80 160)'::geometry, 30) AS b
)
SELECT b AS polygon_b,
       ST_ClosestPoint(a, b) AS closest_point
FROM source;
```

Figure



See Also

ST_3DClosestPoint, ST_Distance, ST_LongestLine, ST_ShortestLine, ST_MaxDistance

7.12.5 ST_3DClosestPoint

ST_3DClosestPoint — Returns the 3D point on g1 that is closest to g2. This is the first point of the 3D shortest line.

Synopsis

geometry **ST_3DClosestPoint**(geometry g1, geometry g2);

Description

Returns the 3-dimensional point on g1 that is closest to g2. This is the first point of the 3D shortest line. The 3D length of the 3D shortest line is the 3D distance.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

Availability: 2.0.0

Changed: 2.2.0 - if 2 2D geometries are input, a 2D point is returned (instead of old behavior assuming 0 for missing Z). In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z.

Examples

Compare the 3D closest point to the 2D closest point for a LineString and a Point.

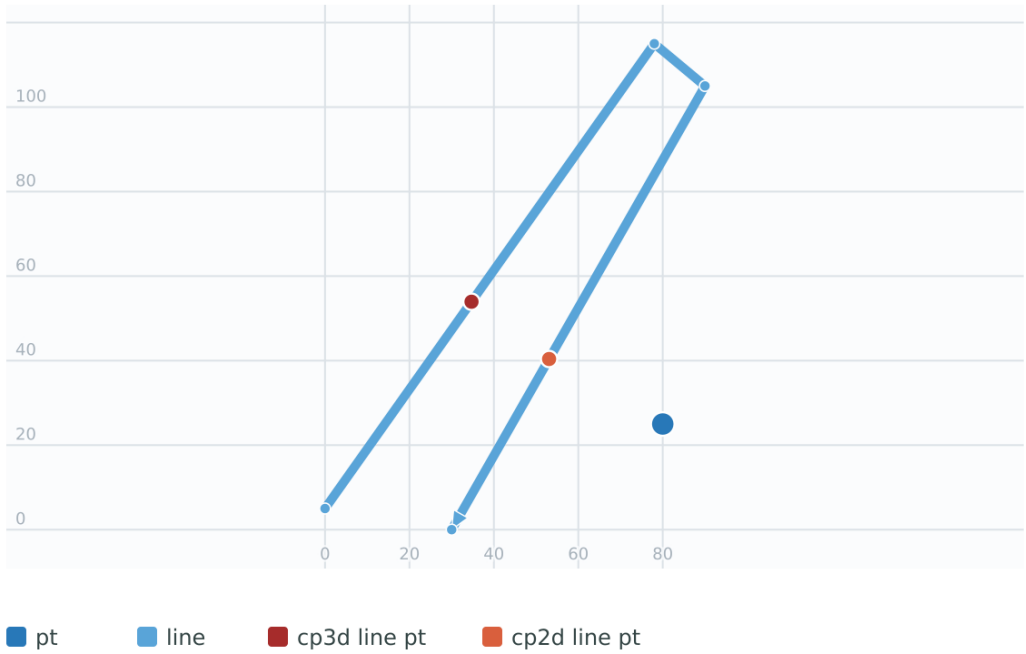
Code

```
WITH input AS (
  SELECT 'POINT(100 100 30)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 1000)'::geometry AS line
)
SELECT ST_3DClosestPoint(line, pt) AS cp3d_line_pt,
       ST_ClosestPoint(line, pt) AS cp2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+------------------------------------------------------
cp3d_line_pt   | POINT(54.69937988676193 128.93502291722837 11.547586950660556)
cp2d_line_pt   | POINT(73.07692307692307 115.38461538461539)
```

Figure



Compare the 3D closest point to the 2D closest point for a LineString and a MultiPoint.

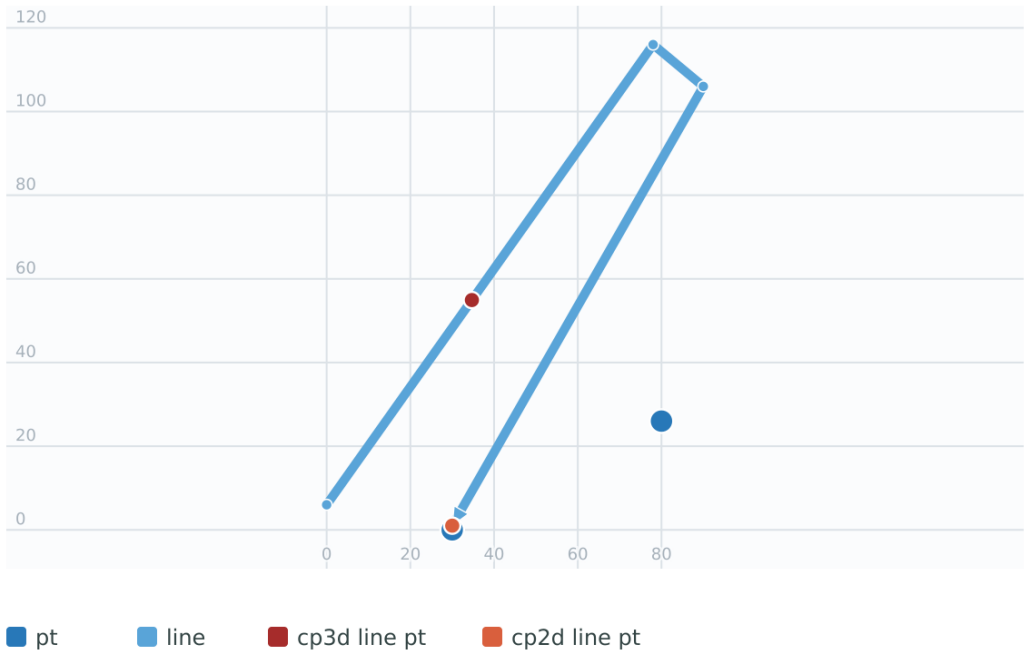
Code

```
WITH input AS (
  SELECT 'MULTIPOINT(100 100 30,50 74 1000)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 900)'::geometry AS line
)
SELECT ST_3DClosestPoint(line, pt) AS cp3d_line_pt,
       ST_ClosestPoint(line, pt) AS cp2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+------------------------------------------------------
cp3d_line_pt   | POINT(54.69937988676193 128.93502291722837 11.547586950660556)
cp2d_line_pt   | POINT(50 75)
```

Figure



Compare the 3D closest point to the 2D closest point for a Polygon and a MultiLineString.

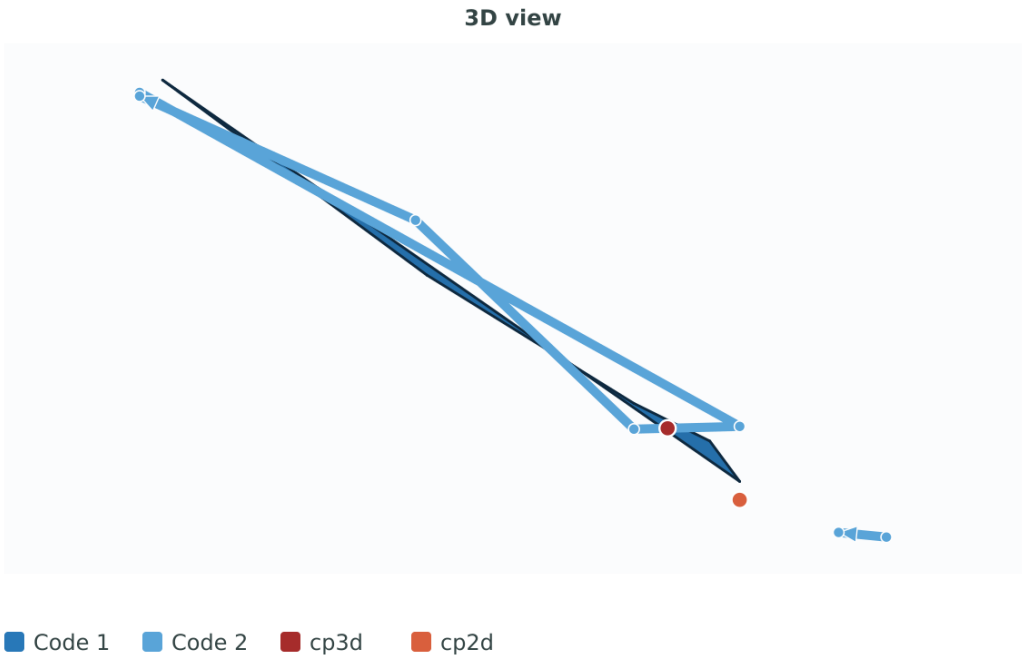
Code

```
WITH input AS (
  SELECT ST_GeomFromEWKT(
           'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))'
         ) AS poly,
         ST_GeomFromEWKT(
           'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),
                            (1 10 2,5 20 1))'
         ) AS mline
)
SELECT ST_3DClosestPoint(poly, mline) AS cp3d,
       ST_ClosestPoint(poly, mline) AS cp2d
FROM input;
```

Output

```
-[ RECORD 1 ]--+---------------------------------------
cp3d           | POINT(40.45454545454545 53.63636363636364 5)
cp2d           | POINT(20 40)
```

Figure



See Also

ST_AsEWKT, ST_ClosestPoint, ST_3DDistance, ST_3DShortestLine

7.12.6 ST_Distance

ST_Distance — Returns the distance between two geometry or geography values.

Synopsis

float **ST_Distance**(geometry g1, geometry g2);
float **ST_Distance**(geography geog1, geography geog2, boolean use_spheroid = true);

Description

For geometry types returns the minimum 2D Cartesian (planar) distance between two geometries, in projected units (spatial ref units).

For geography types defaults to return the minimum geodesic distance between two geographies in meters, compute on the spheroid determined by the SRID. If `use_spheroid` is false, a faster spherical calculation is used.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.23

This method supports Circular Strings and Curves.

Availability: 1.5.0 geography support was introduced in 1.5. Speed improvements for planar to better handle large or many vertex geometries

Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details.

Enhanced: 2.1.0 - support for curved geometries was introduced.

Enhanced: 2.2.0 - measurement on spheroid performed with GeographicLib for improved accuracy and robustness. Requires PROJ >= 4.9.0 to take advantage of the new feature.

Changed: 3.0.0 - does not depend on SFCGAL anymore.

Geometry Examples

Geometry example - units in planar degrees 4326 is WGS 84 long lat, units are degrees.

Code

```
SELECT ST_Distance(
'SRID=4326;POINT(-72.1235 42.3521)'::geometry,
'SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry );
```

Geometry example - units in meters (SRID: 3857, proportional to pixels on popular web maps). Although the value is off, nearby ones can be compared correctly, which makes it a good choice for algorithms like KNN or KMeans.

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 3857),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 3857) );
```

Geometry example - units in meters (SRID: 3857 as above, but corrected by cos(lat) to account for distortion)

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 3857),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 3857)
        ) * cosd(42.3521);
```

Geometry example - units in meters (SRID: 26986 Massachusetts state plane meters) (most accurate for Massachusetts)

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 26986),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 26986) );
```

Geometry example - units in meters (SRID: 2163 US National Atlas Equal area) (least accurate)

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 2163),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 2163) );
```

Geography Examples

Same as geometry example but note units in meters - use sphere for slightly faster and less accurate computation.

Code

```
SELECT ST_Distance(gg1, gg2) As spheroid_dist, ST_Distance(gg1, gg2, false) As sphere_dist
FROM (SELECT
    'SRID=4326;POINT(-72.1235 42.3521)'::geography as gg1,
    'SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geography as gg2
    ) As foo  ;
```

Output

```
  spheroid_dist    |   sphere_dist
-------------------+-----------------
 123.802076746848 | 123.475736916397
```

See Also

ST_3DDistance, ST_DWithin, ST_DistanceSphere, ST_DistanceSpheroid, ST_MaxDistance, ST_HausdorffDistance, ST_FrechetDistan
ST_Transform

7.12.7 ST_3DDistance

ST_3DDistance — Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.

Synopsis

float **ST_3DDistance**(geometry g1, geometry g2);

Description

Returns the 3-dimensional minimum cartesian distance between two geometries in projected units (spatial ref units).

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3

Availability: 2.0.0

Changed: 2.2.0 - In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z.

Changed: 3.0.0 - SFCGAL version removed

Examples

Geometry example - units in meters (SRID: 2163 US National Atlas Equal area) (3D point and line compared 2D point and line) Note that currently no vertical datum support so Z is not transformed and assumed to be same units as final.

Code

```
SELECT ST_3DDistance(
            ST_Transform('SRID=4326;POINT(-72.1235 42.3521 4)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)':: ↩
    geometry, 2163)
            ) As dist_3d,
            ST_Distance(
                ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)':: ↩
    geometry, 2163)
            ) As dist_2d;
```

Output

```
     dist_3d      |     dist_2d
------------------+-----------------
 127.295059324629 | 126.66425605671
```

Multilinestring and polygon both 3D and 2D distance. Same example as 3D closest point example.

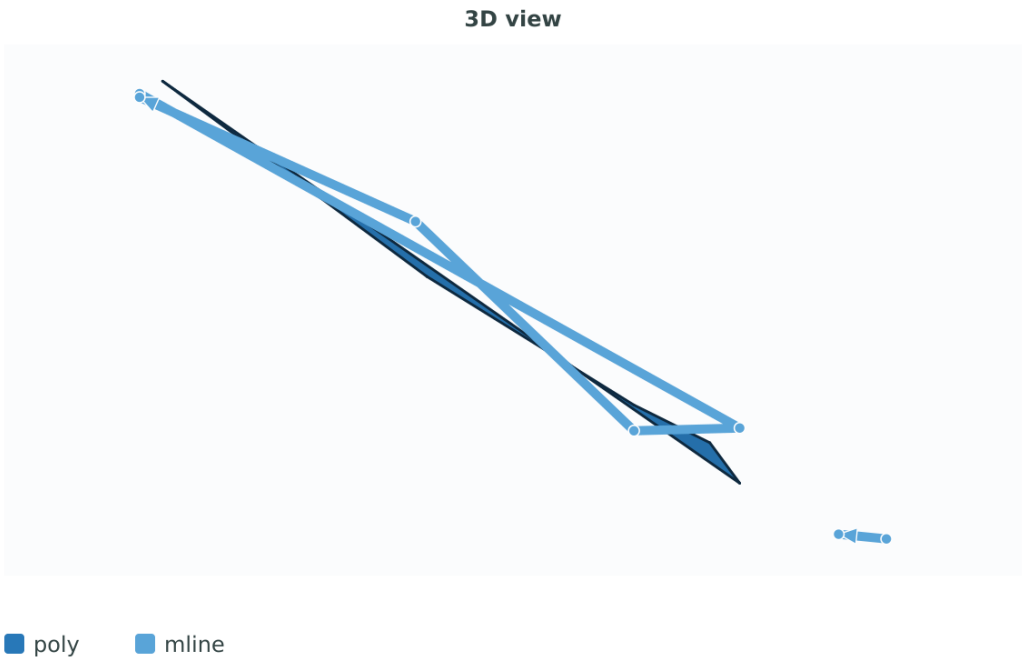
Code

```
SELECT ST_3DDistance(poly, mline) As dist3d,
    ST_Distance(poly, mline) As dist2d
        FROM (SELECT  'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))':: ↩
    geometry as poly,
                'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),(1 10 2,5  ↩
    20 1))'::geometry as mline) as foo;
```

Output

```
      dist3d       | dist2d
-------------------+--------
                 0 |      0
```

Figure



See Also

ST_Distance, ST_3DClosestPoint, ST_3DDWithin, ST_3DMaxDistance, ST_3DShortestLine, ST_Transform

7.12.8 ST_DistanceSphere

ST_DistanceSphere — Returns minimum distance in meters between two lon/lat geometries using a spherical earth model.

Synopsis

float **ST_DistanceSphere**(geometry geomlonlatA, geometry geomlonlatB, float8 radius=6371008);

Description

Returns minimum distance in meters between two lon/lat points. Uses a spherical earth and radius derived from the spheroid defined by the SRID. Faster than ST_DistanceSpheroid, but less accurate. PostGIS Versions prior to 1.5 only implemented for points.

Availability: 1.5 - support for other geometry types besides points was introduced. Prior versions only work with points.

Changed: 2.2.0 In prior versions this used to be called ST_Distance_Sphere

Examples

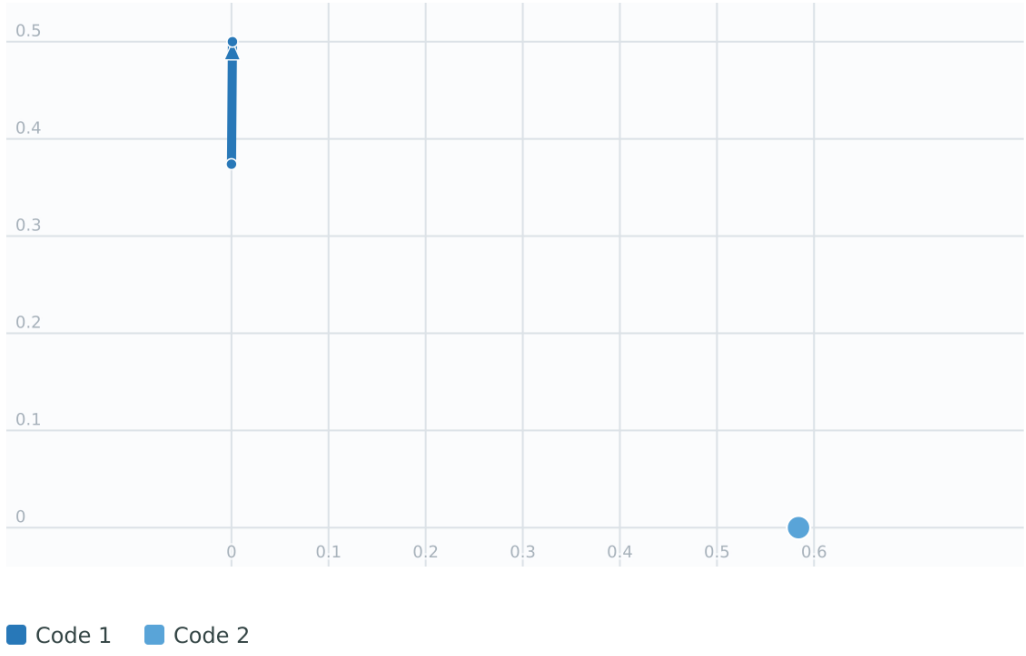
Code

```
WITH sample AS (
    SELECT
        ST_GeomFromText('LINESTRING(-118.584 38.374,-118.583 38.5)', 4326) AS geom,
        ST_GeomFromText('POINT(-118 38)', 4326) AS pt
)
SELECT
    round(ST_DistanceSphere(ST_Centroid(geom), pt)::numeric, 2) AS dist_meters,
    round(ST_Distance(
        ST_Transform(ST_Centroid(geom), 32611),
        ST_Transform(pt, 32611)
    )::numeric, 2) AS dist_utm11_meters,
    round(ST_Distance(ST_Centroid(geom), pt)::numeric, 5) AS dist_degrees,
    round(ST_Distance(
        ST_Transform(geom, 32611),
        ST_Transform(pt, 32611)
    )::numeric, 2) AS min_dist_line_point_meters
FROM sample;
```

Output

```
 dist_meters | dist_utm11_meters | dist_degrees | min_dist_line_point_meters
-------------+-------------------+--------------+----------------------------
    70424.71 |          70438.00 |      0.72900 |                   65871.18
```

Figure



See Also

ST_Distance, ST_DistanceSpheroid

7.12.9 ST_DistanceSpheroid

ST_DistanceSpheroid — Returns the minimum distance between two lon/lat geometries using a spheroidal earth model.

Synopsis

float **ST_DistanceSpheroid**(geometry geomlonlatA, geometry geomlonlatB, spheroid measurement_spheroid=WGS84);

Description

Returns minimum distance in meters between two lon/lat geometries given a particular spheroid. See the explanation of spheroids given for ST_LengthSpheroid.



Note
This function does not look at the SRID of the geometry. It assumes the geometry coordinates are based on the provided spheroid.

Availability: 1.5 - support for other geometry types besides points was introduced. Prior versions only work with points.

Changed: 2.2.0 In prior versions this was called ST_Distance_Spheroid

Examples

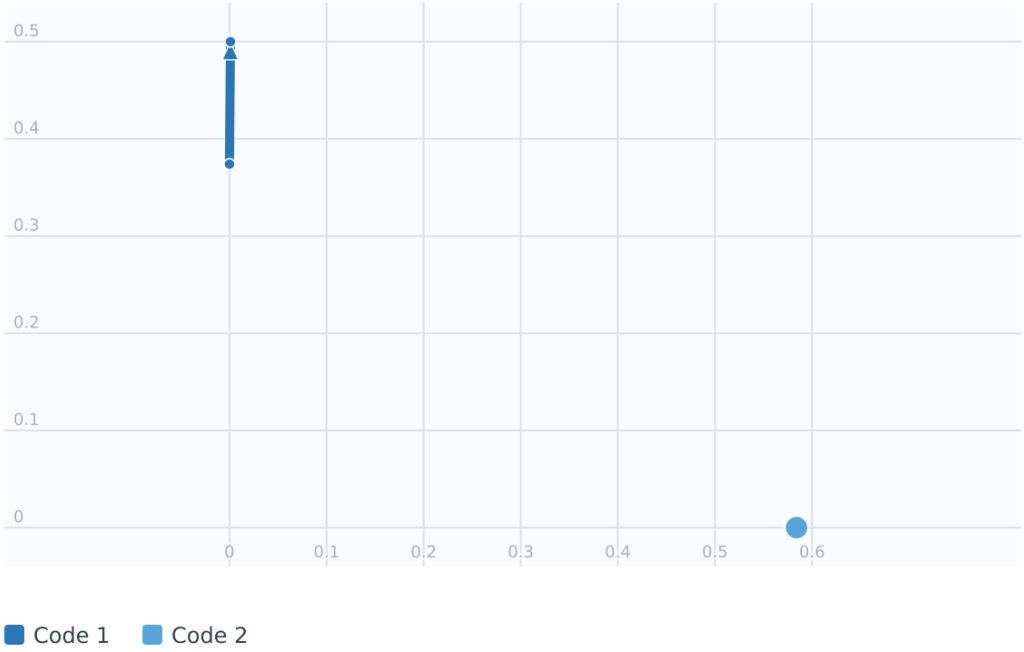
Code

```
WITH input AS (
  SELECT ST_Centroid(
           ST_GeomFromText(
             'LINESTRING(-118.584 38.374,-118.583 38.5)', 4326
           )
         ) AS source,
         ST_GeomFromText('POINT(-118 38)', 4326) AS target
)
SELECT round(
         ST_DistanceSpheroid(
           source,
           target,
           'SPHEROID["WGS 84",6378137,298.257223563]'
         )::numeric,
         2
       ) AS dist_meters_spheroid,
       round(ST_DistanceSphere(source, target)::numeric, 2)
         AS dist_meters_sphere,
       round(
         ST_Distance(
           ST_Transform(source, 32611),
           ST_Transform(target, 32611)
         )::numeric,
         2
       ) AS dist_utm11_meters
FROM input;
```

Output

```
 dist_meters_spheroid | dist_meters_sphere | dist_utm11_meters
----------------------+--------------------+-------------------
             70454.92 |           70424.71 |          70438.00
```

Figure



See Also

ST_Distance, ST_DistanceSphere

7.12.10 ST_FrechetDistance

ST_FrechetDistance — Returns the Fréchet distance between two geometries.

Synopsis

float **ST_FrechetDistance**(geometry g1, geometry g2, float densifyFrac = -1);

Description

Implements algorithm for computing the Fréchet distance restricted to discrete points for both geometries, based on Computing Discrete Fréchet Distance. The Fréchet distance is a measure of similarity between curves that takes into account the location and ordering of the points along the curves. Therefore it is often better than the Hausdorff distance.

When the optional densifyFrac is specified, this function performs a segment densification before computing the discrete Fréchet distance. The densifyFrac parameter sets the fraction by which to densify each segment. Each segment will be split into a number of equal-length subsegments, whose fraction of the total length is closest to the given fraction.

Units are in the units of the spatial reference system of the geometries.



Note
The current implementation supports only vertices as the discrete locations. This could be extended to allow an arbitrary density of points to be used.



Note
The smaller densifyFrac we specify, the more accurate Fréchet distance we get. But, the computation time and the memory usage increase with the square of the number of subsegments.

Performed by the GEOS module.

Availability: 2.4.0 - requires GEOS >= 3.7.0

Examples

Both lines are traversed from left to right. The first line is straight, while the second bows upward through its middle vertex. The point ordering matters, so keeping both inputs in the same direction makes the comparison clearer.

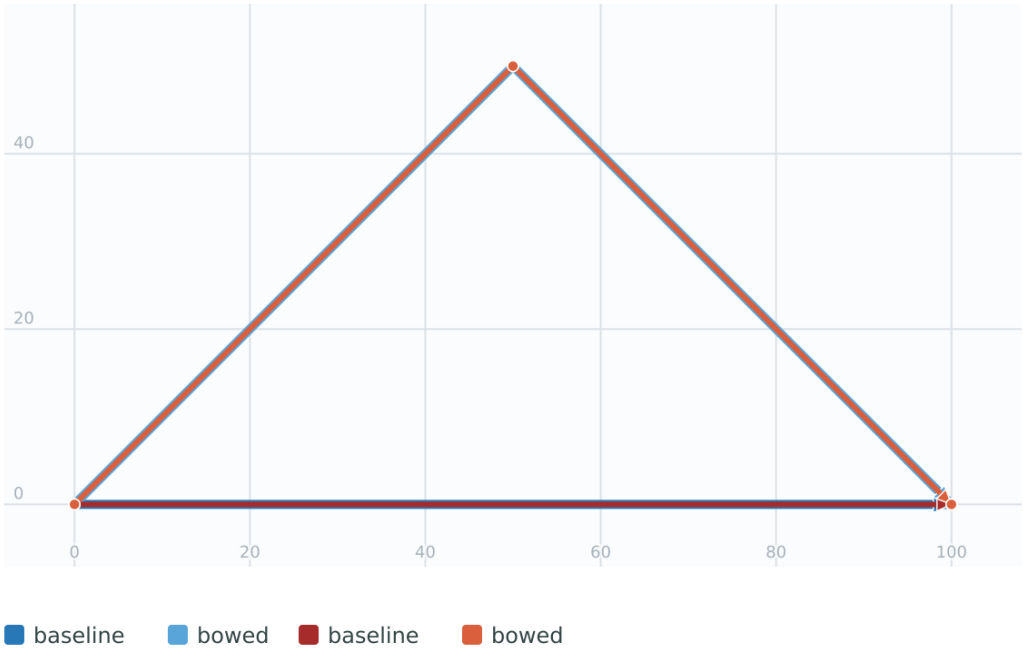
Code

```
WITH lines AS (
  SELECT 'LINESTRING (0 0,100 0)'::geometry AS baseline,
         'LINESTRING (0 0,50 50,100 0)'::geometry AS bowed
)
SELECT baseline AS baseline,
       bowed AS bowed,
       ST_FrechetDistance(baseline, bowed) AS discrete_distance,
       ST_FrechetDistance(baseline, bowed, 0.5) AS densified_distance
FROM lines;
```

Output

```
-[ RECORD 1 ]-----+---------------------------
baseline          | LINESTRING(0 0,100 0)
bowed             | LINESTRING(0 0,50 50,100 0)
discrete_distance | 70.7106781186548
densified_distance | 50
```

Figure



See Also

ST_HausdorffDistance

7.12.11 ST_HausdorffDistance

ST_HausdorffDistance — Returns the Hausdorff distance between two geometries.

Synopsis

float **ST_HausdorffDistance**(geometry g1, geometry g2);
float **ST_HausdorffDistance**(geometry g1, geometry g2, float densifyFrac);

Description

Returns the Hausdorff distance between two geometries. The Hausdorff distance is a measure of how similar or dissimilar 2 geometries are.

The function actually computes the "Discrete Hausdorff Distance". This is the Hausdorff distance computed at discrete points on the geometries. The *densifyFrac* parameter can be specified, to provide a more accurate answer by densifying segments before computing the discrete Hausdorff distance. Each segment is split into a number of equal-length subsegments whose fraction of the segment length is closest to the given fraction.

Units are in the units of the spatial reference system of the geometries.



Note
This algorithm is NOT equivalent to the standard Hausdorff distance. However, it computes an approximation that is correct for a large subset of useful cases. One important case is Linestrings that are roughly parallel to each other, and roughly equal in length. This is a useful metric for line matching.

Availability: 1.5.0

Examples

Hausdorff distance and minimum distance between two lines. The Hausdorff witness segment from the original illustration is retained and its length is checked against the computed distance.

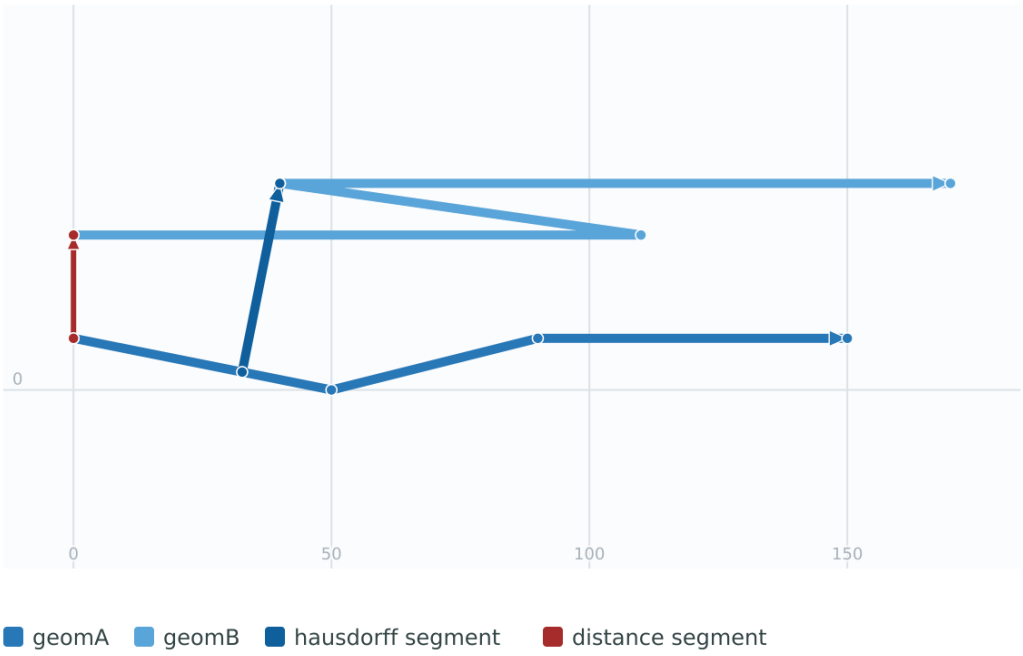
Code

```
WITH lines AS (
  SELECT 'LINESTRING (20 70,70 60,110 70,170 70)'::geometry AS geomA,
         'LINESTRING (20 90,130 90,60 100,190 100)'::geometry AS geomB,
         'LINESTRING (52.69230769230769 63.46153846153846,60 100)'::geometry AS ↩
    hausdorff_segment
)
SELECT round(ST_HausdorffDistance(geomA, geomB)::numeric, 6) AS hausdorff_distance,
       round(ST_Distance(geomA, geomB)::numeric, 6) AS distance,
       round(ST_Length(hausdorff_segment)::numeric, 6) AS witness_length,
       ST_ShortestLine(geomA, geomB) AS distance_segment
FROM lines;
```

Output

```
37.262066 | 20.000000 | 37.262066 | LINESTRING(20 70,20 90)
```

Figure



Example: Hausdorff distance with densification.

Code

```
SELECT ST_HausdorffDistance(
            'LINESTRING (130 0,0 0,0 150)'::geometry,
            'LINESTRING (10 10,10 150,130 10)'::geometry,
            0.5);
```

Figure



Example: For each building, find the parcel that best represents it. First we require that the parcel intersect with the building geometry. `DISTINCT ON` guarantees we get each building listed only once. `ORDER BY .. ST_HausdorffDistance` selects the parcel that is most similar to the building.

Code

```
SELECT DISTINCT ON (buildings.gid) buildings.gid, parcels.parcel_id
   FROM buildings
       INNER JOIN parcels
       ON ST_Intersects(buildings.geom, parcels.geom)
   ORDER BY buildings.gid, ST_HausdorffDistance(buildings.geom, parcels.geom);
```

See Also

ST_FrechetDistance

7.12.12 ST_Length

ST_Length — Returns the 2D length of a linear geometry.

Synopsis

float **ST_Length**(geometry a_2dlinestring);
float **ST_Length**(geography geog, boolean use_spheroid = true);

Description

For geometry types: returns the 2D Cartesian length of the geometry if it is a LineString, MultiLineString, ST_Curve, ST_MultiCurve. For areal geometries 0 is returned; use ST_Perimeter instead. The units of length is determined by the spatial reference system of the geometry.

For geography types: computation is performed using the inverse geodesic calculation. Units of length are in meters. The spheroid is specified by the SRID; when no SRID is provided, WGS84 is used. If `use_spheroid = false`, then the calculation is based on a sphere instead of a spheroid.

Currently for geometry this is an alias for ST_Length2D, but this may change to support higher dimensions.

Changed: 2.0.0 Breaking change -- in prior versions applying this to a MULTI/POLYGON of type geography would give you the perimeter of the POLYGON/MULTIPOLYGON. In 2.0.0 this was changed to return 0 to be in line with geometry behavior. Please use ST_Perimeter if you want the perimeter of a polygon

Note
For geography the calculation defaults to using a spheroidal model. To use the faster but less accurate spherical calculation use ST_Length(gg,false);



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.5.1



This method implements the SQL/MM specification. SQL-MM 3: 7.1.2, 9.3.4

Availability: 1.5.0 geography support was introduced in 1.5.

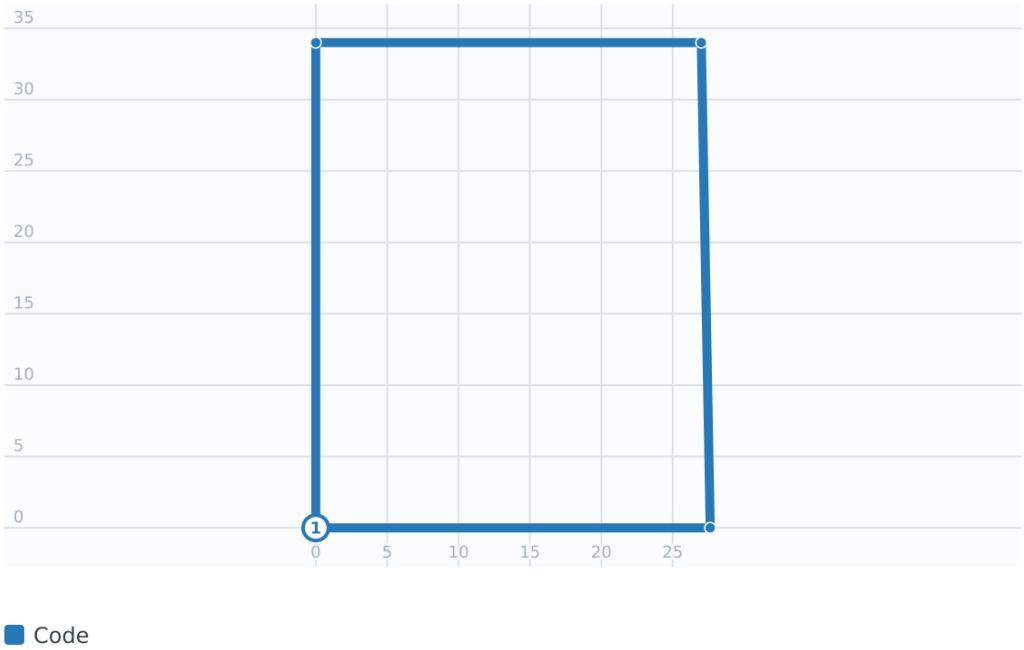
Geometry Examples

Return length in feet for line string. Note this is in feet because EPSG:2249 is Massachusetts State Plane Feet

Code

```
SELECT ST_Length(ST_GeomFromText(
'LINESTRING(743238 2967416,743238 2967450,743265 2967450,
743265.625 2967416,743238 2967416)',
2249));
```

Figure

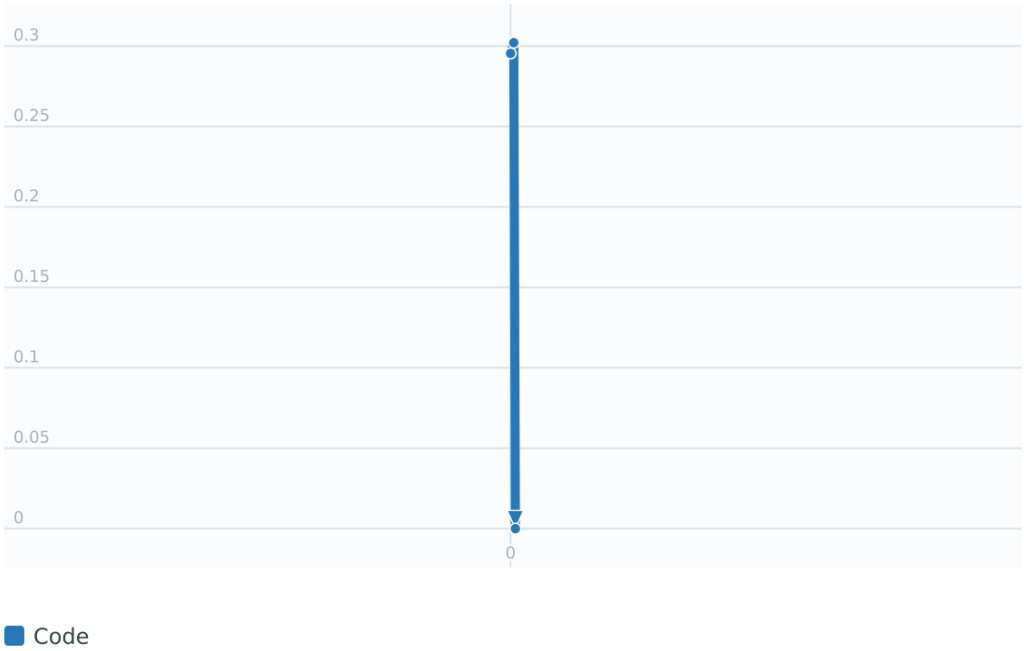


This example transforms a WGS 84 LineString to Massachusetts State Plane meters.

Code

```
SELECT ST_Length(ST_Transform(
        'SRID=4326;LINESTRING(-72.1260 42.45,-72.1240 42.45666,-72.123 42.1546)'::geometry,
            26986
    )
);
```

Figure



Geography Examples

Return length of WGS 84 geography line

This example uses the default spheroid calculation.

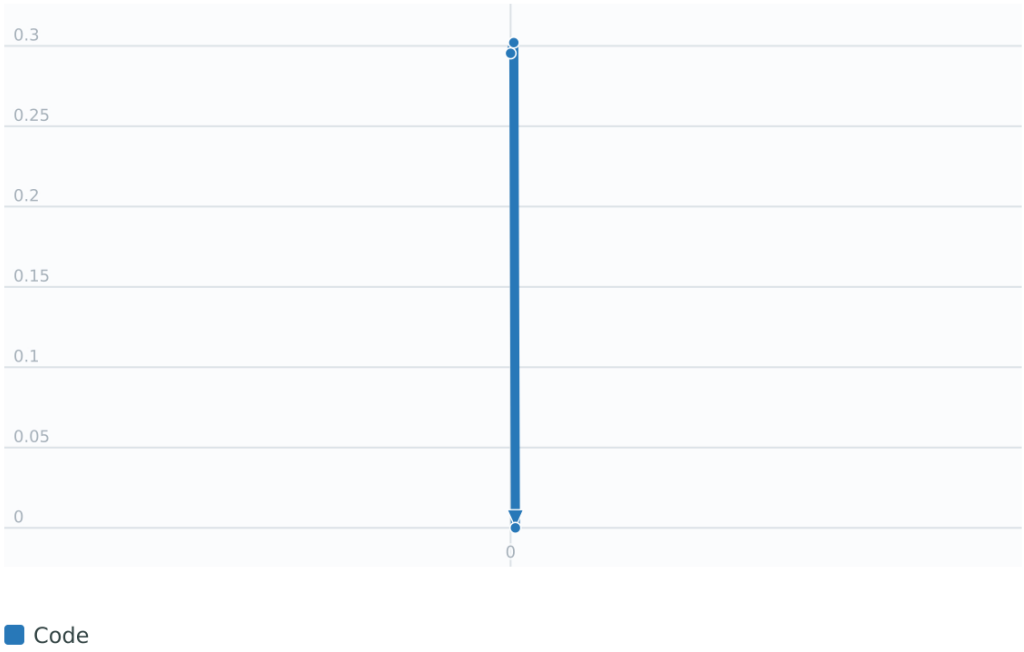
Code

```
SELECT ST_Length(the_geog) As length_spheroid, ST_Length(the_geog, false) As length_sphere
FROM (SELECT ST_GeographyFromText('SRID=4326;LINESTRING(-72.1260 42.45,-72.1240 ↩
    42.45666,-72.123 42.1546)') As the_geog)
 As foo;
```

Output

```
 length_spheroid  |  length_sphere
------------------+------------------
34310.5703627284 | 34346.20609607416
```

Figure



See Also

ST_GeographyFromText, ST_GeomFromEWKT, ST_LengthSpheroid, ST_Perimeter, ST_Transform

7.12.13 ST_Length2D

ST_Length2D — Returns the 2D length of a linear geometry. Alias for `ST_Length`

Synopsis

float **ST_Length2D**(geometry a_2dlinestring);

Description

Returns the 2D length of the geometry if it is a linestring or multi-linestring. This is an alias for `ST_Length`

See Also

ST_Length, ST_3DLength

7.12.14 ST_3DLength

ST_3DLength — Returns the 3D length of a linear geometry.

Synopsis

float **ST_3DLength**(geometry a_3dlinestring);

Description

Returns the 3-dimensional or 2-dimensional length of the geometry if it is a LineString or MultiLineString. For 2-d lines it will just return the 2-d length (same as ST_Length and ST_Length2D)

This function supports 3d and will not drop the z-index.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 7.1, 10.3

Changed: 2.0.0 In prior versions this used to be called ST_Length3D

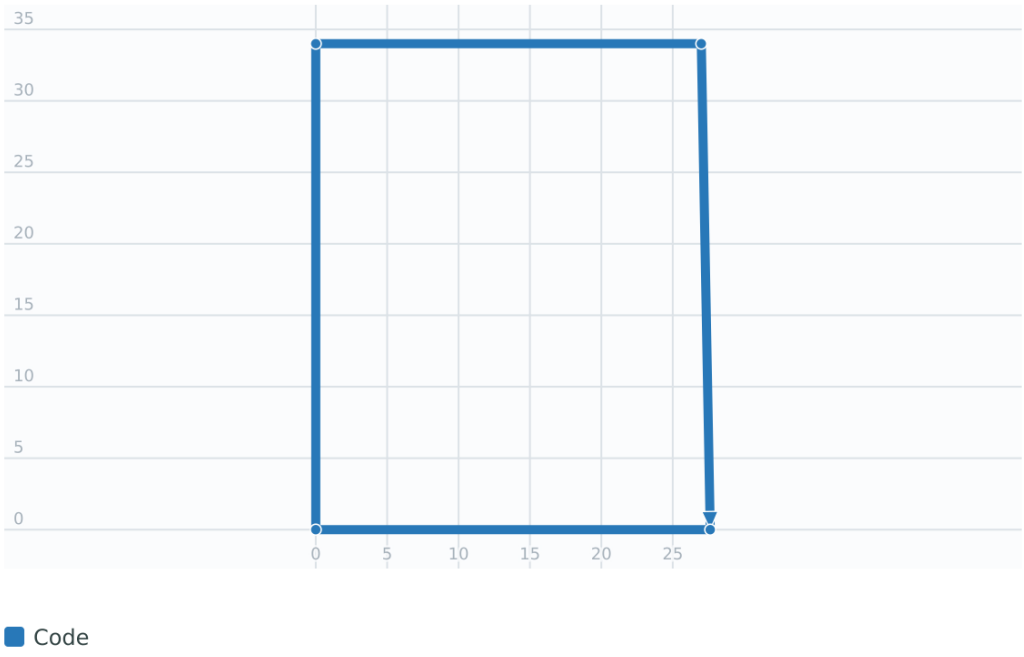
Examples

Return length in feet for a 3D cable. Note this is in feet because EPSG:2249 is Massachusetts State Plane Feet

Code

```
SELECT ST_3DLength(ST_GeomFromText(
'LINESTRING(743238 2967416 1,743238 2967450 1,743265 2967450 3,
743265.625 2967416 3,743238 2967416 3)',
2249));
```

Figure



See Also

ST_Length, ST_Length2D

7.12.15 ST_LengthSpheroid

ST_LengthSpheroid — Returns the 2D or 3D length/perimeter of a lon/lat geometry on a spheroid.

Synopsis

float **ST_LengthSpheroid**(geometry a_geometry, spheroid a_spheroid);

Description

Calculates the length or perimeter of a geometry on a spheroid. This is useful if the coordinates of the geometry are in longitude/latitude and a length is desired without reprojection. The spheroid is specified by a text value as follows:

```
SPHEROID[<NAME>,<SEMI-MAJOR AXIS>,<INVERSE FLATTENING>]
```

For example:

```
SPHEROID["GRS_1980",6378137,298.257222101]
```

Availability: 1.2.2

Changed: 2.2.0 In prior versions this was called ST_Length_Spheroid and had the alias ST_3DLength_Spheroid

 This function supports 3d and will not drop the z-index.

Examples

Measure a table geometry directly on a spheroid when the coordinates are stored in longitude and latitude.

Code

```
SELECT ST_LengthSpheroid(
    geometry_column,
    'SPHEROID["GRS_1980",6378137,298.257222101]'
    )
FROM geometry_table;
```

This example measures a 2D MultiLineString and each component separately on the same spheroid.

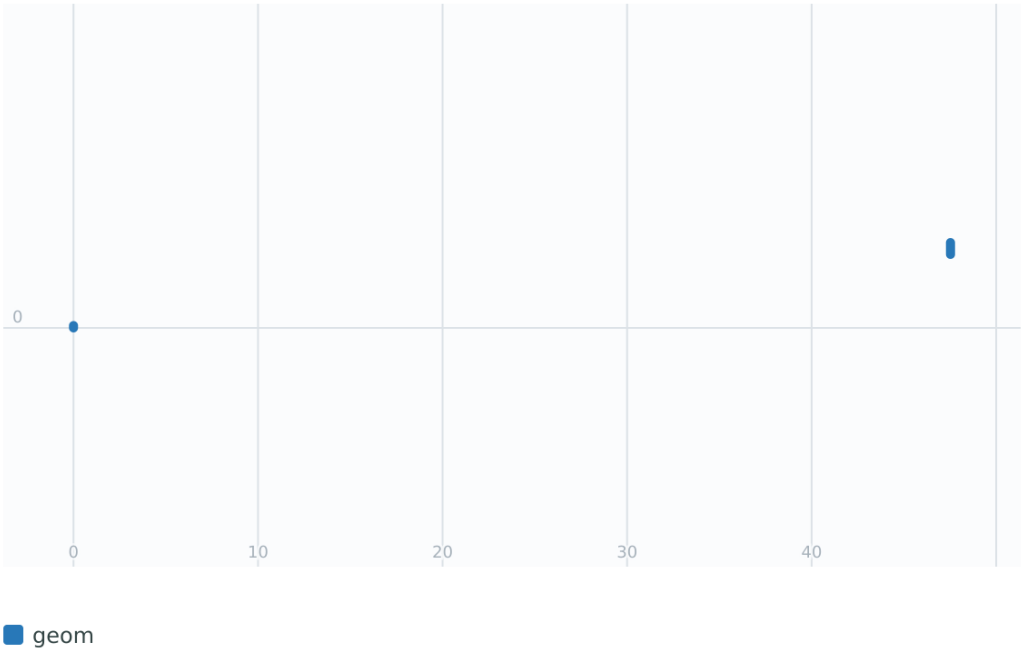
Code

```
WITH sample AS (
  SELECT 'MULTILINESTRING((-118.584 38.374,-118.583 38.5),
          (-71.05957 42.3589,-71.061 43))'::geometry AS geom,
         CAST('SPHEROID["GRS_1980",6378137,298.257222101]' AS spheroid) AS sph_m
)
SELECT ST_LengthSpheroid(geom, sph_m) AS tot_len,
       ST_LengthSpheroid(ST_GeometryN(geom, 1), sph_m) AS len_line1,
       ST_LengthSpheroid(ST_GeometryN(geom, 2), sph_m) AS len_line2
FROM sample;
```

Output

```
     tot_len      |    len_line1     |    len_line2
------------------+------------------+------------------
 85204.52077117894 | 13986.87252824332 | 71217.64824293563
```

Figure



This example uses 3D input; Z contributes to the measured result.

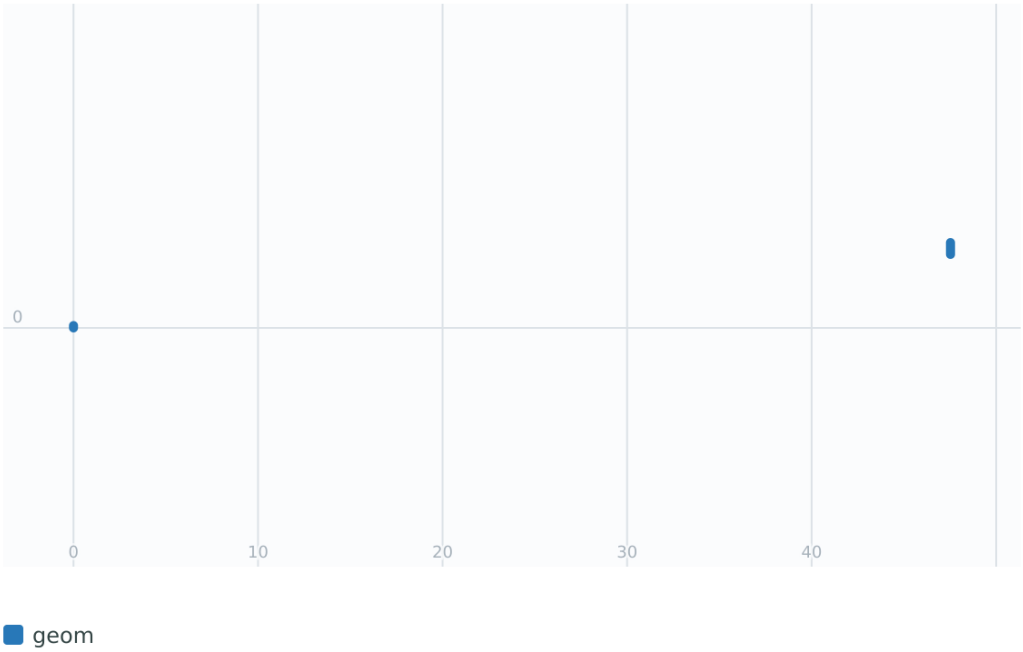
Code

```
WITH sample AS (
  SELECT 'MULTILINESTRING((-118.584 38.374 20,-118.583 38.5 30),
          (-71.05957 42.3589 75,-71.061 43 90))'::geometry AS geom,
         CAST('SPHEROID["GRS_1980",6378137,298.257222101]' AS spheroid) AS sph_m
)
SELECT ST_LengthSpheroid(geom, sph_m) AS tot_len,
       ST_LengthSpheroid(ST_GeometryN(geom, 1), sph_m) AS len_line1,
       ST_LengthSpheroid(ST_GeometryN(geom, 2), sph_m) AS len_line2
FROM sample;
```

Output

```
     tot_len      |    len_line1     |    len_line2
------------------+------------------+------------------
 85204.52592562366 | 13986.876103023424 | 71217.64982260024
```

Figure



See Also

ST_GeometryN, ST_Length

7.12.16 ST_LongestLine

ST_LongestLine — Returns the 2D longest line between two geometries.

Synopsis

geometry **ST_LongestLine**(geometry g);
geometry **ST_LongestLine**(geometry g1, geometry g2);

Description

Returns the 2-dimensional longest line between the points of two geometries. The line returned starts on `g1` and ends on `g2`.

The longest line always occurs between two vertices. The function returns the first longest line if more than one is found. The length of the line is equal to the distance returned by ST_MaxDistance.

If only one geometry is provided, or g1 and g2 are the same geometry, returns the line between the two vertices farthest apart in the geometry. The endpoints of the line lie on the circle computed by ST_MinimumBoundingCircle.

Availability: 1.5.0

Enhanced: 3.7.0 - support for a single geometry input.

Examples

Longest line between a Point and a LineString.

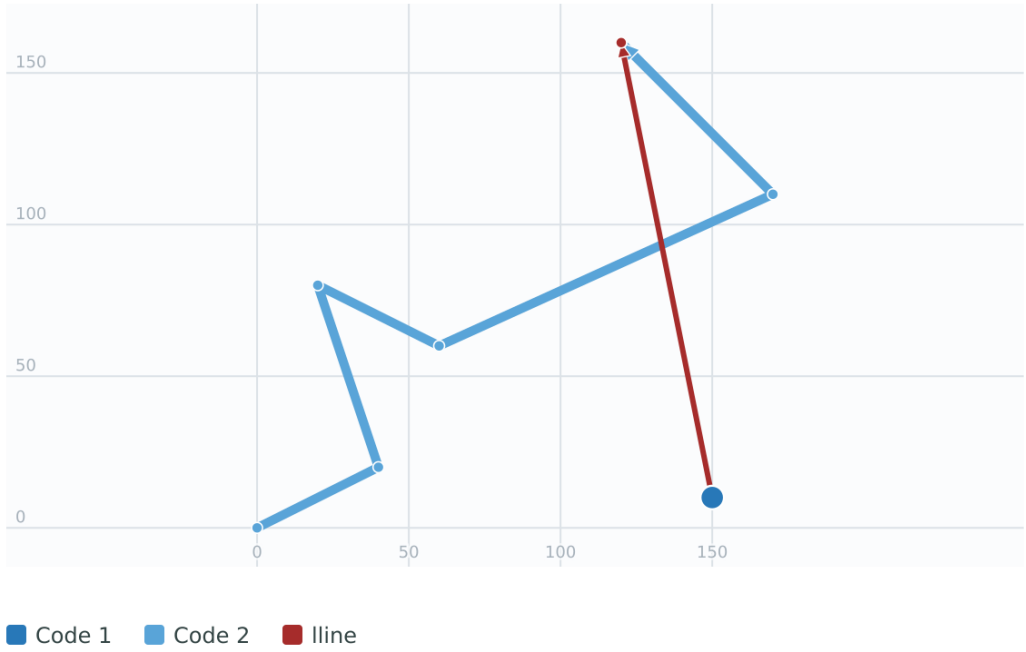
Code

```
SELECT ST_LongestLine(
        'POINT (160 40)',
        'LINESTRING (10 30,50 50,30 110,70 90,180 140,130 190)' ) AS lline;
```

Output

```
LINESTRING(160 40,130 190)
```

Figure



Longest line between two polygons.

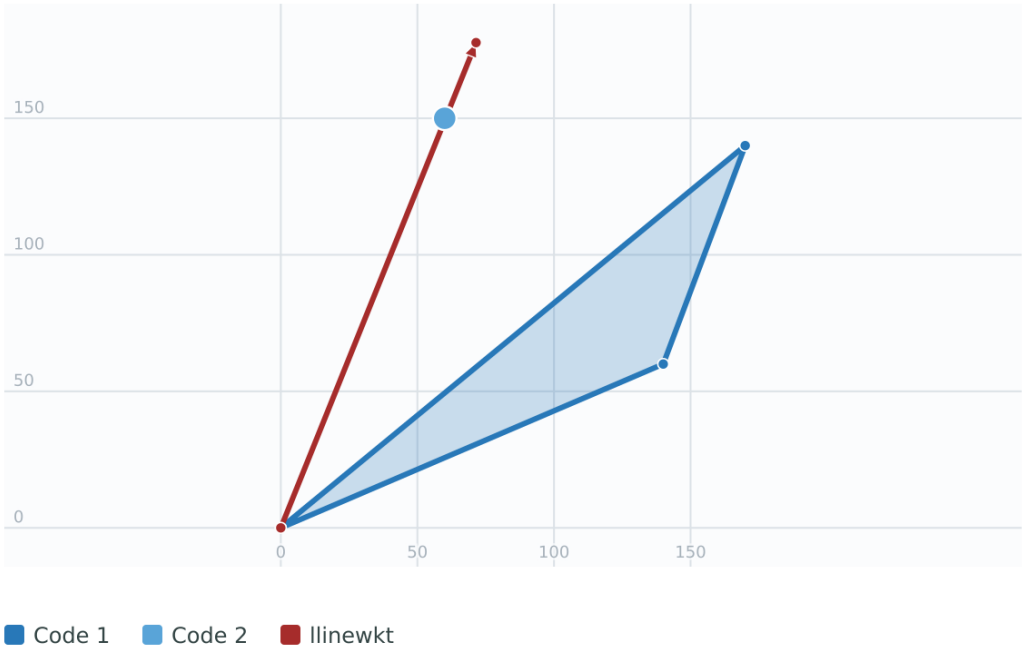
Code

```
SELECT ST_LongestLine(
        'POLYGON ((190 150,20 10,160 70,190 150))',
        ST_Buffer('POINT(80 160)', 30)
            ) AS llinewkt;
```

Output

```
LINESTRING(20 10,91.4805029709527 187.7163859753386)
```

Figure



Longest line across a single geometry. The length of the line is equal to the Maximum Distance. The endpoints of the line lie on the Minimum Bounding Circle.

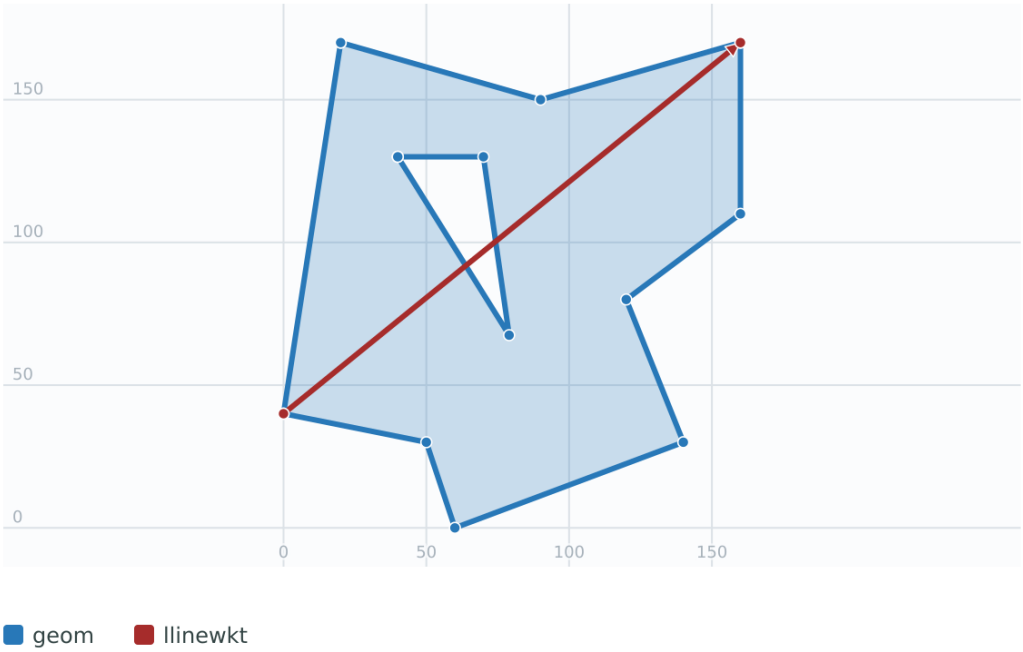
Code

```
SELECT ST_LongestLine(geom) AS llinewkt,
                  ST_MaxDistance(geom) AS max_dist,
                  ST_Length(ST_LongestLine(geom)) AS lenll
FROM (SELECT 'POLYGON ((40 180,110 160,180 180,180 120,140 90,160 40,80 10,70 40,20 50,40 ↩
    180),
              (60 140,99 77.5,90 140,60 140))'::geometry AS geom) AS t;
```

Output

```
         llinewkt          |      max_dist      |       lenll
---------------------------+--------------------+--------------------
 LINESTRING(20 50,180 180) | 206.15528128088303 | 206.15528128088303
```

Figure



See Also

ST_MaxDistance, ST_ShortestLine, ST_3DLongestLine, ST_MinimumBoundingCircle

7.12.17 ST_3DLongestLine

ST_3DLongestLine — Returns the 3D longest line between two geometries

Synopsis

geometry **ST_3DLongestLine**(geometry g1, geometry g2);

Description

Returns the 3-dimensional longest line between two geometries. The function returns the first longest line if more than one. The line returned starts in g1 and ends in g2. The 3D length of the line is equal to the distance returned by ST_3DMaxDistance.

Availability: 2.0.0

Changed: 2.2.0 - if 2 2D geometries are input, a 2D point is returned (instead of old behavior assuming 0 for missing Z). In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

Examples

Longest line between a LineString and a Point, comparing the 3D and 2D results.

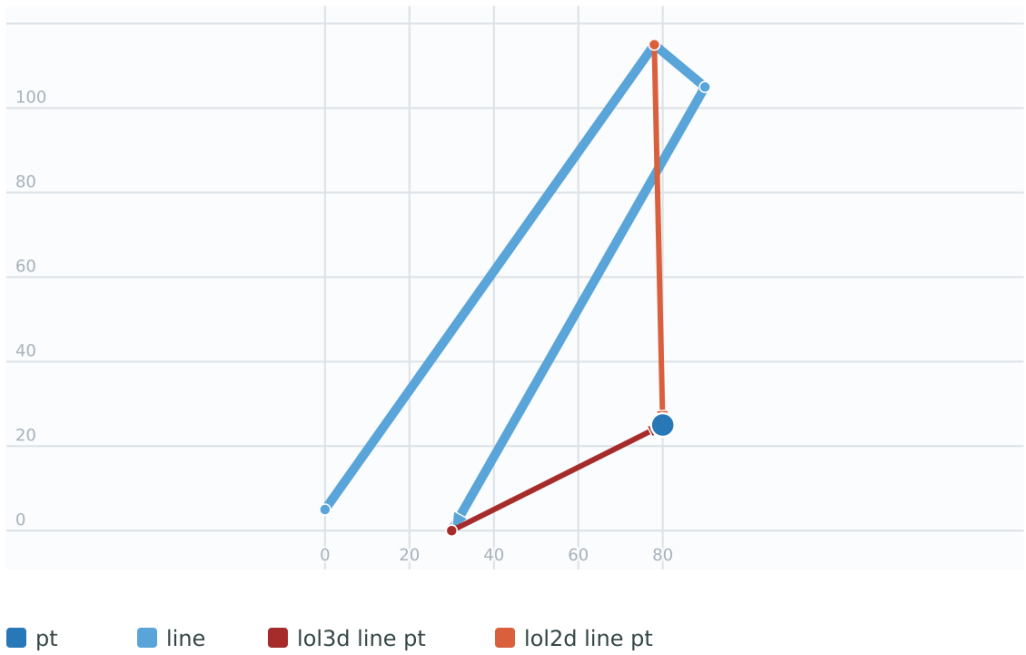
Code

```
SELECT ST_3DLongestLine(line, pt) AS lol3d_line_pt, ST_LongestLine(line, pt) AS  ↩
    lol2d_line_pt
FROM (
    SELECT 'POINT(100 100 30)'::geometry AS pt,
           'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 1000)'::geometry AS line
) AS foo;
```

Output

```
          lol3d_line_pt           |       lol2d_line_pt
----------------------------------+---------------------------
 LINESTRING(50 75 1000,100 100 30) | LINESTRING(98 190,100 100)
```

Figure



Longest line between a LineString and a MultiPoint, comparing the 3D and 2D results.

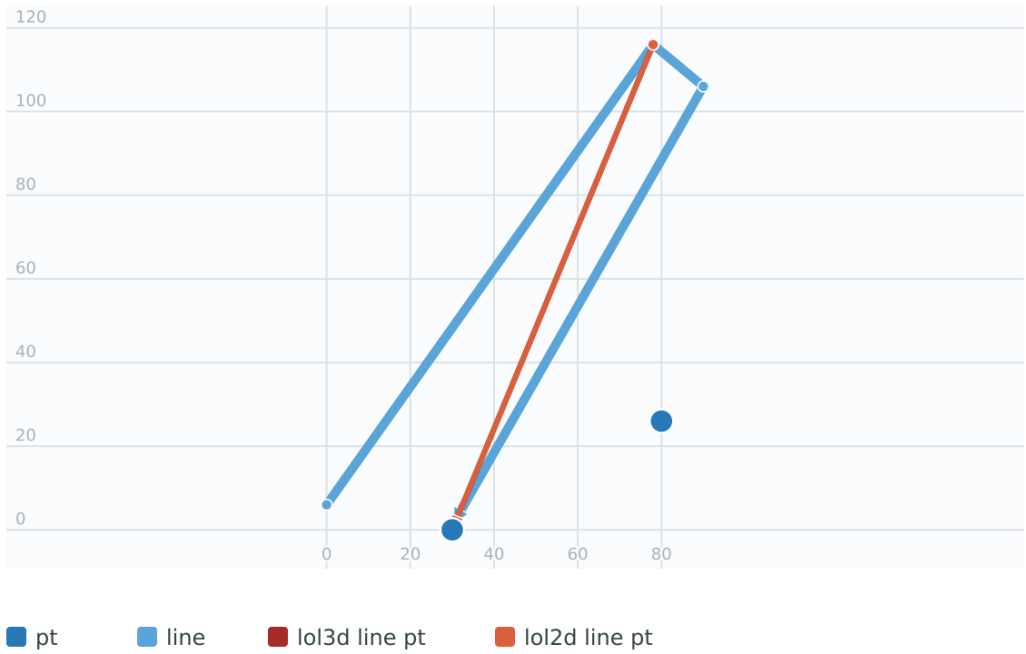
Code

```
SELECT ST_3DLongestLine(line, pt) AS lol3d_line_pt, ST_LongestLine(line, pt) AS  ↩
    lol2d_line_pt
FROM (
    SELECT 'MULTIPOINT(100 100 30,50 74 1000)'::geometry AS pt,
           'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 900)'::geometry AS line
) AS foo;
```

Output

```
          lol3d_line_pt          |      lol2d_line_pt
---------------------------------+-------------------------
 LINESTRING(98 190 1,50 74 1000) | LINESTRING(98 190,50 74)
```

Figure



Longest line between a Polygon and a MultiLineString, comparing the 3D and 2D results.

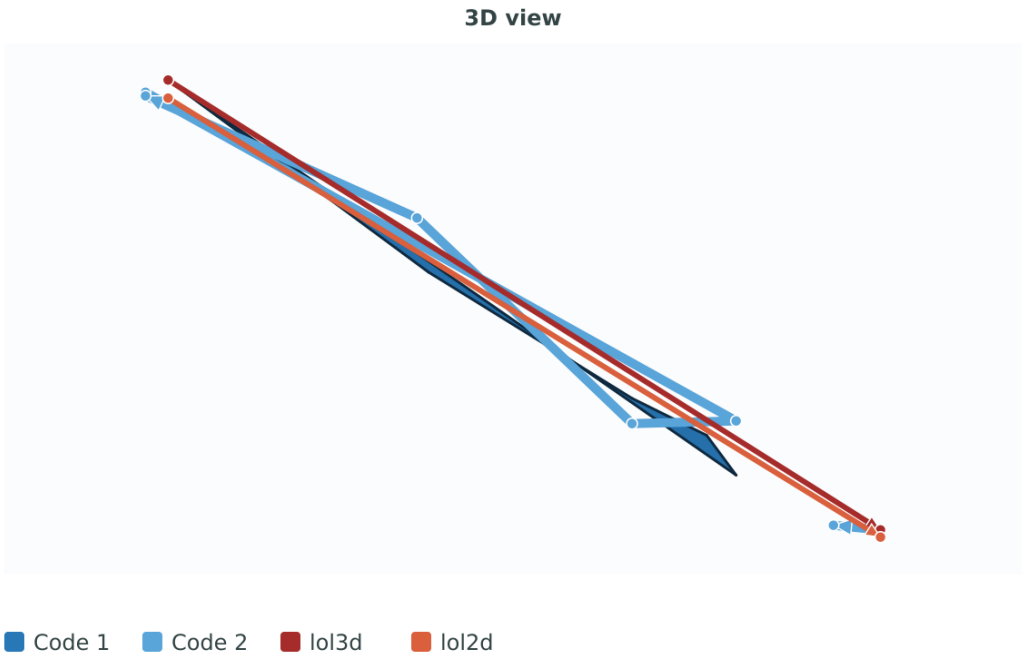
Code

```
SELECT ST_3DLongestLine(poly, mline) AS lol3d, ST_LongestLine(poly, mline) AS lol2d
FROM (
    SELECT ST_GeomFromEWKT(
            'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))'
        ) AS poly,
        ST_GeomFromEWKT(
            'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),
            (1 10 2,5 20 1))'
        ) AS mline
) AS foo;
```

Output

```
            lol3d             |          lol2d
------------------------------+-------------------------
 LINESTRING(175 150 5,1 10 2) | LINESTRING(175 150,1 10)
```

Figure



See Also

ST_3DClosestPoint, ST_3DDistance, ST_LongestLine, ST_3DShortestLine, ST_3DMaxDistance

7.12.18 ST_MaxDistance

ST_MaxDistance — Returns the 2D largest distance between two geometries in projected units.

Synopsis

float **ST_MaxDistance**(geometry g);
float **ST_MaxDistance**(geometry g1, geometry g2);

Description

Returns the 2-dimensional maximum distance between two geometries, in projected units. The maximum distance always occurs between two vertices. This is the length of the line returned by ST_LongestLine.

If only one geometry is provided, or g1 and g2 are the same geometry, returns the distance between the two vertices farthest apart in that geometry.

Availability: 1.5.0

Enhanced: 3.7.0 - support for a single geometry input.

Examples

Maximum distance between a point and lines.

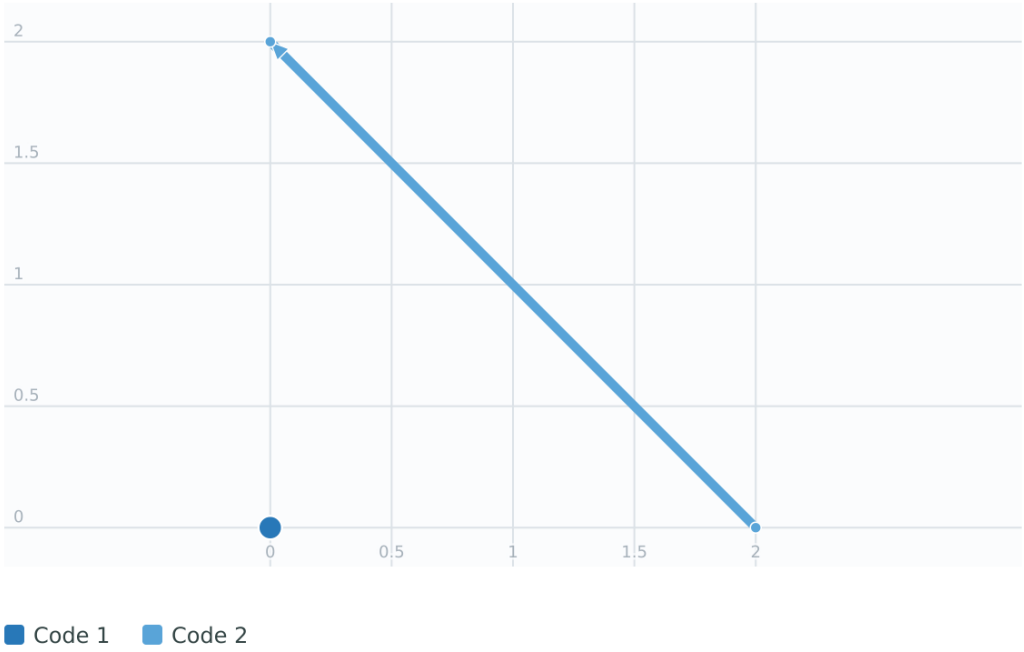
Code

```
SELECT ST_MaxDistance('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
2
```

Figure



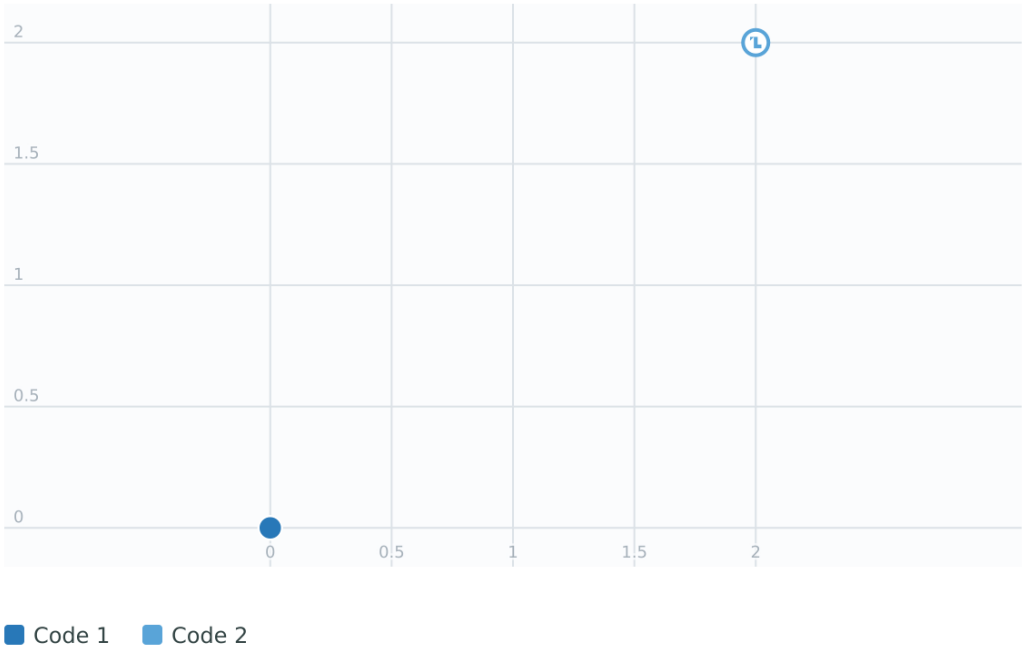
Code

```
SELECT ST_MaxDistance('POINT(0 0)'::geometry, 'LINESTRING ( 2 2,2 2 )'::geometry);
```

Output

```
2.82842712474619
```

Figure



Maximum distance between vertices of a single geometry.

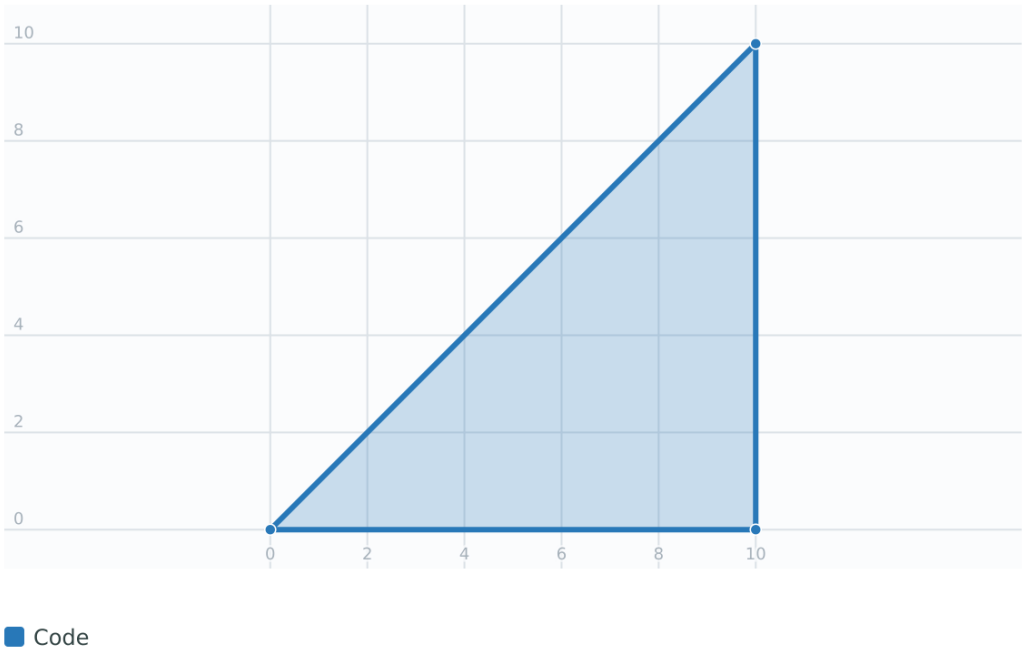
Code

```
SELECT ST_MaxDistance('POLYGON ((10 10,10 0,0 0,10 10))'::geometry);
```

Output

```
14.142135623730951
```

Figure



See Also

ST_Distance, ST_LongestLine, ST_DFullyWithin

7.12.19 ST_3DMaxDistance

ST_3DMaxDistance — Returns the 3D cartesian maximum distance (based on spatial ref) between two geometries in projected units.

Synopsis

float **ST_3DMaxDistance**(geometry g1, geometry g2);

Description

Returns the 3-dimensional maximum cartesian distance between two geometries in projected units (spatial ref units).

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

Availability: 2.0.0

Changed: 2.2.0 - In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z.

Examples

Geometry example - units in meters (SRID: 2163 US National Atlas Equal area) (3D point and line compared 2D point and line) Note that currently no vertical datum support so Z is not transformed and assumed to be same units as final.

Code

```
SELECT ST_3DMaxDistance(
            ST_Transform('SRID=4326;POINT(-72.1235 42.3521 10000)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)':: ↩
    geometry, 2163)
            ) As dist_3d,
            ST_MaxDistance(
                ST_Transform('SRID=4326;POINT(-72.1235 42.3521 10000)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)':: ↩
    geometry, 2163)
            ) As dist_2d;
```

Output

```
     dist_3d      |     dist_2d
------------------+-----------------
 24383.7467488441 | 22247.8472107251
```

See Also

ST_Distance, ST_3DDWithin, ST_3DMaxDistance, ST_Transform

7.12.20 ST_MinimumClearance

ST_MinimumClearance — Returns the minimum clearance of a geometry, a measure of a geometry's robustness.

Synopsis

float **ST_MinimumClearance**(geometry g);

Description

It is possible for a geometry to meet the criteria for validity according to ST_IsValid (polygons) or ST_IsSimple (lines), but to become invalid if one of its vertices is moved by a small distance. This can happen due to loss of precision during conversion to text formats (such as WKT, KML, GML, GeoJSON), or binary formats that do not use double-precision floating point coordinates (e.g. MapInfo TAB).

The minimum clearance is a quantitative measure of a geometry's robustness to change in coordinate precision. It is the largest distance by which vertices of the geometry can be moved without creating an invalid geometry. Larger values of minimum clearance indicate greater robustness.

If a geometry has a minimum clearance of `e`, then:

- No two distinct vertices in the geometry are closer than the distance `e`.
- No vertex is closer than `e` to a line segment of which it is not an endpoint.

If no minimum clearance exists for a geometry (e.g. a single point, or a MultiPoint whose points are identical), the return value is `Infinity`.

To avoid validity issues caused by precision loss, ST_ReducePrecision can reduce coordinate precision while ensuring that polygonal geometry remains valid.

Availability: 2.3.0

Examples

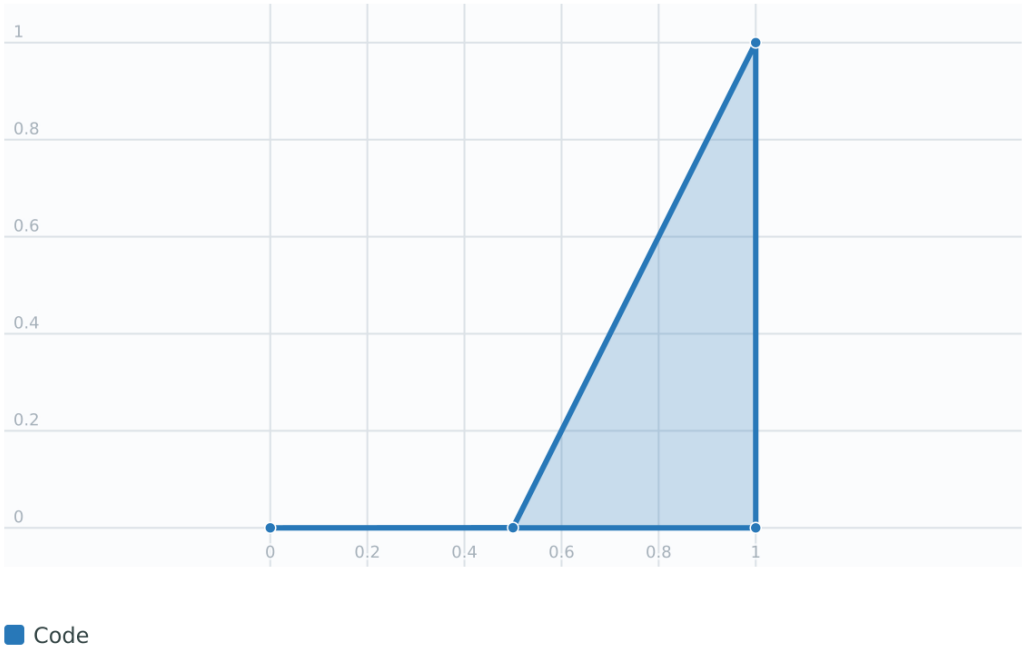
Code

```
SELECT ST_MinimumClearance('POLYGON ((0 0,1 0,1 1,0.5 3.2e-4,0 0))');
```

Output

```
0.00032
```

Figure



See Also

ST_MinimumClearanceLine, ST_IsSimple, ST_IsValid, ST_ReducePrecision

7.12.21 ST_MinimumClearanceLine

ST_MinimumClearanceLine — Returns the two-point LineString spanning a geometry's minimum clearance.

Synopsis

Geometry **ST_MinimumClearanceLine**(geometry g);

Description

Returns the two-point LineString spanning a geometry's minimum clearance. If the geometry does not have a minimum clearance, `LINESTRING EMPTY` is returned.

Performed by the GEOS module.

Availability: 2.3.0 - requires GEOS >= 3.6.0

Examples

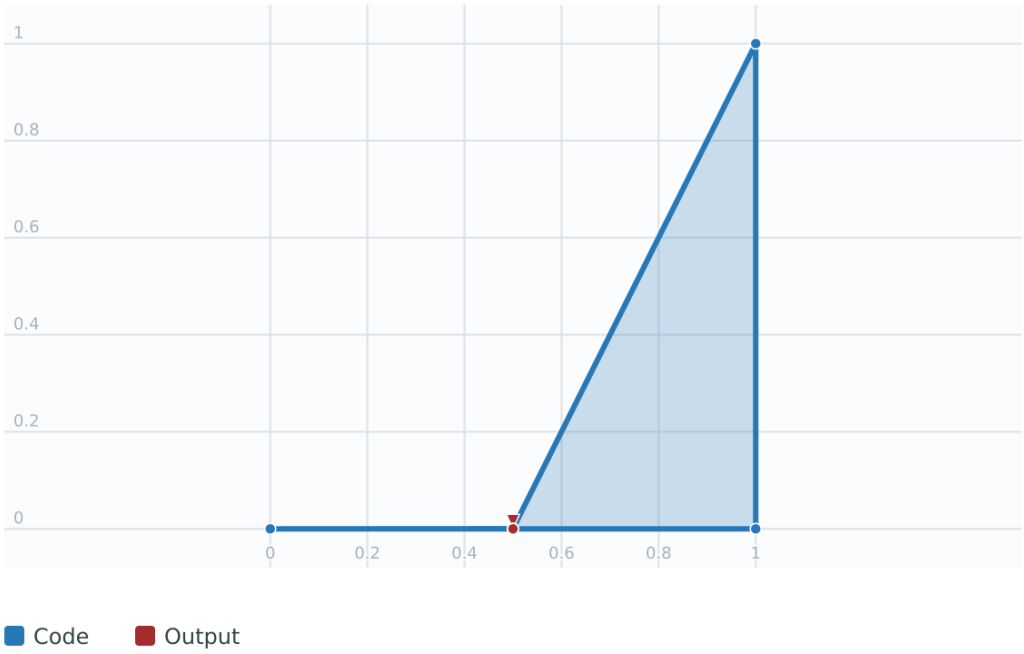
Code

```
SELECT ST_MinimumClearanceLine('POLYGON ((0 0,1 0,1 1,0.5 3.2e-4,0 0))');
```

Output

```
LINESTRING(0.5 0.00032,0.5 0)
```

Figure



See Also

ST_MinimumClearance

7.12.22 ST_Perimeter

ST_Perimeter — Returns the length of the boundary of a polygonal geometry or geography.

Synopsis

float **ST_Perimeter**(geometry g1);
float **ST_Perimeter**(geography geog, boolean use_spheroid = true);

Description

Returns the 2D perimeter of the geometry/geography if it is a ST_Surface, ST_MultiSurface (Polygon, MultiPolygon). 0 is returned for non-areal geometries. For linear geometries use ST_Length. For geometry types, units for perimeter measures are specified by the spatial reference system of the geometry.

For geography types, the calculations are performed using the inverse geodesic problem, where perimeter units are in meters. The spheroid is specified by the SRID; when no SRID is provided, WGS84 is used. If `use_spheroid = false`, then calculations will approximate a sphere instead of a spheroid.

Currently this is an alias for ST_Perimeter2D, but this may change to support higher dimensions.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 8.1.3, 9.5.4

Availability 2.0.0: Support for geography was introduced

Examples

Geometry.

Return perimeter in feet for Polygon and MultiPolygon. Note this is in feet because EPSG:2249 is Massachusetts State Plane Feet

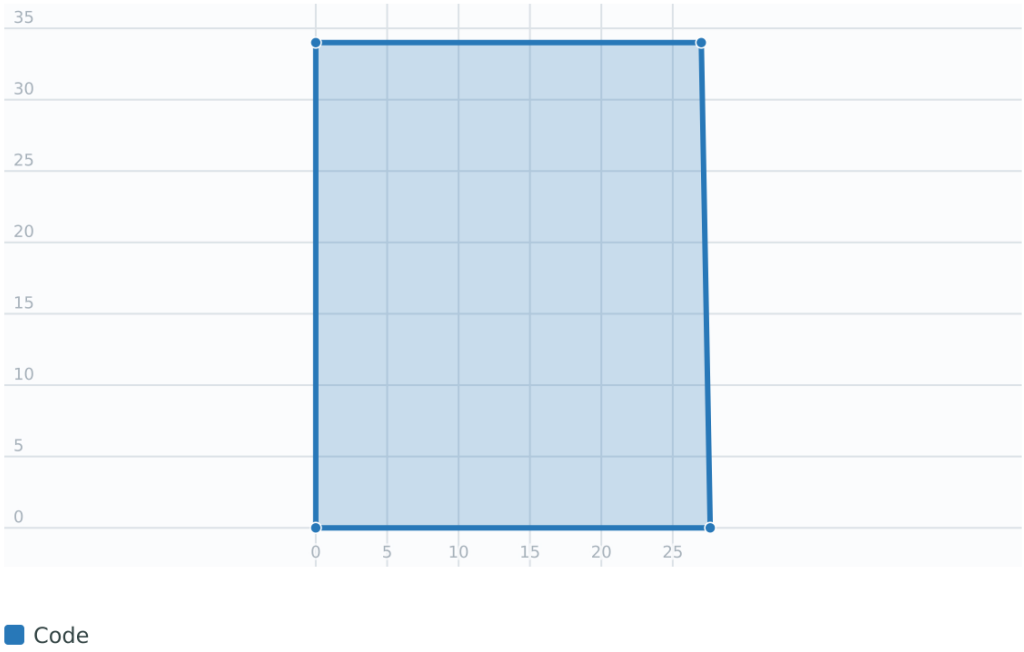
Code

```
SELECT ST_Perimeter(ST_GeomFromText(
'POLYGON((743238 2967416,743238 2967450,743265 2967450,
743265.625 2967416,743238 2967416))',
2249));
```

Output

```
122.63074400009504
```

Figure



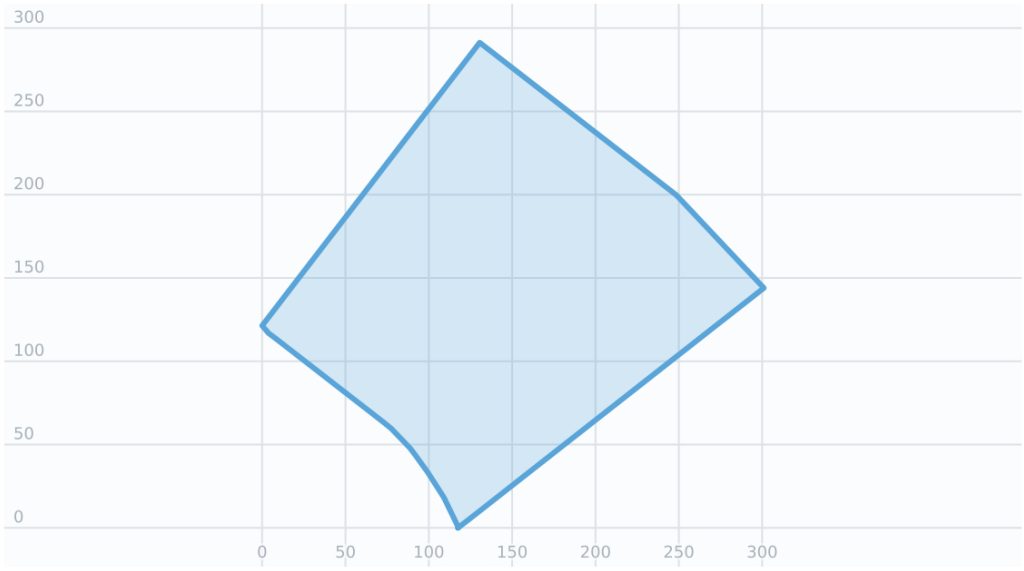
Code

```
SELECT ST_Perimeter(ST_GeomFromText(
'MULTIPOLYGON(((763104.471273676 2949418.44119003,
763104.477769673 2949418.42538203,
763104.189609677 2949418.22343004,763104.471273676 2949418.44119003)),
((763104.471273676 2949418.44119003,763095.804579742 2949436.33850239,
763086.132105649 2949451.46730207,763078.452329651 2949462.11549407,
763075.354136904 2949466.17407812,763064.362142565 2949477.64291974,
763059.953961626 2949481.28983009,762994.637609571 2949532.04103014,
762990.568508415 2949535.06640477,762986.710889563 2949539.61421415,
763117.237897679 2949709.50493431,763235.236617789 2949617.95619822,
763287.718121842 2949562.20592617,763111.553321674 2949423.91664605,
763104.471273676 2949418.44119003)))',
2249));
```

Output

```
845.227713366825
```

Figure



Return perimeter in meters and feet for Polygon and MultiPolygon. Note this is geography (WGS 84 long lat)

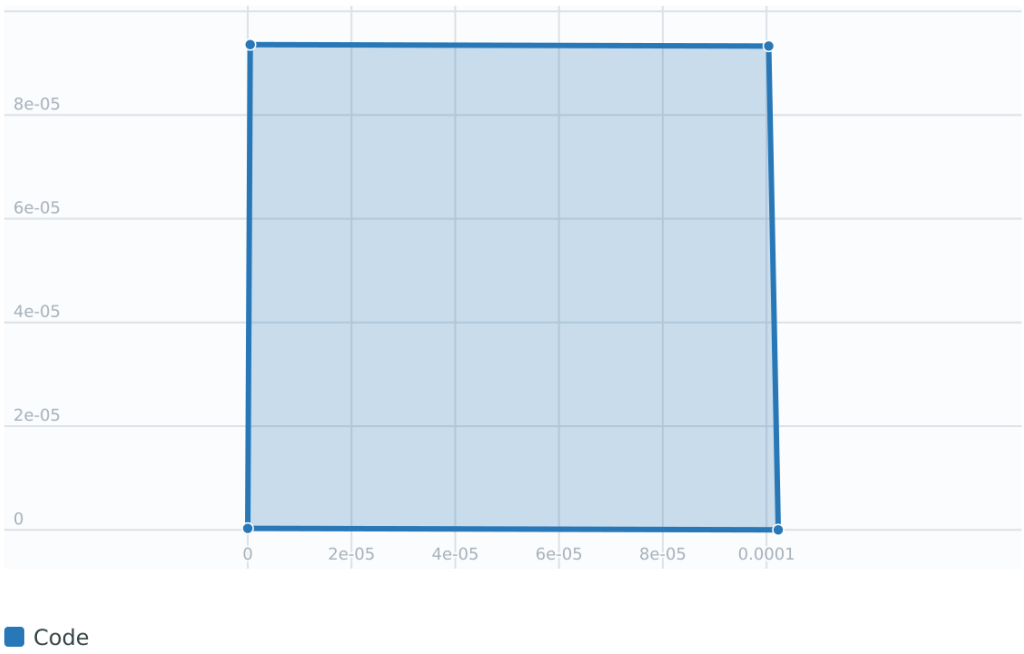
Code

```
SELECT round(ST_Perimeter(geog)::numeric, 8) As per_meters,
       round((ST_Perimeter(geog)/0.3048)::numeric, 8) As per_ft
FROM ST_GeogFromText('POLYGON((-71.1776848522251 42.3902896512902,-71.1776843766326 ↩
    42.3903829478009,
-71.1775844305465 42.3903826677917,-71.1775825927231 42.3902893647987,-71.1776848522251 ↩
    42.3902896512902))') As geog;
```

Output

```
 per_meters  |    per_ft
-------------+--------------
 37.37904625 | 122.63466619
```

Figure



This example uses a MultiPolygon.

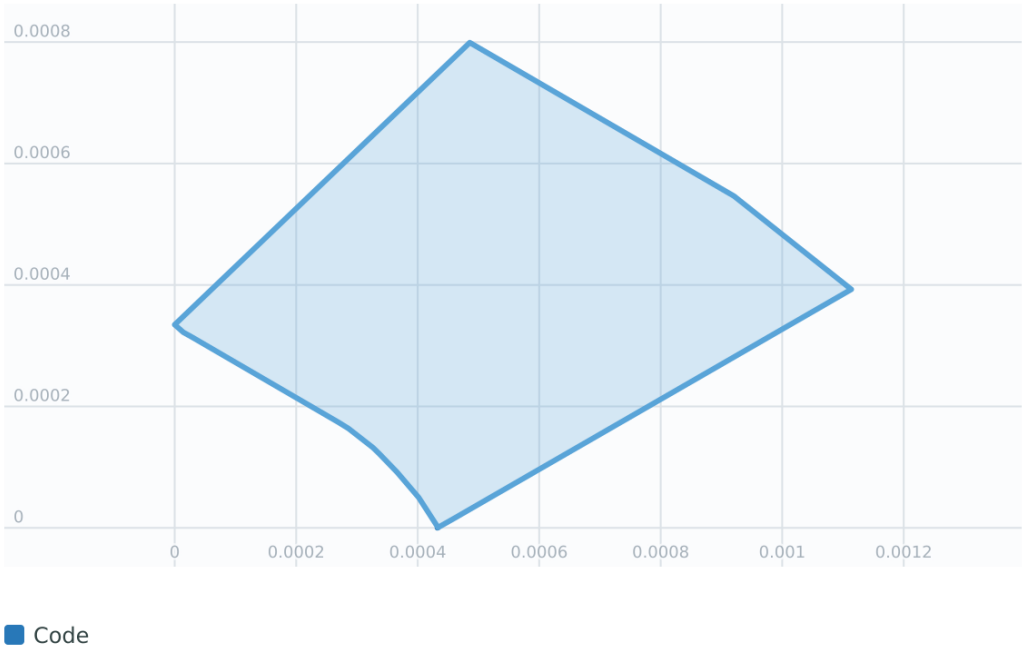
Code

```
SELECT round(ST_Perimeter(geog)::numeric, 8) As per_meters,
       round(ST_Perimeter(geog, false)::numeric, 8) As per_sphere_meters,
       round((ST_Perimeter(geog)/0.3048)::numeric, 8) As per_ft
FROM ST_GeogFromText('MULTIPOLYGON(((-71.1044543107478 42.340674480411,-71.1044542869917 ↩
    42.3406744369506,
-71.1044553562977 42.340673886454,-71.1044543107478 42.340674480411)),
((-71.1044543107478 42.340674480411,-71.1044860600303 42.3407237015564,-71.1045215770124 ↩
    42.3407653385914,
-71.1045498002983 42.3407946553165,-71.1045611902745 42.3408058316308,-71.1046016507427 ↩
    42.340837442371,
-71.104617893173 42.3408475056957,-71.1048586153981 42.3409875993595,-71.1048736143677 ↩
    42.3409959528211,
-71.1048878050242 42.3410084812078,-71.1044020965803 42.3414730072048,
-71.1039672113619 42.3412202916693,-71.1037740497748 42.3410666421308,
-71.1044280218456 42.3406894151355,-71.1044543107478 42.340674480411)))') As geog;
```

Output

```
  per_meters   | per_sphere_meters |    per_ft
---------------+-------------------+--------------
 257.63428368 |       257.41231145 | 845.25683623
```

Figure



See Also

ST_GeogFromText, ST_GeomFromText, ST_Length

7.12.23 ST_Perimeter2D

ST_Perimeter2D — Returns the 2D perimeter of a polygonal geometry. Alias for `ST_Perimeter`.

Synopsis

float **ST_Perimeter2D**(geometry geomA);

Description

Returns the 2-dimensional perimeter of a polygonal geometry.



Note
This is currently an alias for ST_Perimeter. In future versions ST_Perimeter may return the highest dimension perimeter for a geometry. This is still under consideration

See Also

ST_Perimeter

7.12.24 ST_3DPerimeter

ST_3DPerimeter — Returns the 3D perimeter of a polygonal geometry.

Synopsis

float **ST_3DPerimeter**(geometry geomA);

Description

Returns the 3-dimensional perimeter of the geometry, if it is a polygon or multi-polygon. If the geometry is 2-dimensional, then the 2-dimensional perimeter is returned.

This function supports 3d and will not drop the z-index.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3: 8.1, 10.5

Changed: 2.0.0 In prior versions this used to be called ST_Perimeter3D

Examples

Perimeter of a slightly elevated polygon in the air in Massachusetts state plane feet

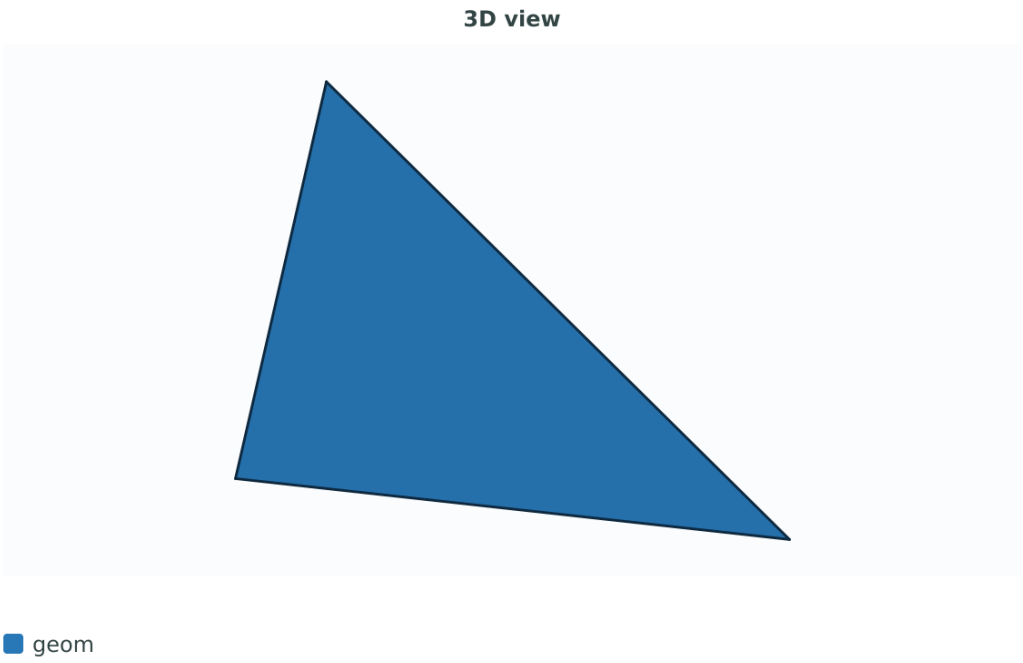
Code

```
SELECT ST_3DPerimeter(geom), ST_Perimeter2D(geom), ST_Perimeter(geom) FROM
            (SELECT 'SRID=2249;POLYGON((743238 2967416 2,743238 2967450 1,
743265.625 2967416 1,743238 2967416 2))'::geometry As geom) As foo;
```

Output

```
  ST_3DPerimeter   |  st_perimeter2d  |   st_perimeter
-------------------+------------------+------------------
105.46579359767418 | 105.43299727218763 | 105.43299727218763
```

Figure



See Also

ST_GeomFromEWKT, ST_Perimeter, ST_Perimeter2D

7.12.25 ST_ShortestLine

ST_ShortestLine — Returns the 2D shortest line between two geometries

Synopsis

geometry **ST_ShortestLine**(geometry geom1, geometry geom2);
geography **ST_ShortestLine**(geography geom1, geography geom2, boolean use_spheroid = true);

Description

Returns the 2-dimensional shortest line between two geometries. The line returned starts in `geom1` and ends in `geom2`. If `geom1` and `geom2` intersect the result is a line with start and end at an intersection point. The length of the line is the same as ST_Distance returns for g1 and g2.

Enhanced: 3.4.0 - support for geography.

Availability: 1.5.0

Examples

Shortest line between a Point and a LineString.

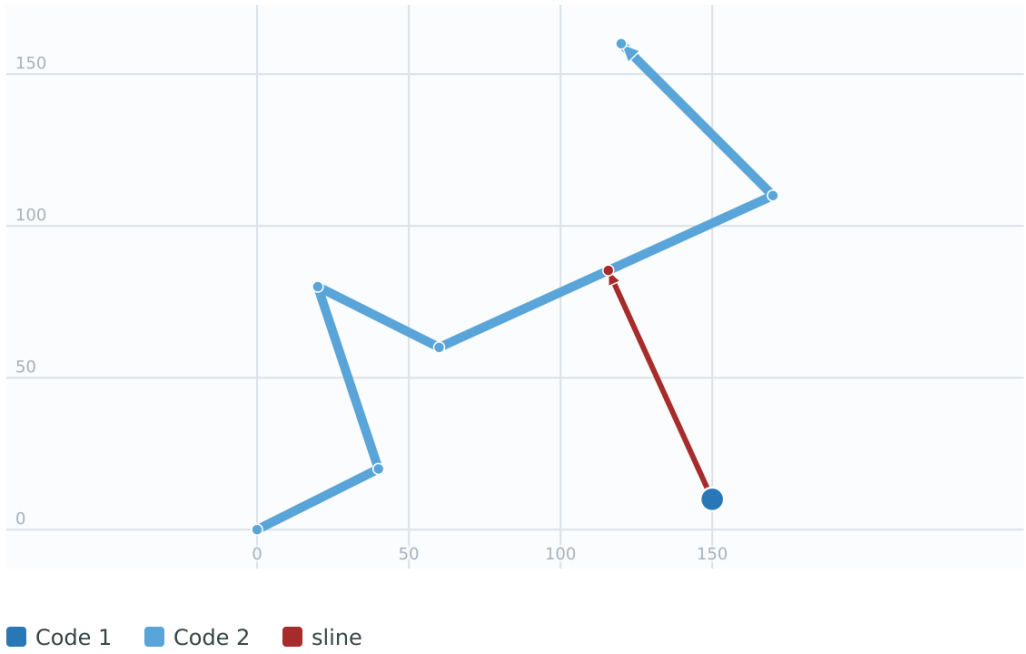
Code

```
SELECT ST_ShortestLine(
        'POINT (160 40)',
        'LINESTRING (10 30,50 50,30 110,70 90,180 140,130 190)') As sline;
```

Output

```
LINESTRING(160 40,125.75342465753425 115.34246575342466)
```

Figure



Shortest line between polygons.

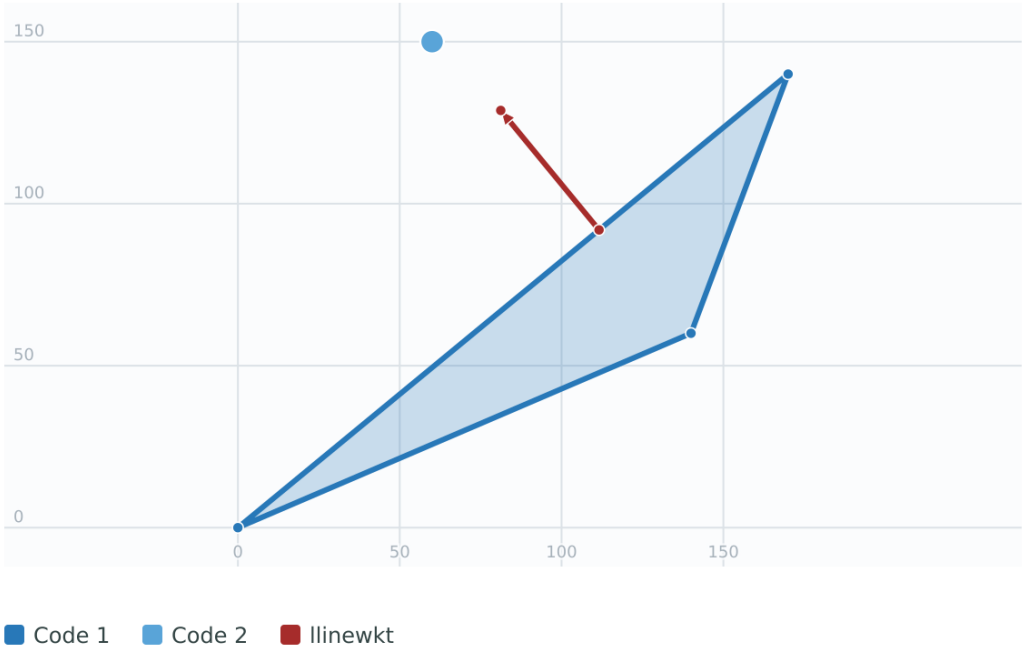
Code

```
SELECT ST_ShortestLine(
         'POLYGON ((190 150,20 10,160 70,190 150))',
         ST_Buffer('POINT(80 160)', 30)
                 ) AS llinewkt;
```

Output

```
LINESTRING(131.59149149528952 101.89887534906195,101.21320343559643 138.78679656440357)
```

Figure



See Also

ST_ClosestPoint, ST_Distance, ST_LongestLine, ST_MaxDistance

7.12.26 ST_3DShortestLine

ST_3DShortestLine — Returns the 3D shortest line between two geometries

Synopsis

geometry **ST_3DShortestLine**(geometry g1, geometry g2);

Description

Returns the 3-dimensional shortest line between two geometries. The function will only return the first shortest line if more than one, that the function finds. If g1 and g2 intersects in just one point the function will return a line with both start and end in that intersection-point. If g1 and g2 are intersecting with more than one point the function will return a line with start and end in the same point but it can be any of the intersecting points. The line returned will always start in g1 and end in g2. The 3D length of the line this function returns will always be the same as ST_3DDistance returns for g1 and g2.

Availability: 2.0.0

Changed: 2.2.0 - if 2 2D geometries are input, a 2D point is returned (instead of old behavior assuming 0 for missing Z). In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

Examples

Compare the 3D shortest line to the 2D shortest line for a LineString and a Point.

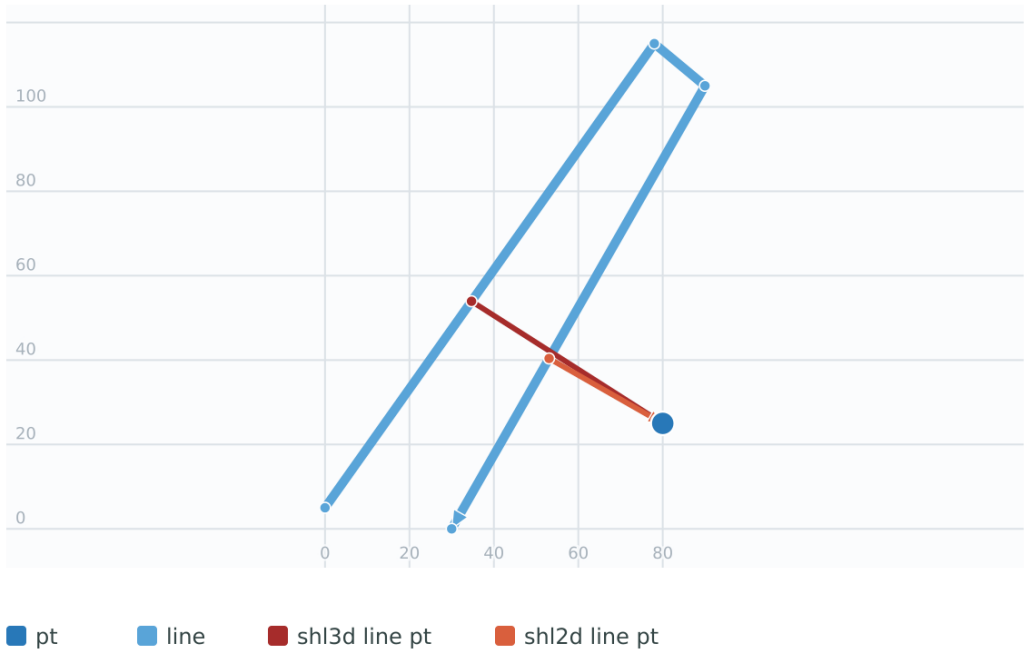
Code

```
WITH input AS (
  SELECT 'POINT(100 100 30)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 1000)'::geometry AS line
)
SELECT ST_3DShortestLine(line, pt) AS shl3d_line_pt,
       ST_ShortestLine(line, pt) AS shl2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------
shl3d_line_pt  | LINESTRING(54.69937988676193 128.93502291722837 11.547586950660556,100 100 30)
shl2d_line_pt  | LINESTRING(73.07692307692307 115.38461538461539,100 100)
```

Figure



Compare the 3D shortest line to the 2D shortest line for a LineString and a MultiPoint.

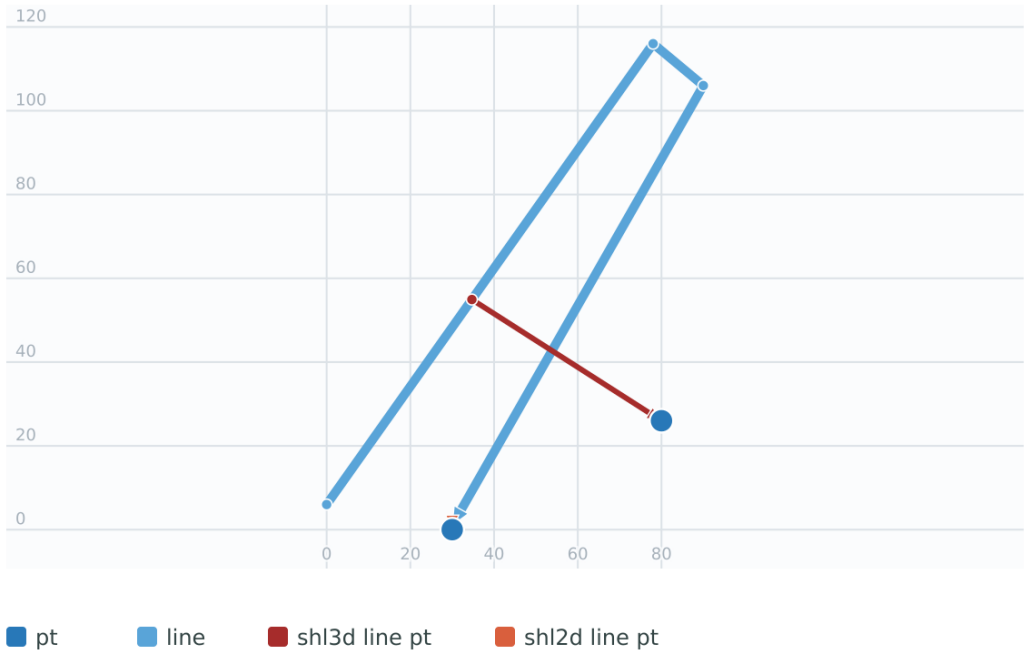
Code

```
WITH input AS (
  SELECT 'MULTIPOINT(100 100 30,50 74 1000)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 900)'::geometry AS line
)
SELECT ST_3DShortestLine(line, pt) AS shl3d_line_pt,
       ST_ShortestLine(line, pt) AS shl2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------
shl3d_line_pt  | LINESTRING(54.69937988676193 128.93502291722837 11.547586950660556,100 100 30)
shl2d_line_pt  | LINESTRING(50 75,50 74)
```

Figure



Compare the 3D shortest line to the 2D shortest line for a Polygon and a MultiLineString.

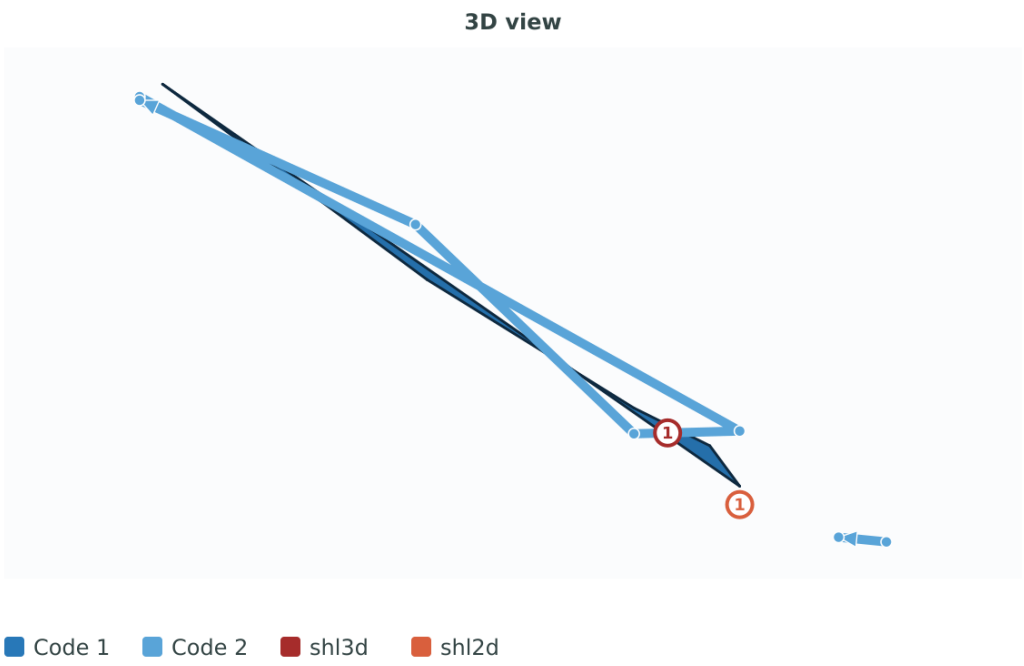
Code

```
WITH input AS (
  SELECT ST_GeomFromEWKT(
           'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))'
         ) AS poly,
         ST_GeomFromEWKT(
           'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),
                            (1 10 2,5 20 1))'
         ) AS mline
)
SELECT ST_3DShortestLine(poly, mline) AS shl3d,
       ST_ShortestLine(poly, mline) AS shl2d
FROM input;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------
shl3d          | LINESTRING(40.45454545454545 53.63636363636364 5,40.45454545454545 53.63636363636364 5)
shl2d          | LINESTRING(20 40,20 40)
```

Figure



See Also

ST_3DClosestPoint, ST_3DDistance, ST_LongestLine, ST_ShortestLine, ST_3DMaxDistance

7.13 Overlay Functions

7.13.1 ST_ClipByBox2D

ST_ClipByBox2D — Computes the portion of a geometry falling within a rectangle.

Synopsis

geometry **ST_ClipByBox2D**(geometry geom, box2d box);

Description

Clips a geometry by a 2D box in a fast and tolerant but possibly invalid way. Topologically invalid input geometries do not result in exceptions being thrown. The output geometry is not guaranteed to be valid (in particular, self-intersections for a polygon may be introduced).

Performed by the GEOS module.

Availability: 2.2.0

Examples

This example relies on the implicit cast from geometry to box2d for the second parameter.

Code

```
SELECT ST_ClipByBox2D(geom, ST_MakeEnvelope(0, 0, 10, 10)) FROM mytab;
```

See Also

ST_Intersection, ST_MakeBox2D, ST_MakeEnvelope

7.13.2 ST_Difference

ST_Difference — Computes a geometry representing the part of geometry A that does not intersect geometry B.

Synopsis

geometry **ST_Difference**(geometry geomA, geometry geomB, float8 gridSize = -1);

Description

Returns a geometry representing the part of geometry A that does not intersect geometry B. This is equivalent to `A - ST_Intersecti`
If A is completely contained in B then an empty atomic geometry of appropriate type is returned.



Note
This is the only overlay function where input order matters. ST_Difference(A, B) always returns a portion of A.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

Performed by the GEOS module

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

This method implements the SQL/MM specification. SQL-MM 3: 5.1.20

This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Examples

The difference of 2D linestrings.

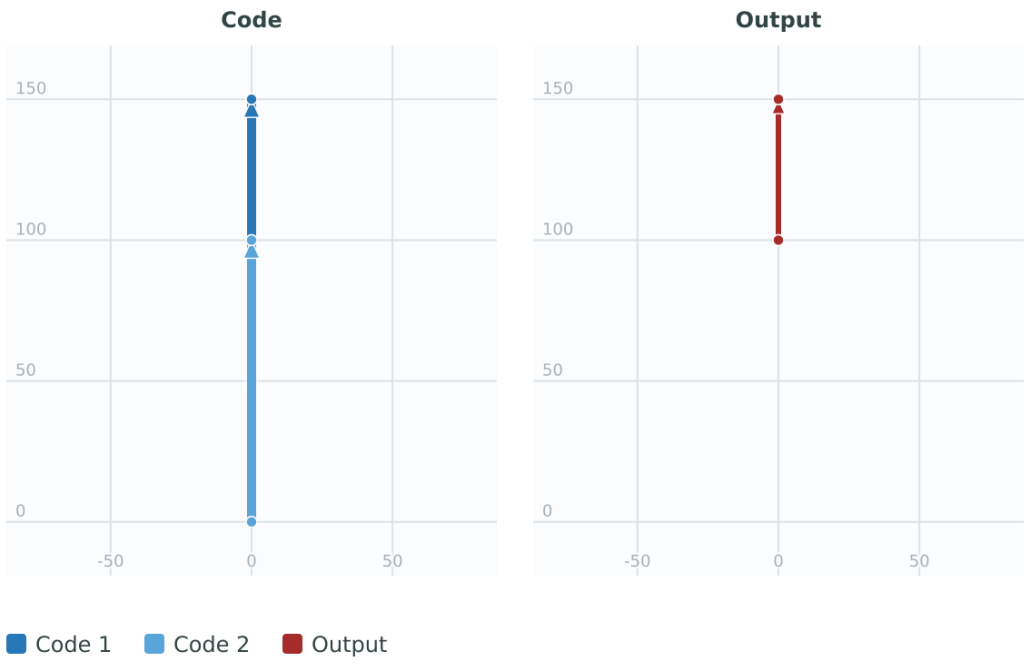
Code

```
SELECT ST_Difference('LINESTRING(50 100,50 200)'::geometry,
        'LINESTRING(50 50,50 150)'::geometry
    );
```

Output

```
LINESTRING(50 150,50 200)
```

Figure



The difference of 3D points.

Code

```
SELECT ST_Difference('MULTIPOINT(-118.58 38.38 5,-118.60 38.329 6,-118.614 38.281 7)' ::  ↩
    geometry,
 'POINT(-118.614 38.281 5)' :: geometry
);
```

Output

```
MULTIPOINT(-118.6 38.329 6,-118.58 38.38 5)
```

Figure



See Also

ST_SymDifference, ST_Intersection, ST_Union, ST_ReducePrecision

7.13.3 ST_Intersection

ST_Intersection — Computes a geometry representing the shared portion of geometries A and B.

Synopsis

geometry **ST_Intersection**(geometry geomA , geometry geomB , float8 gridSize = -1);
geography **ST_Intersection**(geography geogA , geography geogB);

Description

Returns a geometry representing the point-set intersection of two geometries. In other words, that portion of geometry A and geometry B that is shared between the two geometries.

If the geometries have no points in common (i.e. are disjoint) then an empty atomic geometry of appropriate type is returned.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

ST_Intersection in conjunction with ST_Intersects is useful for clipping geometries such as in bounding box, buffer, or region queries where you only require the portion of a geometry that is inside a country or region of interest.

Note

Spatial predicates such as ST_Intersects and overlay functions such as `ST_Intersection` are computed by different algorithms. For very close or nearly coincident coordinates, floating-point precision and geometric robustness may make a predicate return true while the computed intersection is empty, or place an intersection point slightly away from the input line. For repeatable overlay results, snap inputs to a suitable grid first, for example with ST_ReducePrecision, or use the `gridSize` parameter when all input vertices already lie on the grid.

Note

For geography this is a thin wrapper around the geometry implementation. It first determines the best SRID that fits the bounding box of the 2 geography objects (if geography objects are within one half zone UTM but not same UTM will pick one of those) (favoring UTM or Lambert Azimuthal Equal Area (LAEA) north/south pole, and falling back on mercator in worst case scenario) and then intersection in that best fit planar spatial ref and retransforms back to WGS84 geography.

Warning

This function will drop the M coordinate values if present.

Warning

If working with 3D geometries, you may want to use SFCGAL based CG_3DIntersection which does a proper 3D intersection for 3D geometries. Although this function works with Z-coordinate, it does an averaging of Z-Coordinate.

Performed by the GEOS module

Enhanced: 3.1.0 accept a gridSize parameter

Requires GEOS >= 3.9.0 to use the gridSize parameter

Changed: 3.0.0 does not depend on SFCGAL.

Availability: 1.5 support for geography data type was introduced.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

This method implements the SQL/MM specification. SQL-MM 3: 5.1.18

This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Examples

Code

```
SELECT ST_Intersection('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
POINT EMPTY
```

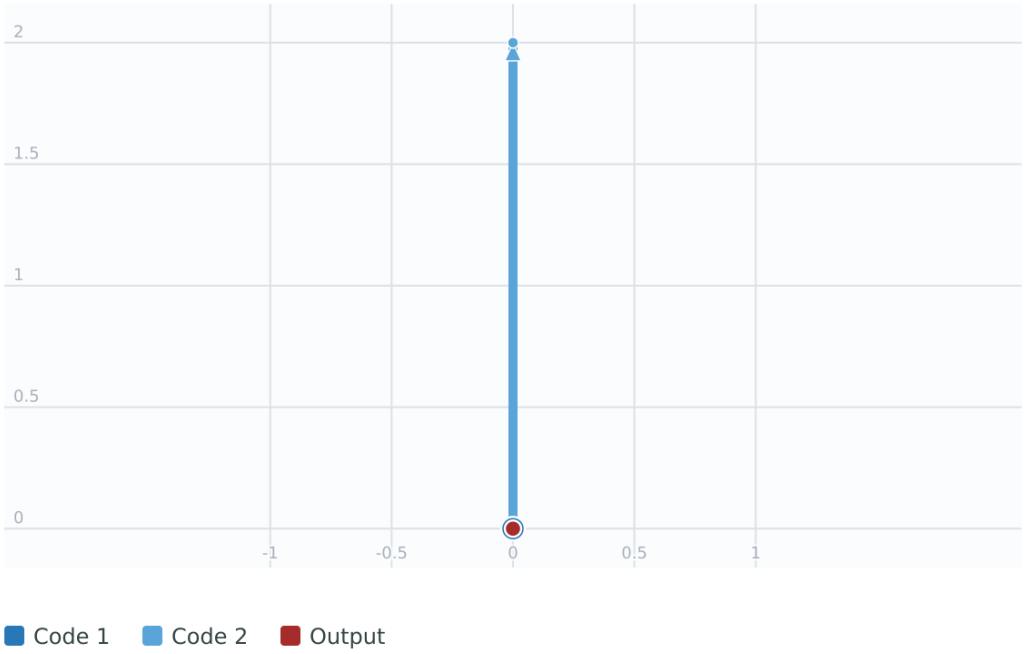
Code

```
SELECT ST_Intersection('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Output

```
POINT(0 0)
```

Figure



Clip all lines (trails) by country. Here we assume country geom are POLYGON or MULTIPOLYGONS. NOTE: we are only keeping intersections that result in a LINESTRING or MULTILINESTRING because we don't care about trails that just share a point. The dump is needed to expand a geometry collection into individual single MULT* parts. The below is fairly generic and will work for polys, etc. by just changing the where clause.

Code

```
select clipped.gid, clipped.f_name, clipped_geom
from (
         select trails.gid, trails.f_name,
             (ST_Dump(ST_Intersection(country.geom, trails.geom))).geom clipped_geom
         from country
              inner join trails on ST_Intersects(country.geom, trails.geom)
     ) as clipped
where ST_Dimension(clipped.clipped_geom) = 1;
```

For polys e.g. polygon landmarks, you can also use the sometimes faster hack that buffering anything by 0.0 except a polygon results in an empty geometry collection. (So a geometry collection containing polys, lines and points buffered by 0.0 would only leave the polygons and dissolve the collection shell.)

Code

```
select poly.gid,
    ST_Multi(ST_Buffer(ST_Intersection(country.geom, poly.geom),
             0.0
         )
    ) clipped_geom
from country
```

```
     inner join poly on ST_Intersects(country.geom, poly.geom)
where not ST_IsEmpty(ST_Buffer(ST_Intersection(country.geom, poly.geom), 0.0));
```

Note this is not a true 3D intersection. It uses 2.5D geometries (geometries with Z ordinates, but where calculations are performed in 2D). The same input is compared with CG_3DIntersection, which performs a true 3D intersection.

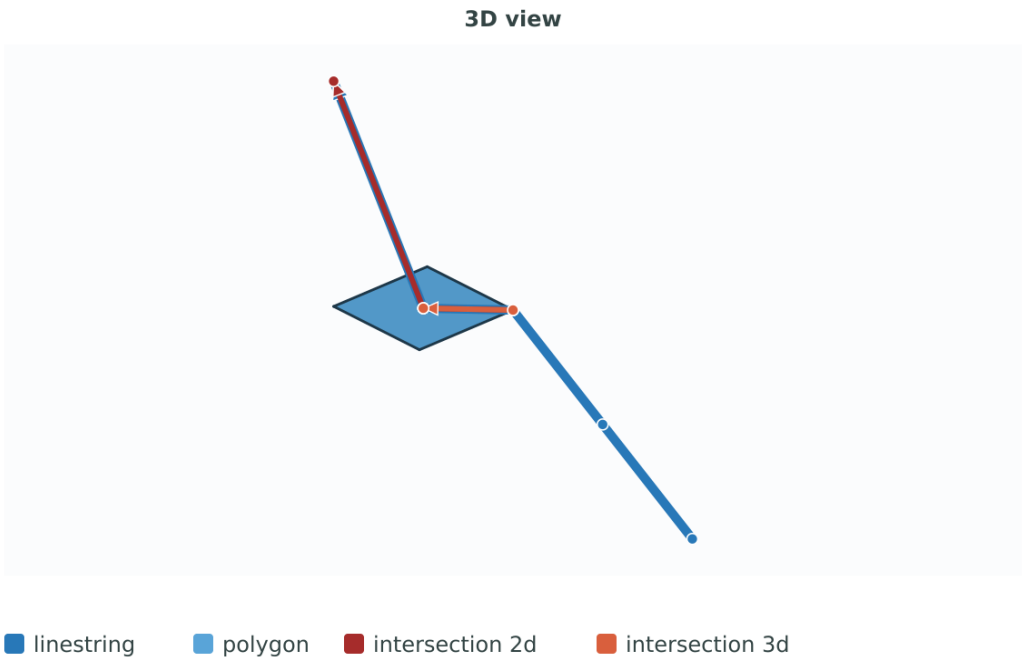
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10)'::geometry AS ↩
    input_linestring,
         'POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8))'::geometry AS input_polygon
)
SELECT input_linestring AS input_linestring,
       input_polygon AS input_polygon,
       ST_Intersection(input_linestring, input_polygon) AS intersection_2d,
       CG_3DIntersection(input_linestring, input_polygon) AS intersection_3d
FROM data;
```

Output

```
LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10) | POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8)) | LINESTRING Z (1 1 ↩
     8,0.5 0.5 8,0 0 10) | LINESTRING Z (1 1 8,0.5 0.5 8)
```

Figure



See Also

CG_3DIntersection, ST_Difference, ST_Union, ST_ClipByBox2D, ST_Dimension, ST_Dump, ST_Force2D, ST_SymDifference, ST_Intersects, ST_Multi, ST_ReducePrecision

7.13.4 ST_MemUnion

ST_MemUnion — Aggregate function which unions geometries in a memory-efficent but slower way

Synopsis

geometry **ST_MemUnion**(geometry set geomfield);

Description

An aggregate function that unions the input geometries, merging them to produce a result geometry with no overlaps. The output may be a single geometry, a MultiGeometry, or a Geometry Collection.



Note
Produces the same result as ST_Union, but uses less memory and more processor time. This aggregate function works by unioning the geometries incrementally, as opposed to the ST_Union aggregate which first accumulates an array and then unions the contents using a fast algorithm.



This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Examples

Code

```
SELECT id,
       ST_MemUnion(geom) as singlegeom
FROM sometable f
GROUP BY id;
```

See Also

ST_Union

7.13.5 ST_Node

ST_Node — Nodes a collection of lines.

Synopsis

geometry **ST_Node**(geometry geom);

Description

Returns a (Multi)LineString representing the fully noded version of a collection of linestrings. The noding preserves all of the input nodes, and introduces the least possible number of new nodes. The resulting linework is dissolved (duplicate lines are removed).

This is a good way to create fully-noded linework suitable for use as input to ST_Polygonize.

ST_UnaryUnion can also be used to node and dissolve linework. It provides an option to specify a gridSize, which can provide simpler and more robust output. See also ST_Union for an aggregate variant.



This function supports 3d and will not drop the z-index.

Performed by the GEOS module.

Availability: 2.0.0

Changed: 2.4.0 this function uses GEOSNode internally instead of GEOSUnaryUnion. This may cause the resulting linestrings to have a different order and direction compared to PostGIS < 2.4.

Examples

Noding a 3D LineString which self-intersects.

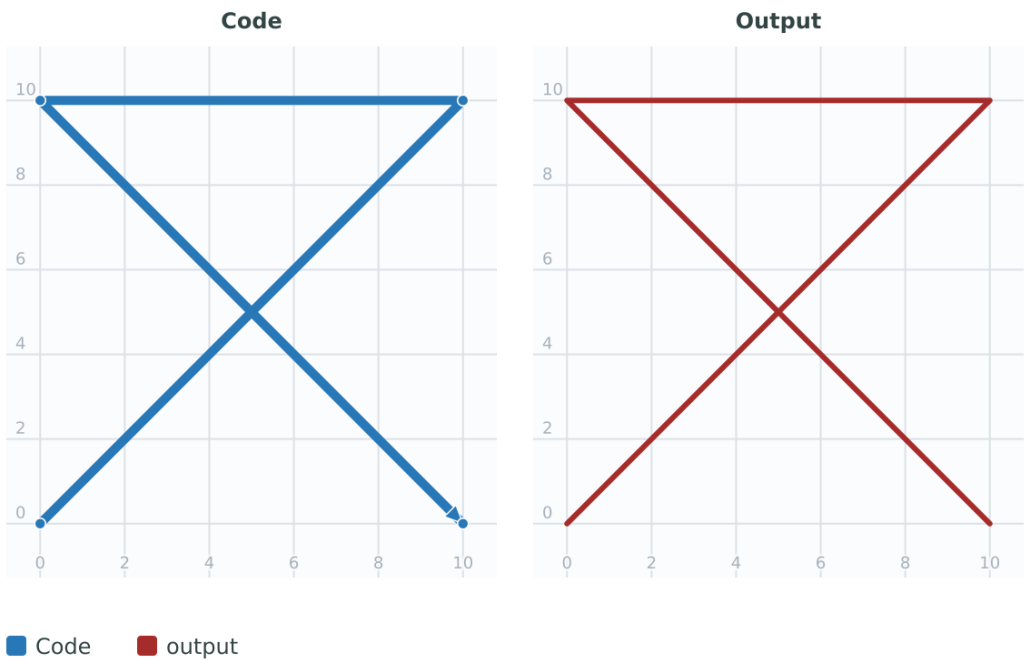
Code

```
SELECT ST_Node('LINESTRINGZ(0 0 0, 10 10 10, 0 10 5, 10 0 3)'::geometry) As  output;
```

Output

```
MULTILINESTRING Z ((0 0 0,5 5 4.5),(5 5 4.5,10 10 10,0 10 5,5 5 4.5),(5 5 4.5,10 0 3))
```

Figure



Noding two LineStrings which share common linework. Note that the result linework is dissolved.

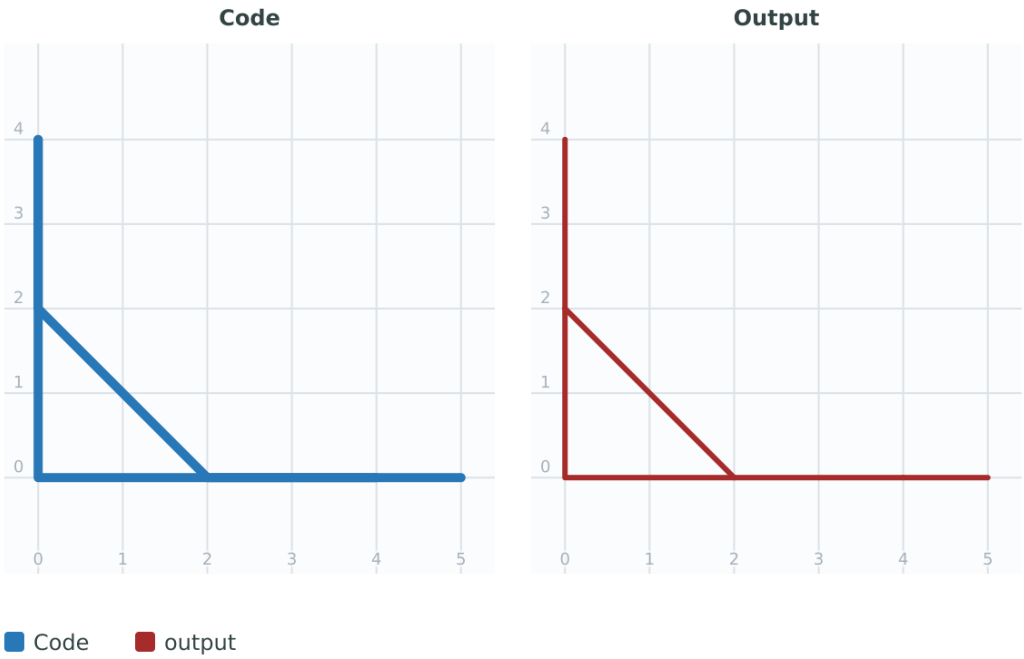
Code

```
SELECT ST_Node('MULTILINESTRING ((2 5,2 1,7 1),(6 1,4 1,2 3,2 5))'::geometry) As  output;
```

Output

```
MULTILINESTRING((2 5,2 3),(2 3,2 1,4 1),(4 1,2 3),(4 1,6 1),(6 1,7 1))
```

Figure



See Also

ST_UnaryUnion, ST_Union

7.13.6 ST_Split

ST_Split — Returns a collection of geometries created by splitting a geometry by another geometry.

Synopsis

geometry **ST_Split**(geometry input, geometry blade);

Description

The function supports splitting a LineString by a (Multi)Point, (Multi)LineString or (Multi)Polygon boundary, or a (Multi)Polygon by a LineString. When a (Multi)Polygon is used as as the blade, its linear components (the boundary) are used for splitting the input. The result geometry is always a collection.

This function is in a sense the opposite of ST_Union. Applying ST_Union to the returned collection should theoretically yield the original geometry (although due to numerical rounding this may not be exactly the case).



Note
If the the input and blade do not intersect due to numerical precision issues, the input may not be split as expected. To avoid this situation it may be necessary to snap the input to the blade first, using ST_Snap with a small tolerance.

Availability: 2.0.0 requires GEOS

Enhanced: 2.2.0 support for splitting a line by a multiline, a multipoint or (multi)polygon boundary was introduced.

Enhanced: 2.5.0 support for splitting a polygon by a multiline was introduced.

Examples

Split a circular polygon by a diagonal line.

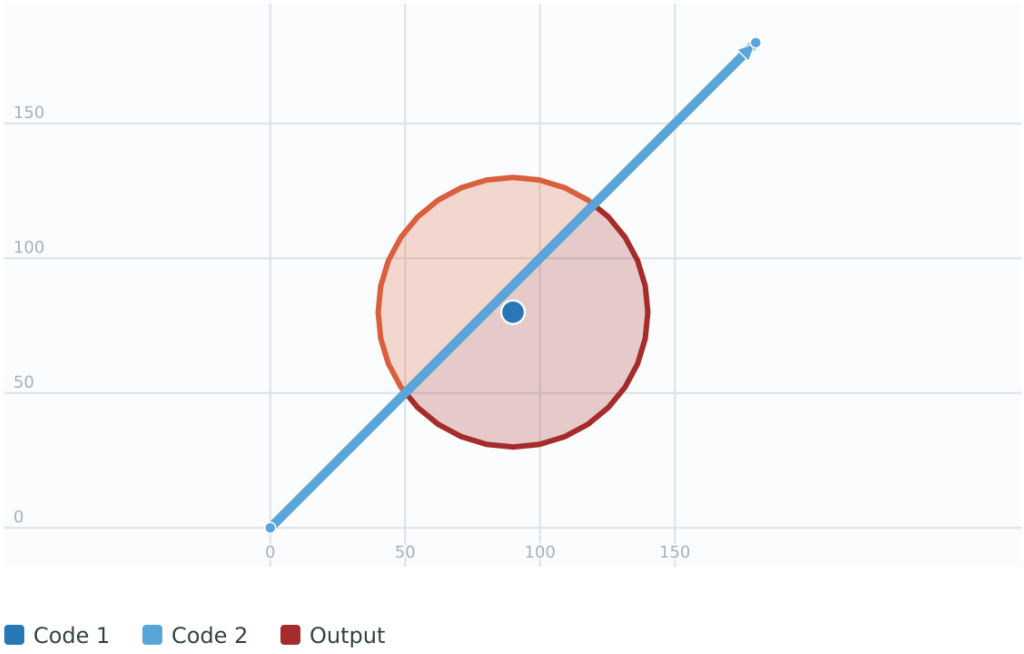
Code

```
SELECT
          ST_Split(
            ST_Buffer('POINT(100 90)'::geometry, 50),
            'LINESTRING(10 10,190 190)'::geometry
          );
```

Output

```
GEOMETRYCOLLECTION(POLYGON((150 90,149 80.2,146.2 70.9,141.6 62.2,135.4 54.6,127.8 48.4,119.1 43.8,109.8 41,100 40,90.2 ↩
     41,80.9 43.8,72.2 48.4,64.6 54.6,60.1 60.1,129.9 129.9,135.4 125.4,141.6 117.8,146.2 109.1,149 99.8,150 90)),  ↩
    POLYGON((60.1 60.1,58.4 62.2,53.8 70.9,51 80.2,50 90,51 99.8,53.8 109.1,58.4 117.8,64.6 125.4,72.2 131.6,80.9  ↩
    136.2,90.2 139,100 140,109.8 139,119.1 136.2,127.8 131.6,129.9 129.9,60.1 60.1)))
```

Figure



Split a MultiLineString by a Point, where the point lies exactly on both LineStrings elements.

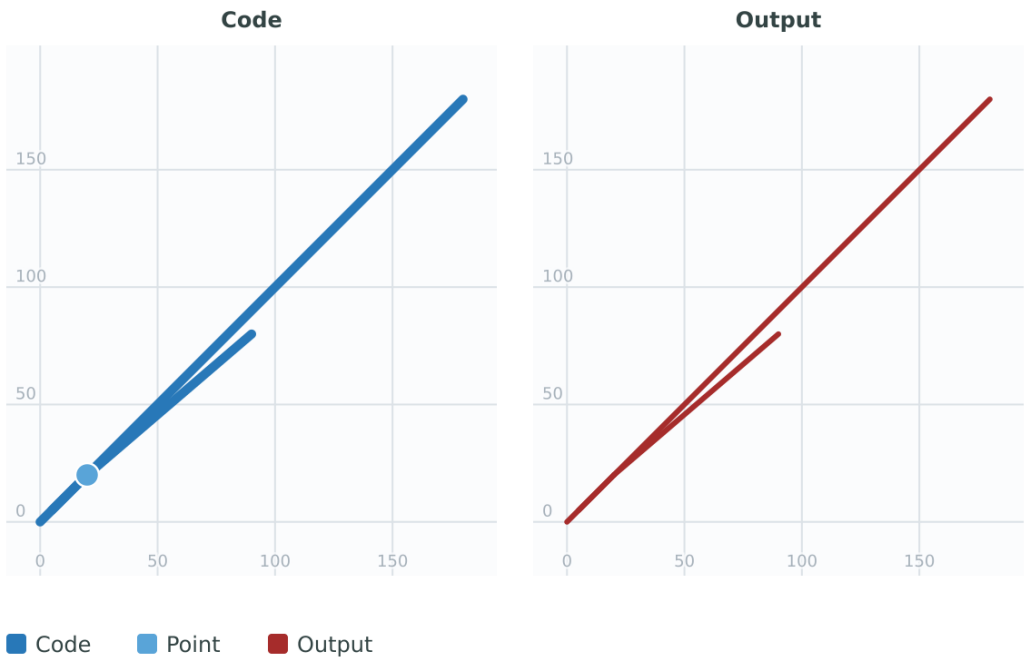
Code

```
SELECT ST_Split('MULTILINESTRING((10 10,190 190),(15 15,30 30,100 90))',
    ST_Point(30, 30));
```

Output

```
GEOMETRYCOLLECTION(
    LINESTRING(10 10,30 30),
    LINESTRING(30 30,190 190),
    LINESTRING(15 15,30 30),
    LINESTRING(30 30,100 90)
)
```

Figure

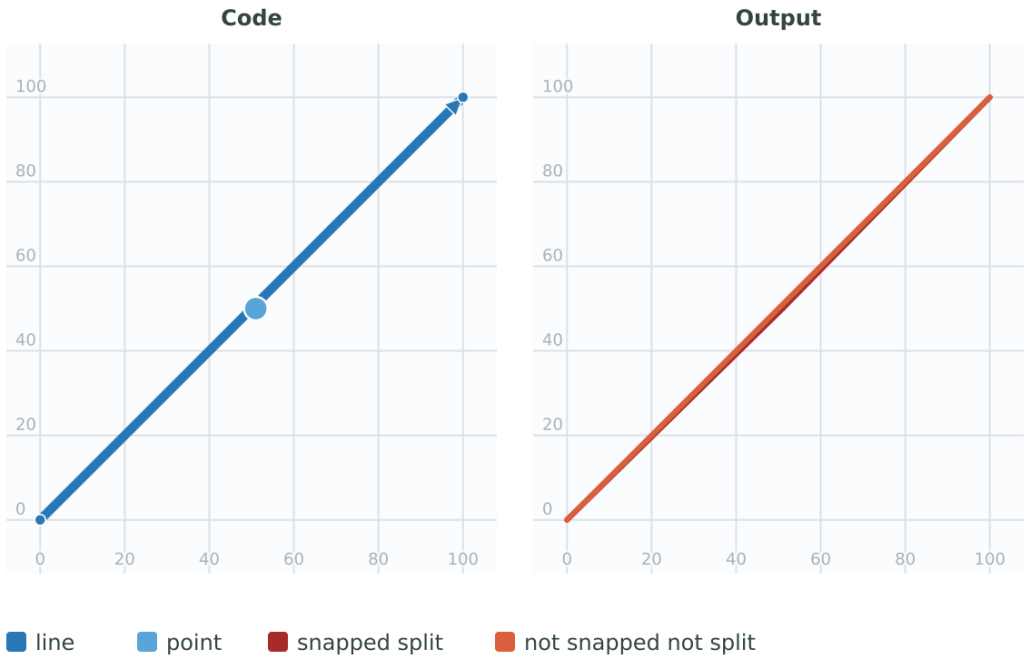


Split a LineString by a Point, where the point does not lie exactly on the line. Shows using ST_Snap to snap the line to the point to allow it to be split.

Code

```
WITH data AS (SELECT
  'LINESTRING(0 0,100 100)'::geometry AS line,
  'POINT(51 50)':: geometry AS point
)
SELECT ST_Split(ST_Snap(line, point, 1), point) AS snapped_split,
       ST_Split(line, point) AS not_snapped_not_split
       FROM data;
```

Figure



See Also

ST_Snap, ST_Union

7.13.7 ST_Subdivide

ST_Subdivide — Computes a rectilinear subdivision of a geometry.

Synopsis

setof geometry **ST_Subdivide**(geometry geom, integer max_vertices=256, float8 gridSize = -1);

Description

Returns a set of geometries that are the result of dividing `geom` into parts using rectilinear lines, with each part containing no more than `max_vertices`.

`max_vertices` must be 5 or more, as 5 points are needed to represent a closed box.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

Point-in-polygon and other spatial operations are normally faster for indexed subdivided datasets. Since the bounding boxes for the parts usually cover a smaller area than the original geometry bbox, index queries produce fewer "hit" cases. The "hit" cases are faster because the spatial operations executed by the index recheck process fewer points.



Note
When casting a subdivided geometry to geography, the resulting geography may differ from the original. Subdivision adds vertices in planar (geometry) space. If vertices are inserted along the boundary, they will alter the geographical representation, where edges are interpreted as geodesic segments. To minimize distortion, first densify the geography using ST_Segmentize to add geodesic vertices, then cast to geometry before subdivision.

Note
This is a set-returning function (SRF) that return a set of rows containing single geometry values. It can be used in a SELECT list or a FROM clause to produce a result set with one record for each result geometry.

Performed by the GEOS module.

Availability: 2.2.0

Enhanced: 2.5.0 reuses existing points on polygon split, vertex count is lowered from 8 to 5.

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

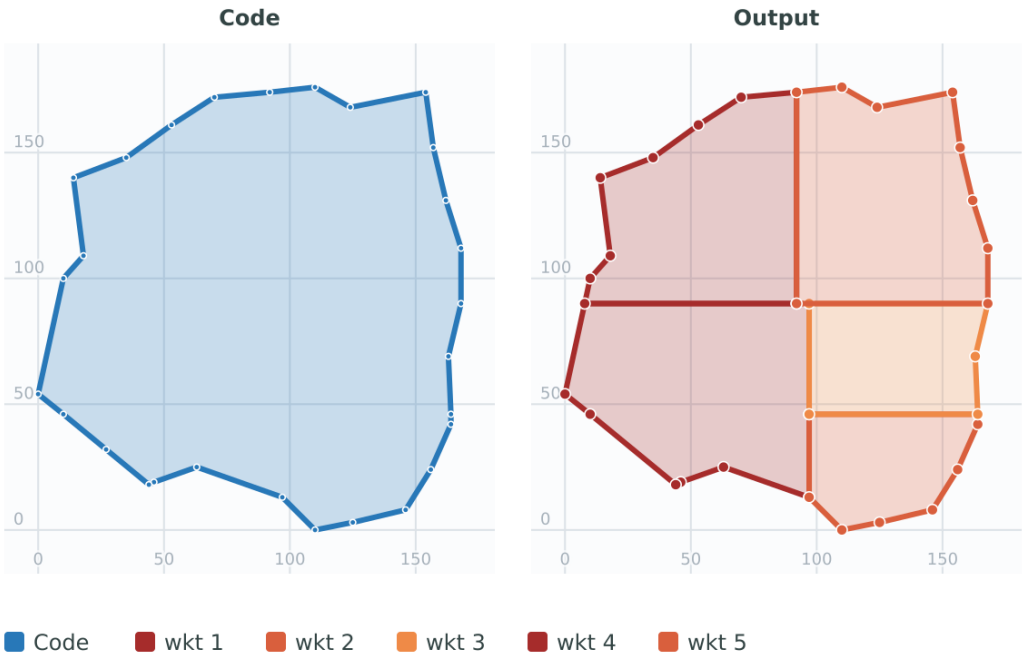
Examples

Example: Subdivide a polygon into parts with no more than 10 vertices, and assign each part a unique id.

Code

```
SELECT row_number() OVER() As rn,geom As wkt
    FROM (SELECT ST_Subdivide('POLYGON((132 10,119 23,85 35,68 29,66 28,49 42,32 56,22 64,32 ↩
     110,40 119,36 150,
        57 158,75 171,92 182,114 184,132 186,146 178,176 184,179 162,184 141,190 122,
        190 100,185 79,186 56,186 52,178 34,168 18,147 13,132 10))'::geometry, 10))  AS f( ↩
    geom);
```

Figure



Example: Densify a long geography line using ST_Segmentize(geography, distance), and use ST_Subdivide to split the resulting line into sublines of 8 vertices. Densification minimizes the impact of changes to the geography representation of a geometry when subdividing.

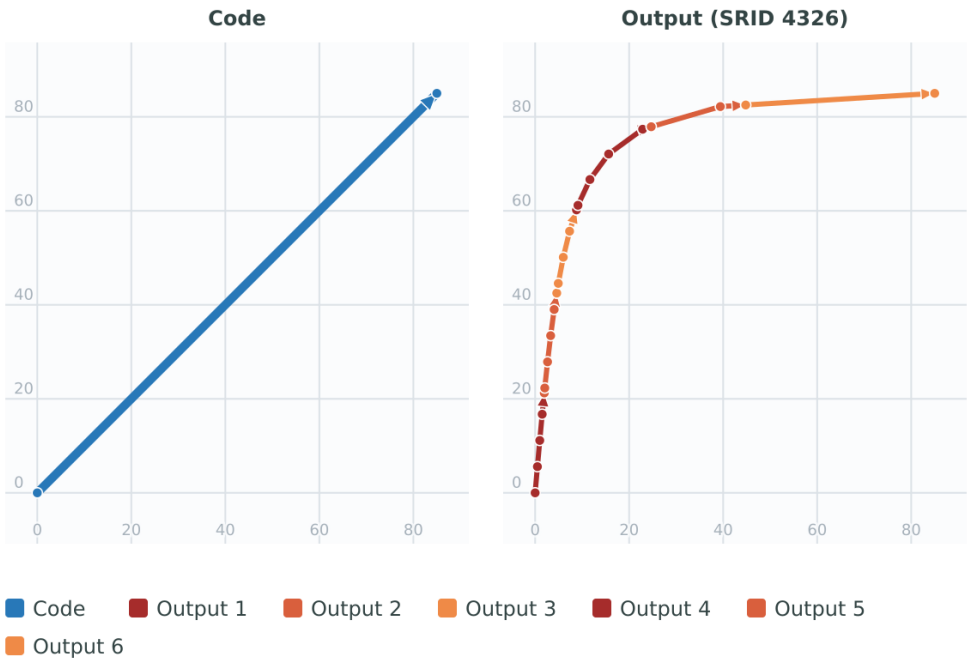
Code

```
SELECT ST_Subdivide(ST_Segmentize('LINESTRING(0 0,85 85)'::geography,
                          1200000)::geometry, 8);
```

Output

```
LINESTRING(0 0,0.487578359029357 5.57659056746196,0.984542144675897 11.1527721155093,1.50101059639722 ↩
    16.7281035483571,1.94532113630331 21.25)
LINESTRING(1.94532113630331 21.25,2.04869538062779 22.3020741387339,2.64204641967673 27.8740533545155,3.29994062412787 ↩
    33.443216802941,4.04836719489742 39.0084282520239,4.59890468420694 42.5)
LINESTRING(4.59890468420694 42.5,4.92498503922732 44.5680389206321,5.98737409390639 50.1195229244701,7.3290919767674 ↩
    55.6587646879025,8.79638749938413 60.1969505994924)
LINESTRING(8.79638749938413 60.1969505994924,9.11375579533779 61.1785363177625,11.6558166691368 ↩
    66.6648504160202,15.642041247655 72.0867690601745,22.8716627200212 77.3609628116894,24.6991785131552 ↩
    77.8939011989848)
LINESTRING(24.6991785131552 77.8939011989848,39.4046096622744 82.1822848017636,44.7994523421035 82.5156766227011)
LINESTRING(44.7994523421035 82.5156766227011,85 85)
```

Figure



Example: Subdivide the complex geometries of a table in-place. The original geometry records are deleted from the source table, and new records for each subdivided result geometry are inserted.

Code

```
WITH complex_areas_to_subdivide AS (
    DELETE from polygons_table
    WHERE ST_NPoints(geom) > 255
    RETURNING id, column1, column2, column3, geom
)
INSERT INTO polygons_table (fid, column1, column2, column3, geom)
    SELECT fid, column1, column2, column3,
           ST_Subdivide(geom, 255) as geom
    FROM complex_areas_to_subdivide;
```

Example: Create a new table containing subdivided geometries, retaining the key of the original geometry so that the new table can be joined to the source table. Since ST_Subdivide is a set-returning (table) function that returns a set of single-value rows, this syntax automatically produces a table with one row for each result part.

Code

```
CREATE TABLE subdivided_geoms AS
    SELECT pkey, ST_Subdivide(geom) AS geom
    FROM original_geoms;
```

See Also

ST_ClipByBox2D, ST_Segmentize, ST_Split, ST_NPoints, ST_ReducePrecision

7.13.8 ST_SymDifference

ST_SymDifference — Computes a geometry representing the portions of geometries A and B that do not intersect.

Synopsis

geometry **ST_SymDifference**(geometry geomA, geometry geomB, float8 gridSize = -1);

Description

Returns a geometry representing the portions of geometries A and B that do not intersect. This is equivalent to `ST_Union(A,B) - ST_Intersection(A,B)`. It is called a symmetric difference because `ST_SymDifference(A,B) = ST_SymDifference`

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

Performed by the GEOS module

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

This method implements the SQL/MM specification. SQL-MM 3: 5.1.21

This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Examples

This example shows a 2D-safe symmetric difference of two LineStrings.

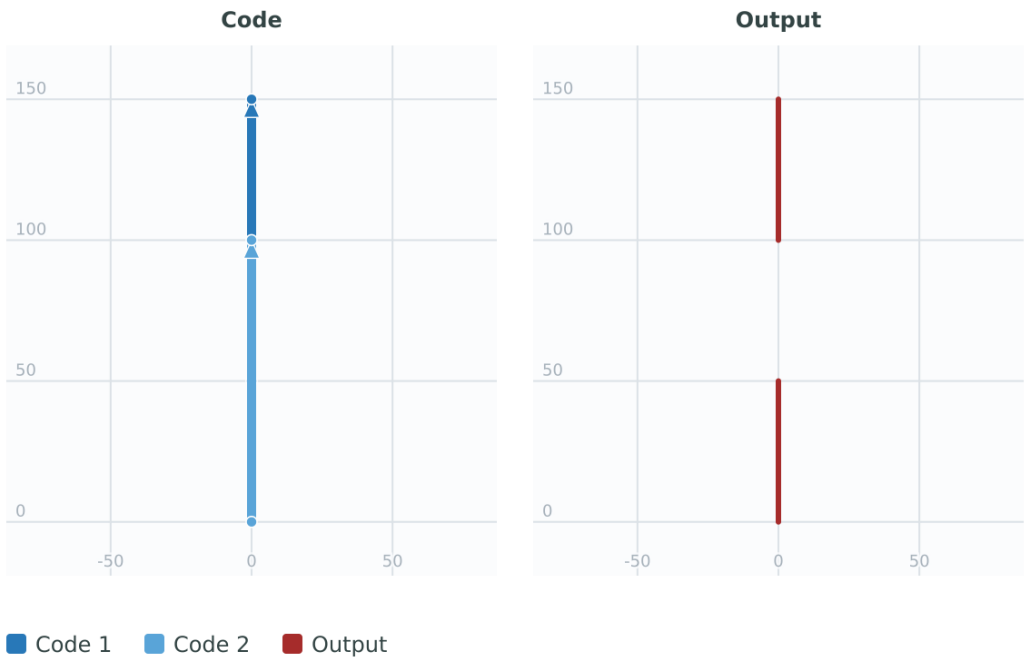
Code

```
SELECT ST_SymDifference('LINESTRING(50 100,50 200)',
        'LINESTRING(50 50,50 150)'
    );
```

Output

```
MULTILINESTRING((50 150,50 200),(50 50,50 100))
```

Figure



In 3D, this example does not produce the expected result.

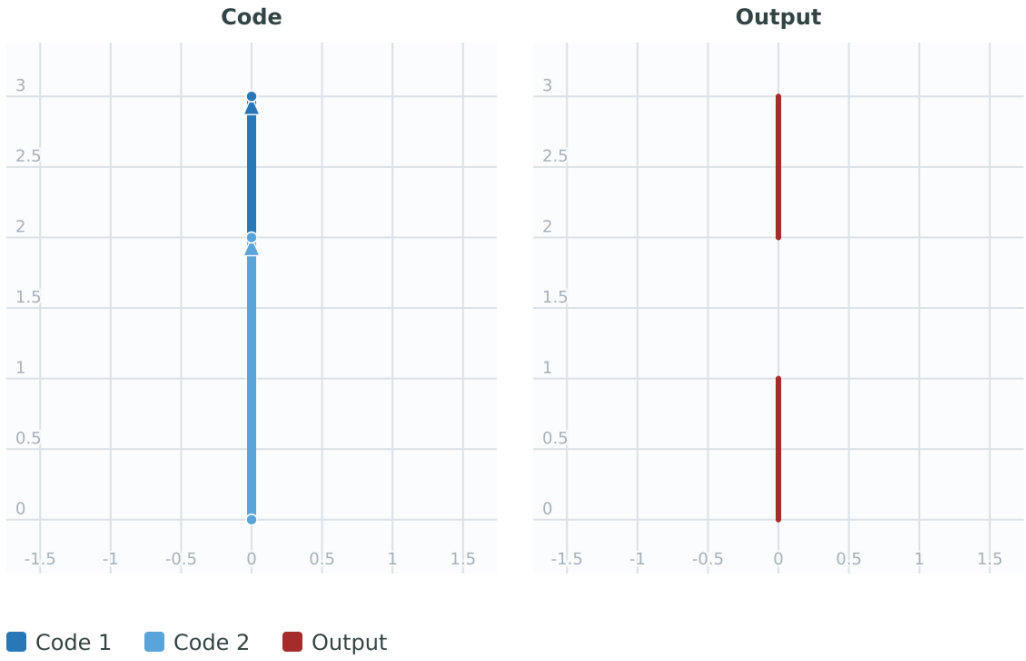
Code

```
SELECT ST_SymDifference('LINESTRING(1 2 1,1 4 2)',
    'LINESTRING(1 1 3,1 3 4)');
```

Output

```
MULTILINESTRING((1 3 4,1 4 2),(1 1 3,1 2 1))
```

Figure



See Also

ST_Difference, ST_Intersection, ST_Union, ST_ReducePrecision

7.13.9 ST_UnaryUnion

ST_UnaryUnion — Computes the union of the components of a single geometry.

Synopsis

geometry **ST_UnaryUnion**(geometry geom, float8 gridSize = -1);

Description

A single-input variant of ST_Union. The input may be a single geometry, a MultiGeometry, or a GeometryCollection. The union is applied to the individual elements of the input.

This function can be used to fix MultiPolygons which are invalid due to overlapping components. However, the input components must each be valid. An invalid input component such as a bow-tie polygon may cause an error. For this reason it may be better to use ST_MakeValid.

Another use of this function is to node and dissolve a collection of linestrings which cross or overlap to make them simple. (ST_Node also does this, but it does not provide the `gridSize` option.)

It is possible to combine ST_UnaryUnion with ST_Collect to fine-tune how many geometries are be unioned at once. This allows trading off between memory usage and compute time, striking a balance between ST_Union and ST_MemUnion.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

 This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

Availability: 2.0.0

See Also

ST_Union, ST_MemUnion, ST_MakeValid, ST_Collect, ST_Node, ST_ReducePrecision

7.13.10 ST_Union

ST_Union — Computes a geometry representing the point-set union of the input geometries.

Synopsis

geometry **ST_Union**(geometry g1, geometry g2);
geometry **ST_Union**(geometry g1, geometry g2, float8 gridSize);
geometry **ST_Union**(geometry[] g1_array);
geometry **ST_Union**(geometry set g1field);
geometry **ST_Union**(geometry set g1field, float8 gridSize);

Description

Unions the input geometries, merging geometry to produce a result geometry with no overlaps. The output may be an atomic geometry, a MultiGeometry, or a Geometry Collection. Comes in several variants:

Two-input variant: returns a geometry that is the union of two input geometries. If either input is NULL, then NULL is returned.

Array variant: returns a geometry that is the union of an array of geometries.

Aggregate variant: returns a geometry that is the union of a rowset of geometries. The ST_Union() function is an "aggregate" function in the terminology of PostgreSQL. That means that it operates on rows of data, in the same way the SUM() and AVG() functions do and like most aggregates, it also ignores NULL geometries.

See ST_UnaryUnion for a non-aggregate, single-input variant.

The ST_Union array and set variants use the fast Cascaded Union algorithm described in http://blog.cleverelephant.ca/2009/01/-must-faster-unions-in-postgis-14.html

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.



Note
ST_Collect may sometimes be used in place of ST_Union, if the result is not required to be non-overlapping. ST_Collect is usually faster than ST_Union because it performs no processing on the collected geometries.

Performed by the GEOS module.

ST_Union creates MultiLineString and does not sew LineStrings into a single LineString. Use ST_LineMerge to sew LineStrings.

NOTE: this function was formerly called GeomUnion(), which was renamed from "Union" because UNION is an SQL reserved word.

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

Changed: 3.0.0 does not depend on SFCGAL.

Availability: 1.4.0 - ST_Union was enhanced. ST_Union(geomarray) was introduced and also faster aggregate collection in PostgreSQL.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3



Note
Aggregate version is not explicitly defined in OGC SPEC.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.19 the z-index (elevation) when polygons are involved.

This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Examples

Aggregate example

Code

```
SELECT id,
       ST_Union(geom) as singlegeom
FROM sometable f
GROUP BY id;
```

Non-Aggregate example

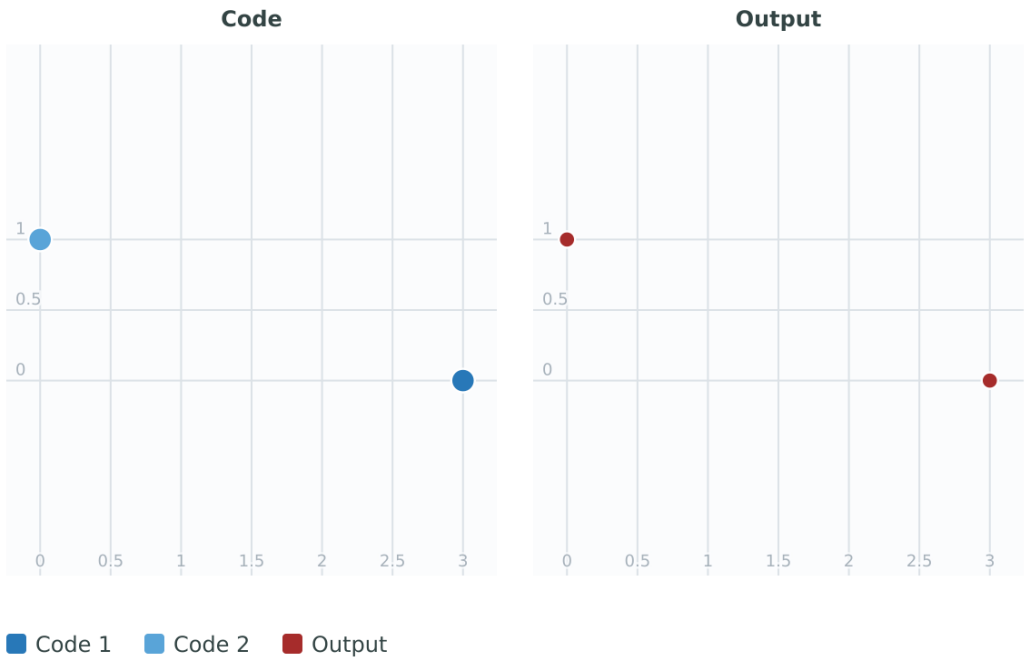
Code

```
SELECT ST_Union('POINT(1 2)'::geometry, 'POINT(-2 3)'::geometry);
```

Output

```
MULTIPOINT((1 2),(-2 3))
```

Figure



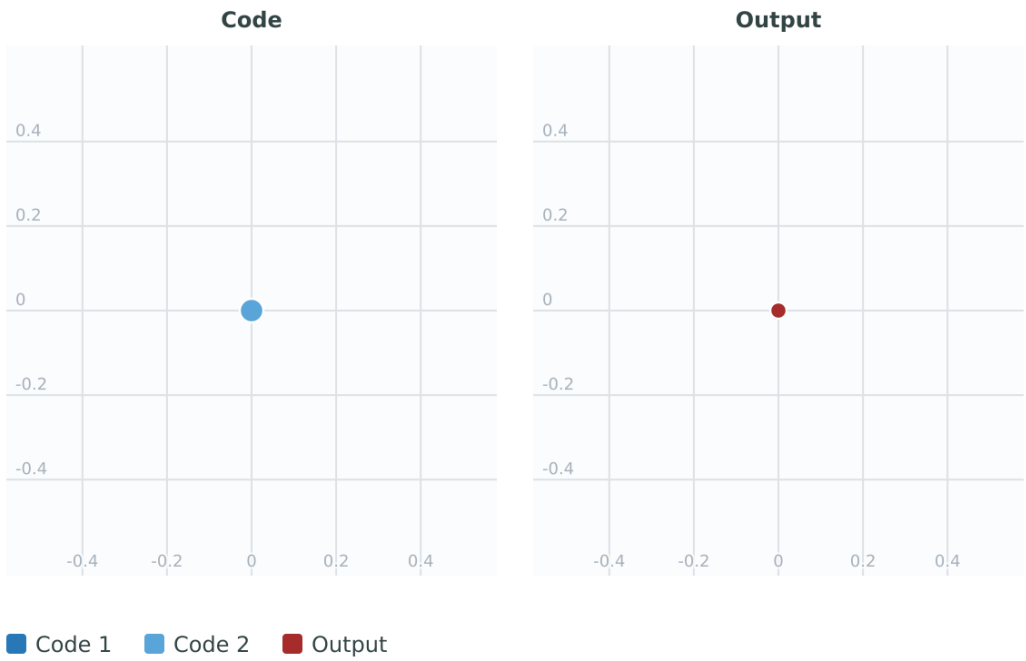
Code

```
SELECT ST_Union('POINT(1 2)'::geometry, 'POINT(1 2)'::geometry);
```

Output

```
POINT(1 2)
```

Figure



3D example - sort of supports 3D (and with mixed geometry types!)

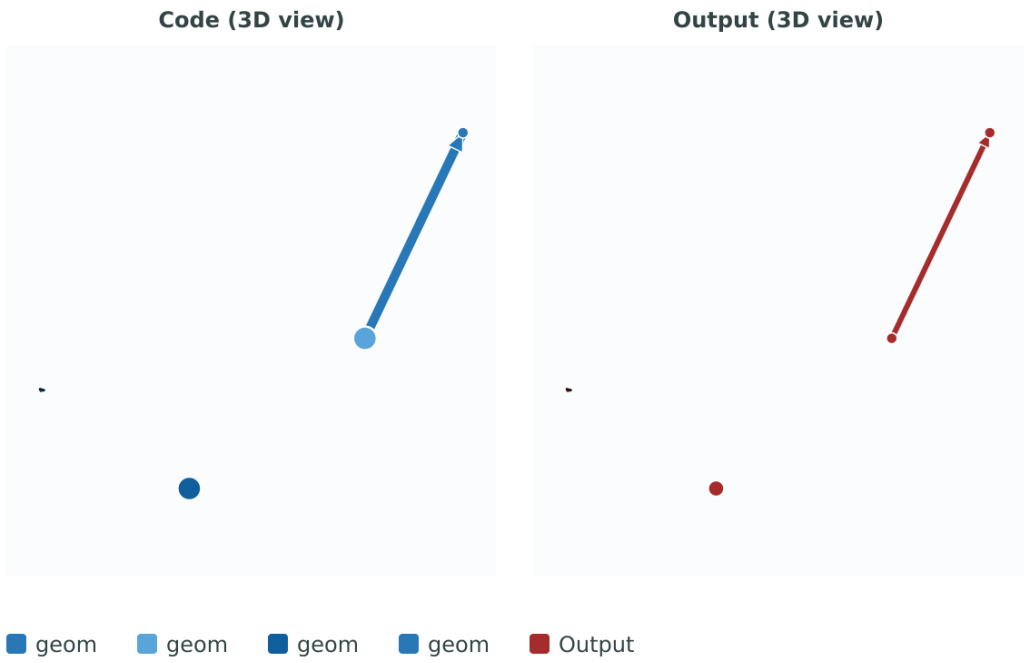
Code

```
SELECT ST_Union(geom)
FROM (
    SELECT 'POLYGON Z((-7 4.2 5,-7.1 4.2 5,-7.1 4.3 5,-7 4.2 5))'::geometry geom
    UNION ALL
    SELECT 'POINT Z(5 5 5)'::geometry geom
    UNION ALL
    SELECT 'POINT Z(-2 3 1)'::geometry geom
    UNION ALL
    SELECT 'LINESTRING Z(5 5 5,10 10 10)'::geometry geom
) AS foo;
```

Output

```
GEOMETRYCOLLECTION(POINT(-2 3 1), LINESTRING(5 5 5,10 10 10), POLYGON((-7 4.2 5,-7.1 4.2 5,-7.1 4.3 5,-7 4.2 5)))
```

Figure



This 3D example does not mix dimensions.

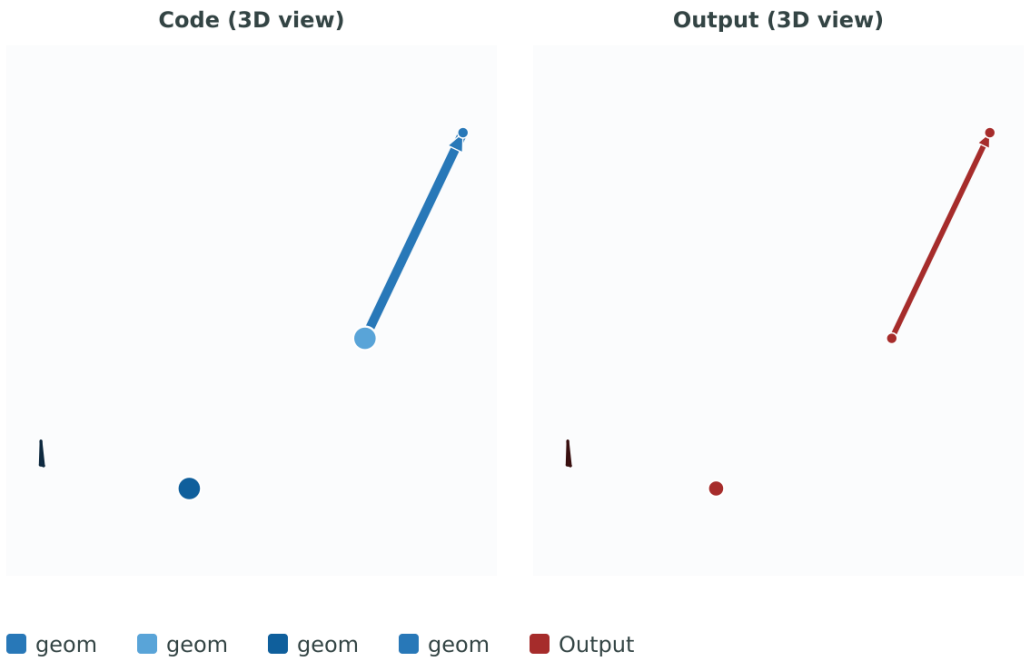
Code

```
SELECT ST_Union(geom)
FROM (
    SELECT 'POLYGON((-7 4.2 2,-7.1 4.2 3,-7.1 4.3 2,-7 4.2 2))'::geometry geom
    UNION ALL
    SELECT 'POINT(5 5 5)'::geometry geom
    UNION ALL
    SELECT 'POINT(-2 3 1)'::geometry geom
    UNION ALL
    SELECT 'LINESTRING(5 5 5,10 10 10)'::geometry geom
) AS foo;
```

Output

```
GEOMETRYCOLLECTION(POINT(-2 3 1), LINESTRING(5 5 5,10 10 10), POLYGON((-7 4.2 2,-7.1 4.2 3,-7.1 4.3 2,-7 4.2 2)))
```

Figure



This example unions an array built from a query.

Code

```
SELECT ST_Union(ARRAY(SELECT geom FROM sometable));
```

This example unions an array of linestrings.

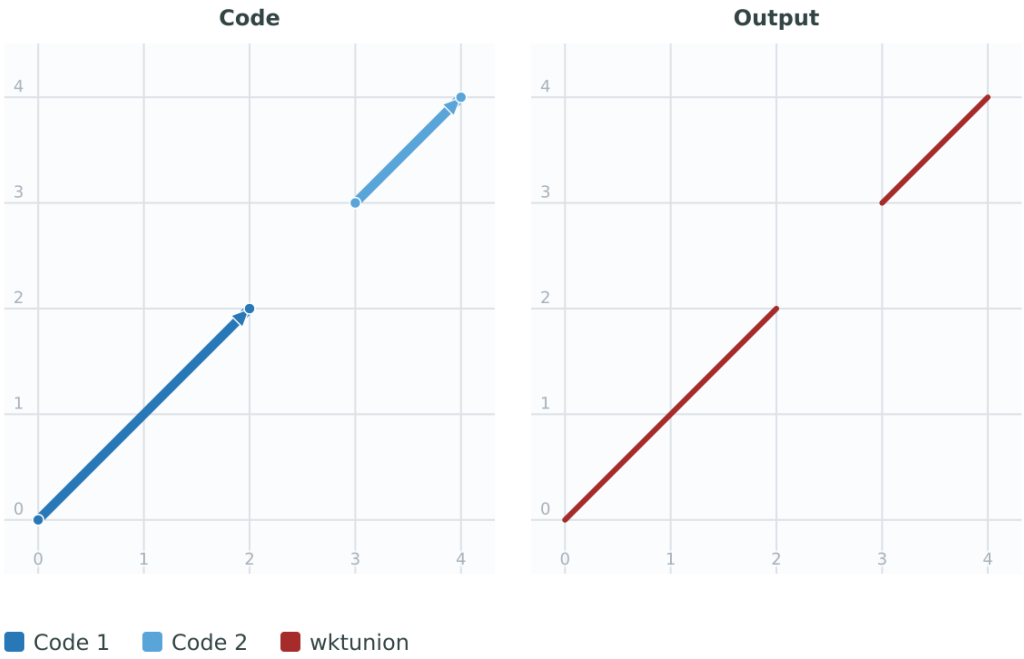
Code

```
SELECT ST_Union(ARRAY[
    'LINESTRING(1 2,3 4)'::geometry,
    'LINESTRING(4 5,5 6)'::geometry
]) As wktunion;
```

Output

```
MULTILINESTRING((1 2,3 4),(4 5,5 6))
```

Figure



See Also

ST_CoverageUnion, ST_UnaryUnion, ST_MemUnion, ST_Collect, ST_Intersection, ST_Difference, ST_SymDifference, ST_LineMerg

7.14 Geometry Processing

7.14.1 ST_Buffer

ST_Buffer — Computes a geometry covering all points within a given distance from a geometry.

Synopsis

geometry **ST_Buffer**(geometry g1, float radius_of_buffer, text buffer_style_parameters = '');
geometry **ST_Buffer**(geometry g1, float radius_of_buffer, integer num_seg_quarter_circle);
geography **ST_Buffer**(geography g1, float radius_of_buffer, text buffer_style_parameters);
geography **ST_Buffer**(geography g1, float radius_of_buffer, integer num_seg_quarter_circle);

Description

Computes a POLYGON or MULTIPOLYGON that represents all points whose distance from a geometry/geography is less than or equal to a given distance. A negative distance shrinks the geometry rather than expanding it. A negative distance may shrink a polygon completely, in which case POLYGON EMPTY is returned. For points and lines negative distances always return empty results.

For geometry, the distance is specified in the units of the Spatial Reference System of the geometry. For geography, the distance is specified in meters.

The optional third parameter controls the buffer accuracy and style. The accuracy of circular arcs in the buffer is specified as the number of line segments used to approximate a quarter circle (default is 8). The buffer style can be specified by providing a list of blank-separated key=value pairs as follows:

- 'quad_segs=#' : number of line segments used to approximate a quarter circle (default is 8).
- 'endcap=round|flat|square' : endcap style (defaults to "round"). 'butt' is accepted as a synonym for 'flat'.
- 'join=round|mitre|bevel' : join style (defaults to "round"). 'miter' is accepted as a synonym for 'mitre'.
- 'mitre_limit=#.#' : mitre ratio limit (only affects mitered join style). 'miter_limit' is accepted as a synonym for 'mitre_limit'.
- 'side=both|left|right' : defaults to 'both'. 'left' or 'right' performs a single-sided buffer on the geometry, with the buffered side relative to the direction of the line. This is only applicable to LINESTRING geometry and does not affect POINT or POLYGON geometries. By default end caps are square when 'left' or 'right' are specified.



Note



For geography this is a thin wrapper around the geometry implementation. It determines a planar spatial reference system that best fits the bounding box of the geography object (trying UTM, Lambert Azimuthal Equal Area (LAEA) North/South pole, and finally Mercator). The buffer is computed in the planar space, and then transformed back to WGS84. This may not produce the desired behavior if the input object is much larger than a UTM zone or crosses the dateline

---



Note

Buffer can handle invalid inputs and the output is always a valid polygonal geometry. Buffering by distance 0 is sometimes used as a way of repairing invalid polygons. ST_MakeValid is more suitable for this process as it can handle multi-polygons.



Note

Buffering is sometimes used to perform a within-distance search. For this use case it is more efficient to use ST_DWithin.



Note

This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

Enhanced: 2.5.0 - ST_Buffer geometry support was enhanced to allow for side buffering specification `side=both|left|right`.

Availability: 1.5 - ST_Buffer was enhanced to support different endcaps and join types. These are useful for example to convert road linestrings into polygon roads with flat or square edges instead of rounded edges. Thin wrapper for geography was added.

Performed by the GEOS module.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3



This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.30

Examples

The textual results are rounded to whole coordinate units for readability. Each result remains available as text, while the manual initially favors the shared input/output figure.

---

quad_segs=8 (default)

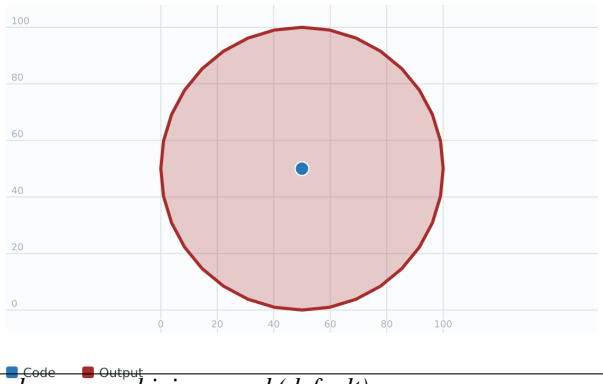
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('POINT(100 90)'),
  50, 'quad_segs=8');
```

Output

```
POLYGON((150 90,149 80,146 71,142 62,135 55,128 48,119
44,110 41,100 40,90 41,81 44,72 48,65 55,58 62,54 71,51
80,50 90,51 100,54 109,58 118,65 125,72 132,81 136,90
139,100 140,110 139,119 136,128 132,135 125,142 118,146
109,149 100,150 90))
```

Figure



quad_segs=2

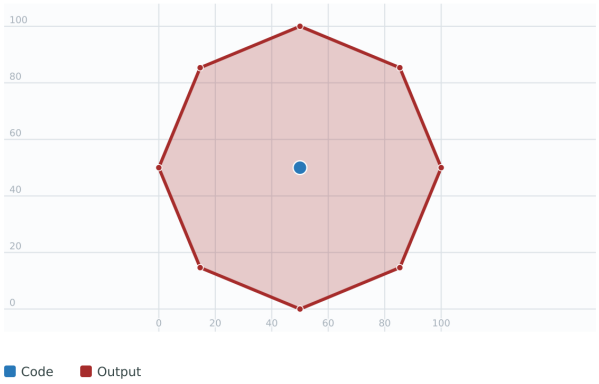
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('POINT(100 90)'),
  50, 'quad_segs=2');
```

Output

```
POLYGON((150 90,135 55,100 40,65 55,50 90,65 125,100
140,135 125,150 90))
```

Figure



endcap=round join=round (default)

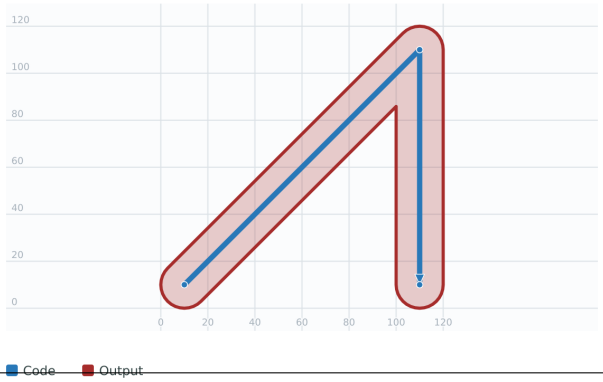
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'endcap=round join=round');
```

Output

```
POLYGON((143 157,144 158,146 159,148 160,150 160,152
160,154 159,156 158,157 157,158 156,159 154,160 152,160
150,160 50,160 48,159 46,158 44,157 43,156 42,154 41,152
40,150 40,148 40,146 41,144 42,143 43,142 44,141 46,140
48,140 50,140 126,57 43,56 42,54 41,52 40,50 40,48 40,46
41,44 42,43 43,42 44,41 46,40 48,40 50,40 52,41 54,42
56,43 57,143 157))
```

Figure



endcap=square

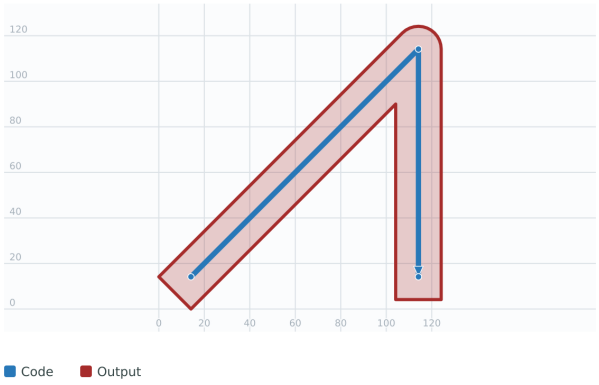
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'endcap=square join=round');
```

Output

```
POLYGON((143 157,144 158,146 159,148 160,150 160,152
160,154 159,156 158,157 157,158 156,159 154,160 152,160
150,160 50,160 40,140 40,140 126,57 43,50 36,36 50,143
157))
```

Figure



endcap=flat

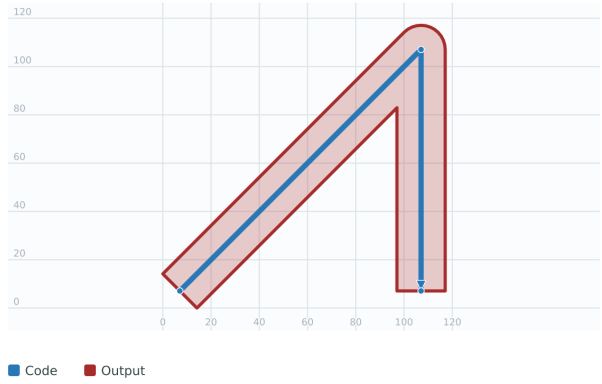
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'endcap=flat join=round');
```

Output

```
POLYGON((143 157,144 158,146 159,148 160,150 160,152
160,154 159,156 158,157 157,158 156,159 154,160 152,160
150,160 50,140 50,140 126,57 43,43 57,143 157))
```

Figure



join=bevel

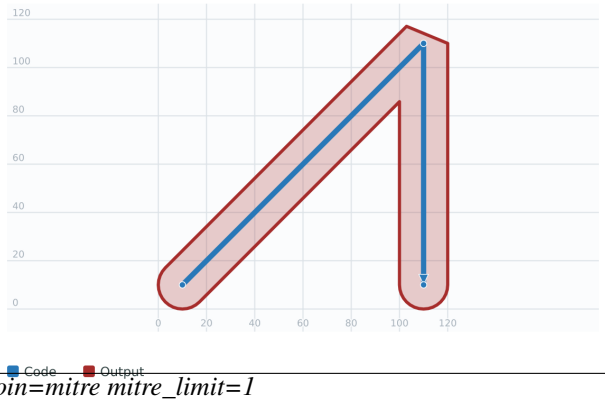
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'join=bevel');
```

Output

```
POLYGON((143 157,160 150,160 50,160 48,159 46,158 44,157
43,156 42,154 41,152 40,150 40,148 40,146 41,144 42,143
43,142 44,141 46,140 48,140 50,140 126,57 43,56 42,54
41,52 40,50 40,48 40,46 41,44 42,43 43,42 44,41 46,40
48,40 50,40 52,41 54,42 56,43 57,143 157))
```

Figure



join=mitre mitre_limit=5.0 (default mitre limit)

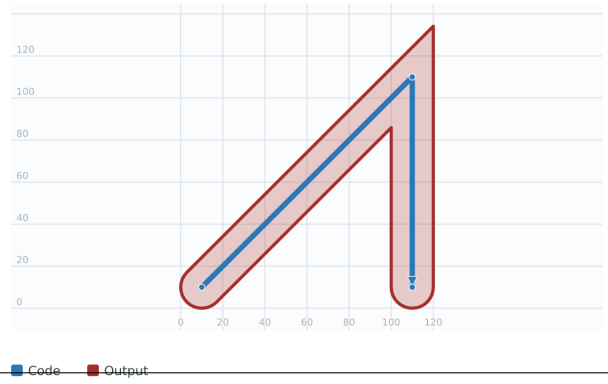
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'join=mitre mitre_limit=5.0');
```

Output

```
POLYGON((160 174,160 50,160 48,159 46,158 44,157 43,156
42,154 41,152 40,150 40,148 40,146 41,144 42,143 43,142
44,141 46,140 48,140 50,140 126,57 43,56 42,54 41,52 40,
50 40,48 40,46 41,44 42,43 43,42 44,41 46,40 48,40 50,40
52,41 54,42 56,43 57,160 174))
```

Figure



join=mitre mitre_limit=1

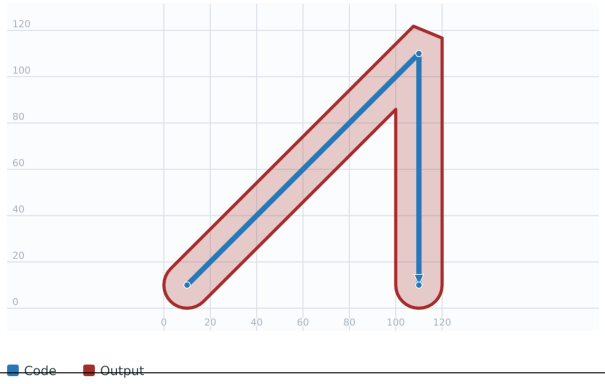
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'join=mitre mitre_limit=1.0');
```

Output

```
POLYGON((148 162,160 157,160 50,160 48,159 46,158 44,157
43,156 42,154 41,152 40,150 40,148 40,146 41,144 42,143
43,142 44,141 46,140 48,140 50,140 126,57 43,56 42,54
41,52 40,50 40,48 40,46 41,44 42,43 43,42 44,41 46,40
48,40 50,40 52,41 54,42 56,43 57,148 162))
```

Figure



side=left

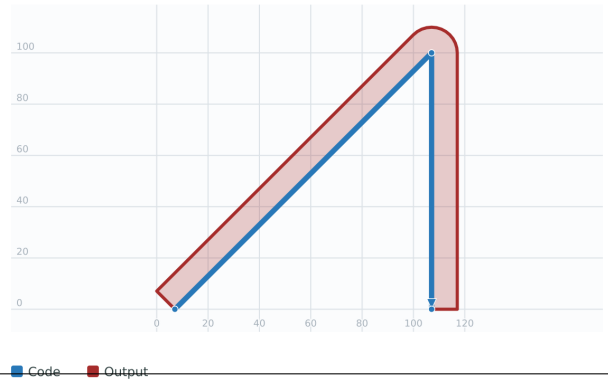
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'side=left');
```

Output

```
POLYGON((150 50,150 150,50 50,43 57,143 157,144 158,146
159,148 160,150 160,152 160,154 159,156 158,157 157,158
156,159 154,160 152,160 150,160 50,150 50))
```

Figure



side=right

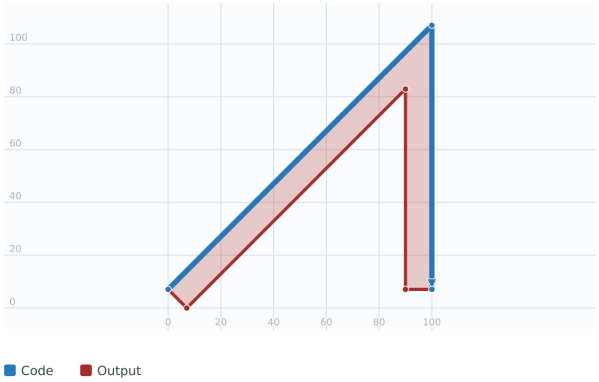
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'side=right');
```

Output

```
POLYGON((50 50,150 150,150 50,140 50,140 126,57 43,50
50))
```

Figure



side=left join=mitre

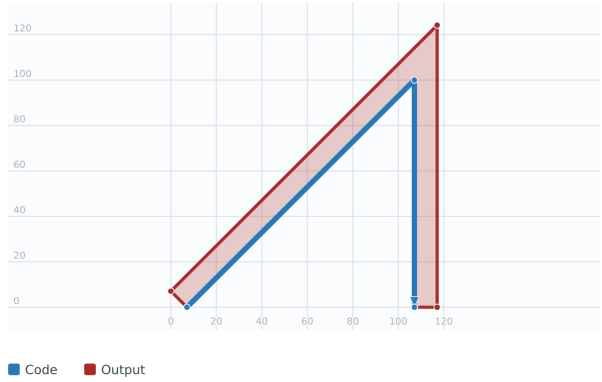
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
    150,150 50)'),
  10, 'side=left join=mitre');
```

Output

```
POLYGON((150 50,150 150,50 50,43 57,160 174,160 50,150
50))
```

Figure



right-hand winding, polygon boundary side=left

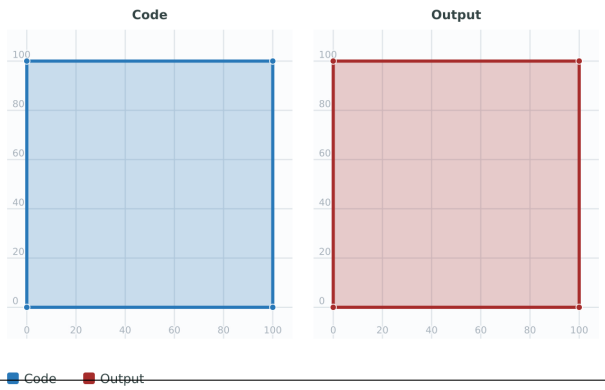
Code

```
SELECT ST_Buffer(
  ST_ForceRHR(ST_Boundary(ST_GeomFromText ↩
    (
    'POLYGON ((50 50,50 150,150 150,150 ↩
    50,50 50))'))),
  20, 'side=left');
```

Output

```
POLYGON((50 50,50 150,150 150,150 50,50 50))
```

Figure



right-hand winding, polygon boundary side=right

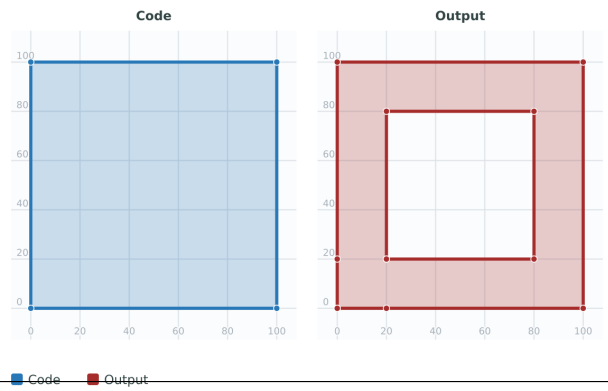
Code

```
SELECT ST_Buffer(
  ST_ForceRHR(ST_Boundary(ST_GeomFromText ↩
    (
    'POLYGON ((50 50,50 150,150 150,150  ↩
    50,50 50))')))),
  20, 'side=right');
```

Output

```
POLYGON((50 50,50 70,50 150,150 150,150 50,70 50,50 50),
(70 70,130 70,130 130,70 130,70 70))
```

Figure



A buffered point approximates a circle. A buffered point forcing approximation of (see diagram) 2 points per quarter circle is poly with 8 sides (see diagram)

Code

```
SELECT ST_NPoints(ST_Buffer('POINT(100 90)'::geometry, 50)) As promisingcircle_pcount,
ST_NPoints(ST_Buffer('POINT(100 90)'::geometry, 50, 2)) As lamecircle_pcount;
```

Output

```
promisingcircle_pcount | lamecircle_pcount
-----------------------+-------------------
                    33 |                 9
```

A lighter, less smooth circle using only 2 points per quarter circle is an octagon.

The following example creates a 100 meter octagon in NAD83 longitude/latitude by transforming to the Massachusetts state plane meter projection and then buffering. The figure shows the input and output in separate panels because the coordinates are in different SRIDs.

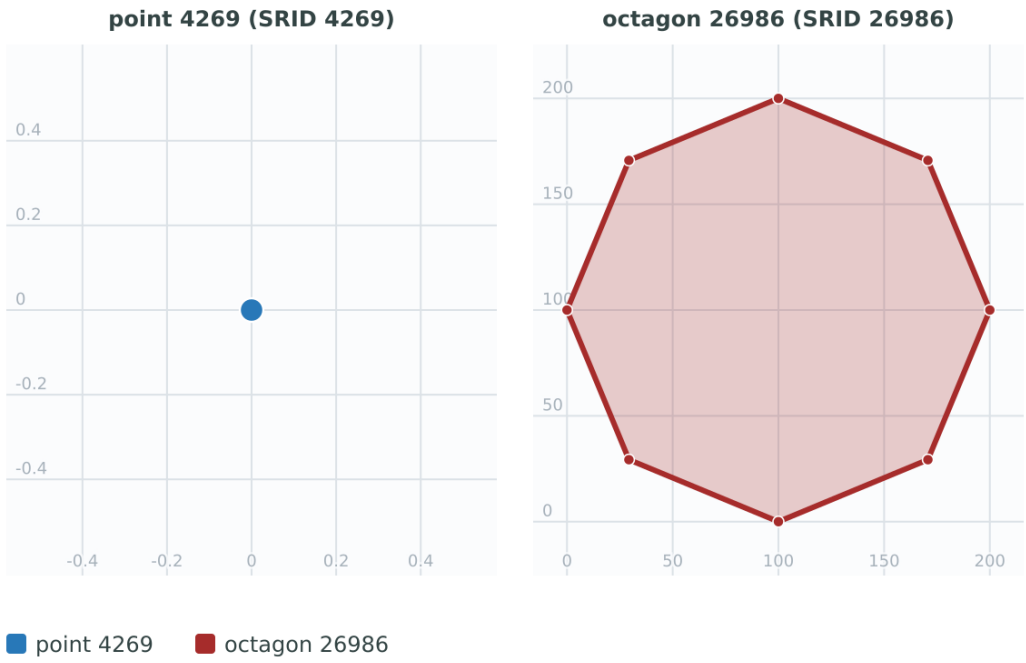
Code

```
WITH input AS (
  SELECT ST_SetSRID(ST_Point(-71.063526, 42.35785), 4269) AS point_4269
), projected AS (
  SELECT point_4269,
         ST_Transform(point_4269, 26986) AS point_26986
  FROM input
)
SELECT point_4269 AS input_point_4269,
       ST_Buffer(point_26986, 100, 2) AS octagon_26986
FROM projected;
```

Output

```
SRID=4269;POINT(-71.063526 42.35785) | SRID=26986;POLYGON((236057.5905746494 900908.7599186979,236028.30125276805 ↩
    900838.0492405792,235957.5905746494 900808.7599186979,235886.87989653074 900838.0492405792,235857.5905746494 ↩
    900908.7599186979,235886.87989653074 900979.4705968165,235957.5905746494 901008.7599186979,236028.30125276805 ↩
    900979.4705968165,236057.5905746494 900908.7599186979))
```

Figure



Buffering a polygon by a positive distance and then by the same negative distance is a morphological closing operation. It can be useful for smoothing noisy boundaries, such as simplifying a coastline by removing narrow inlets and bays. The result intentionally changes the shape and area of the input, and the distance must be chosen in the units of the input SRS. For data in a geographic SRS, transform to a projected SRS suited to the operation first; for cartographic visualization in Web Mercator, using that display SRS can be appropriate.

Smooth a coastline using a 1 km closing operation, then simplify the result. The example uses Massachusetts State Plane meter units for the operation.

Code

```
WITH projected AS (
  SELECT id, ST_Transform(geom, 26986) AS geom
  FROM coast
),
closed AS (
  SELECT id, ST_Buffer(ST_Buffer(geom, 1000), -1000) AS geom
  FROM projected
)
SELECT id, ST_Transform(ST_SimplifyPreserveTopology(geom, 50), 4326) AS geom
FROM closed;
```

The same technique is described in Paul Ramsey's Removing Complexities. The final simplification step is optional; use ST_SimplifyPreserveTopology when polygonal validity matters, and check the result against the cartographic or analytical tolerance required by the application.

A small negative buffer can also be used as a heuristic to find narrow polygon spikes. The query below compares how much area and perimeter remain after erosion. A large `erosion_index` indicates that the perimeter dropped proportionally more than the area. This is only a preselection aid; the threshold and buffer distance must be chosen for the data units and the kind of spike being checked.

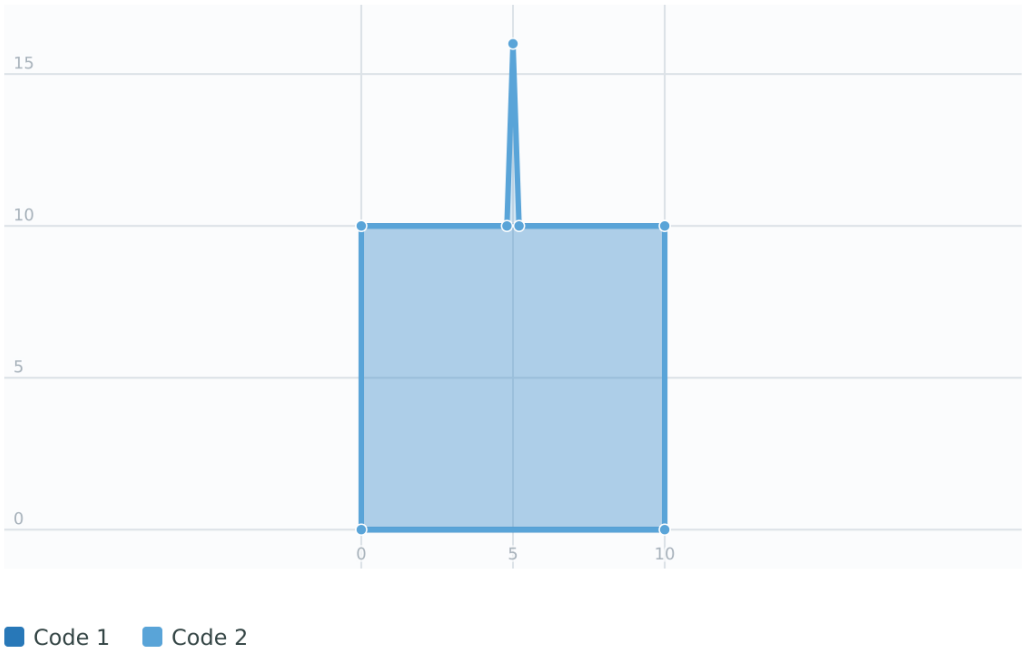
Code

```
WITH sample(id, geom) AS (
  VALUES
    ('plain', 'POLYGON((0 0,10 0,10 10,0 10,0 0))'::geometry),
    ('spike', 'POLYGON((0 0,10 0,10 10,5.2 10,5 16,4.8 10,0 10,0 0))'::geometry)
),
eroded AS (
  SELECT id, geom, ST_Buffer(geom, -0.5) AS eroded_geom
  FROM sample
),
scored AS (
  SELECT id,
         round((
           ST_Area(eroded_geom) / ST_Area(geom) /
           NULLIF(ST_Perimeter(eroded_geom) / ST_Perimeter(geom), 0)
         )::numeric, 3) AS erosion_index
  FROM eroded
  WHERE NOT ST_IsEmpty(eroded_geom)
)
SELECT id, erosion_index, erosion_index > 1.1 AS possible_spike
FROM scored
ORDER BY id;
```

Output

```
  id   | erosion_index | possible_spike
-------+---------------+----------------
 plain |         0.900 | f
 spike |         1.147 | t
```

Figure



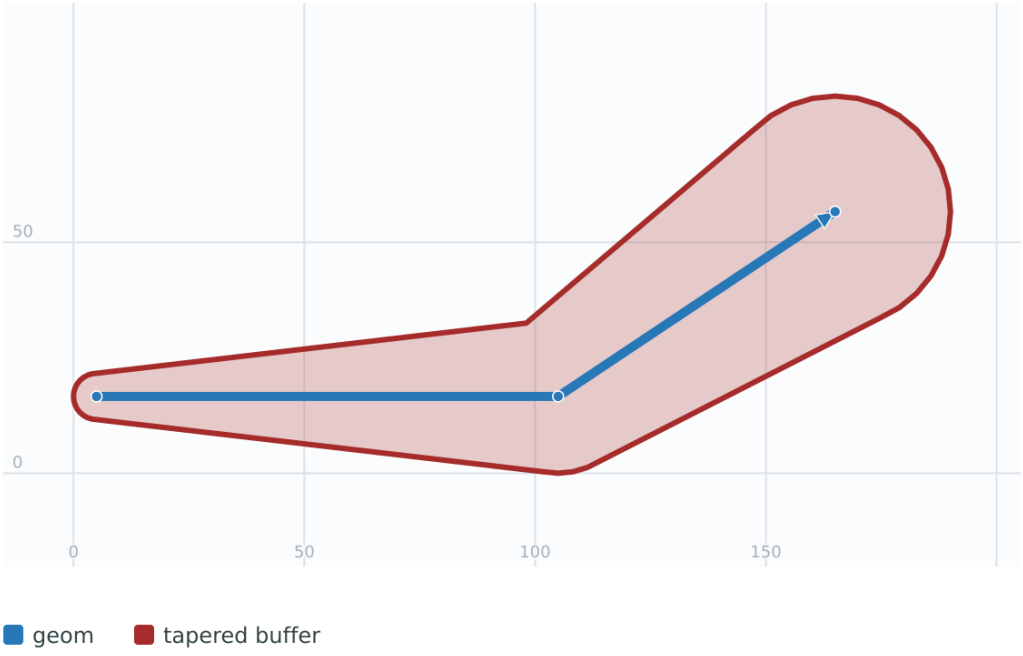
A variable-distance buffer can be approximated by buffering each vertex of a line by a distance interpolated along the line, building the convex hull for each consecutive pair of vertex buffers, and unioning the segment hulls. As with any geometry buffer, distances are in the units of the input SRS.

Taper a line buffer from 5 to 25 units along the line.

Code

```
WITH line AS (
  SELECT 'LINESTRING(0 0, 100 0, 160 40)'::geometry AS geom
),
vertices AS (
  SELECT (dump).path[1] AS n,
         (dump).geom AS geom,
         line.geom AS line_geom
  FROM line
  CROSS JOIN LATERAL ST_DumpPoints(line.geom) AS dump
),
radii AS (
  SELECT n,
         geom,
         5 + (25 - 5) * ST_LineLocatePoint(line_geom, geom) AS radius
  FROM vertices
),
segments AS (
  SELECT n,
         geom,
         radius,
         lead(geom) OVER (ORDER BY n) AS next_geom,
         lead(radius) OVER (ORDER BY n) AS next_radius
  FROM radii
)
SELECT ST_Union(
         ST_ConvexHull(
           ST_Collect(
             ST_Buffer(geom, radius),
             ST_Buffer(next_geom, next_radius)
           )
         )
       ) AS tapered_buffer
FROM segments
WHERE next_geom IS NOT NULL;
```

Figure



See Also

ST_Collect, ST_ConvexHull, ST_DWithin, ST_DumpPoints, ST_LineLocatePoint, ST_SetSRID, ST_SimplifyPreserveTopology, ST_Transform, ST_Union, ST_MakeValid

7.14.2 ST_BuildArea

ST_BuildArea — Creates a polygonal geometry formed by the linework of a geometry.

Synopsis

geometry **ST_BuildArea**(geometry geom);

Description

Creates an areal geometry formed by the constituent linework of the input geometry. The input can be a LineString, MultiLineString, Polygon, MultiPolygon or a GeometryCollection. The result is a Polygon or MultiPolygon, depending on input. If the input linework does not form polygons, NULL is returned.

Unlike ST_MakePolygon, this function accepts rings formed by multiple lines, and can form any number of polygons.

This function converts inner rings into holes. To turn inner rings into polygons as well, use ST_Polygonize.



Note
Input linework must be correctly noded for this function to work properly. ST_Node can be used to node lines.
If the input linework crosses, this function will produce invalid polygons. ST_MakeValid can be used to ensure the output is valid.

Availability: 1.1.0

Examples

Building an area from five input lines.

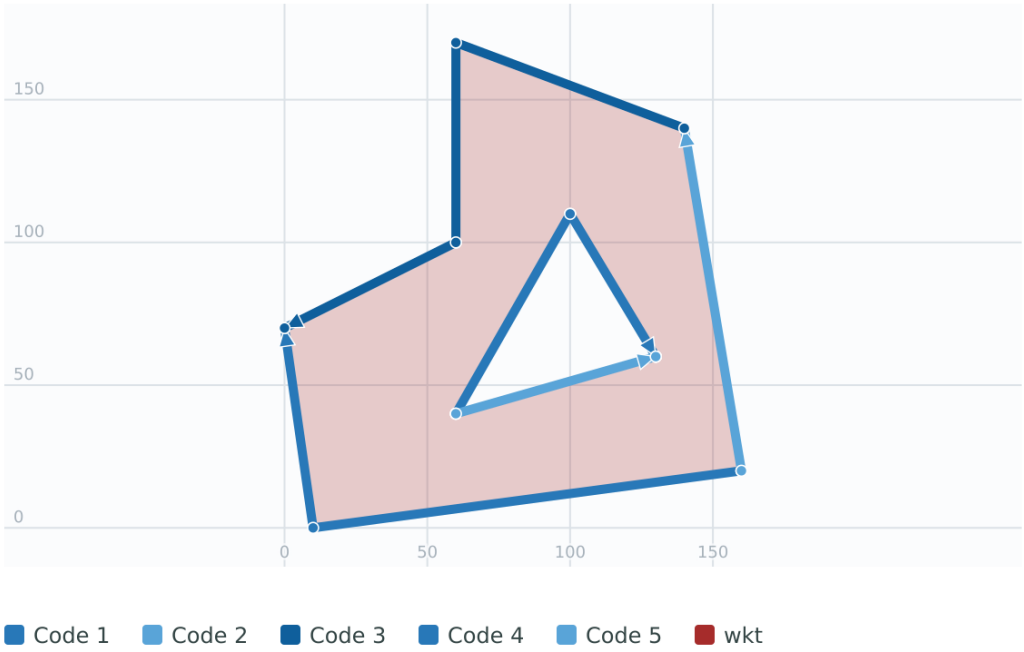
Code

```
WITH data(geom) AS (VALUES
   ('LINESTRING (180 40,30 20,20 90)'::geometry),
  ('LINESTRING (180 40,160 160)'::geometry),
  ('LINESTRING (160 160,80 190,80 120,20 90)'::geometry),
  ('LINESTRING (80 60,120 130,150 80)'::geometry),
  ('LINESTRING (80 60,150 80)'::geometry)
)
SELECT ST_BuildArea(ST_Collect(geom )) AS wkt
    FROM data;
```

Output

```
POLYGON((180 40,30 20,20 90,80 120,80 190,160 160,180 40),(150 80,120 130,80 60,150 80))
```

Figure

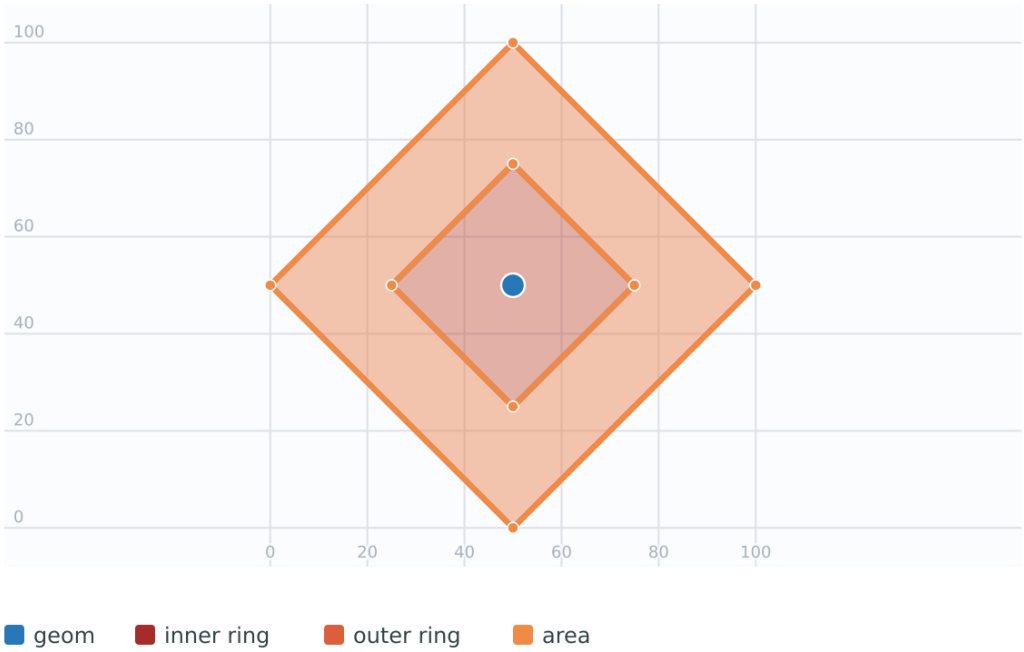


Create a donut from two circular polygons.

Code

```
WITH input AS (
  SELECT 'POINT(100 90)'::geometry AS geom
),
rings AS (
  SELECT ST_Buffer(geom, 25, 'quad_segs=1') AS inring,
         ST_Buffer(geom, 50, 'quad_segs=1') AS outring
  FROM input
)
SELECT inring AS inner_ring,
       outring AS outer_ring,
       ST_BuildArea(ST_Collect(inring, outring)) AS area
FROM rings;
```

Figure



Input linework that crosses at non-endpoint intersections must be noded before building areas:

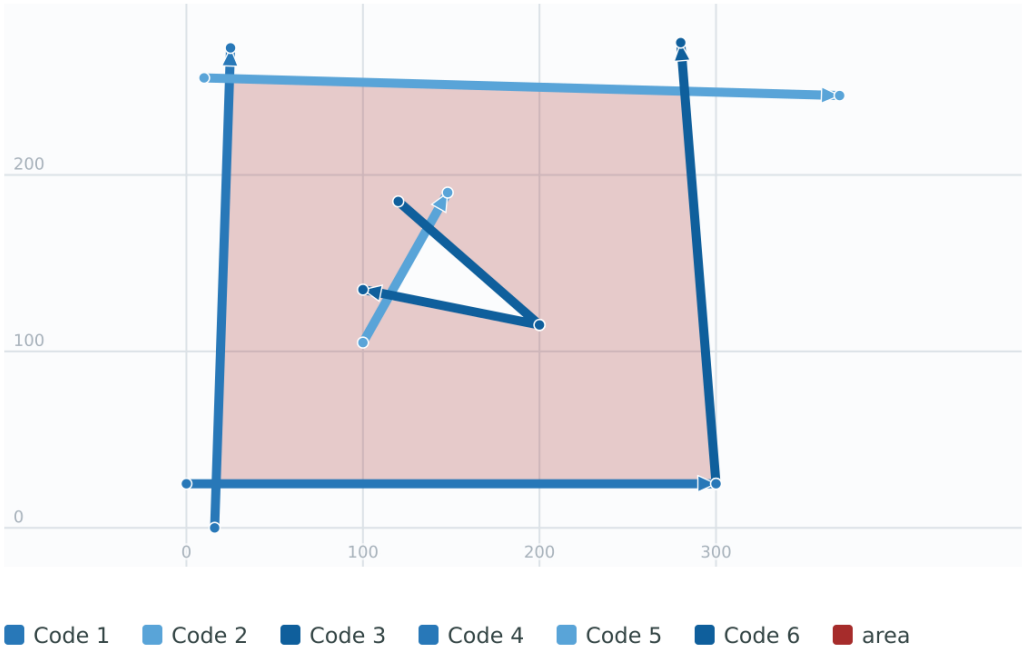
Code

```
WITH lines(geom) AS (VALUES
  ('LINESTRING(36 35,45 307)'::geometry),
  ('LINESTRING(30 290,390 280)'::geometry),
  ('LINESTRING(320 60,300 310)'::geometry),
  ('LINESTRING(20 60,320 60)'::geometry),
  ('LINESTRING(120 140,168 225)'::geometry),
  ('LINESTRING(140 220,220 150,120 170)'::geometry)
),
noded AS (
  SELECT ST_Node(ST_UnaryUnion(ST_Collect(geom))) AS geom FROM lines
)
SELECT ST_BuildArea(geom) AS area
FROM noded;
```

Output

```
POLYGON((320 60,36.8 60,44.4 289.6,302.2 282.4,320 60),(156.9 205.3,135.2 167,220 150,156.9 205.3))
```

Figure



See Also

ST_Collect, ST_MakePolygon, ST_MakeValid, ST_Node, ST_Polygonize, ST_BdPolyFromText, ST_BdMPolyFromText (wrappers to this function with standard OGC interface)

7.14.3 ST_Centroid

ST_Centroid — Returns the geometric center of a geometry.

Synopsis

geometry **ST_Centroid**(geometry g1);
geography **ST_Centroid**(geography g1, boolean use_spheroid = true);

Description

Computes a point which is the geometric center of mass of a geometry. For [MULTI]POINTs, the centroid is the arithmetic mean of the input coordinates. For [MULTI]LINESTRINGs, the centroid is computed using the weighted length of each line segment. For [MULTI]POLYGONs, the centroid is computed in terms of area. If an empty geometry is supplied, an empty GEOMETRYCOLLECTION is returned. If NULL is supplied, NULL is returned. If CIRCULARSTRING or COMPOUNDCURVE are supplied, they are converted to linestring with CurveToLine first, then same than for LINESTRING

For mixed-dimension input, the result is equal to the centroid of the component Geometries of highest dimension (since the lower-dimension geometries contribute zero "weight" to the centroid).

Note that for polygonal geometries the centroid does not necessarily lie in the interior of the polygon. For example, see the diagram below of the centroid of a C-shaped polygon. To construct a point guaranteed to lie in the interior of a polygon use ST_PointOnSurface.

New in 2.3.0 : supports CIRCULARSTRING and COMPOUNDCURVE (using CurveToLine)

Availability: 2.4.0 support for geography was introduced.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 8.1.4, 9.5.5

Examples

Centroid of a MultiPoint.

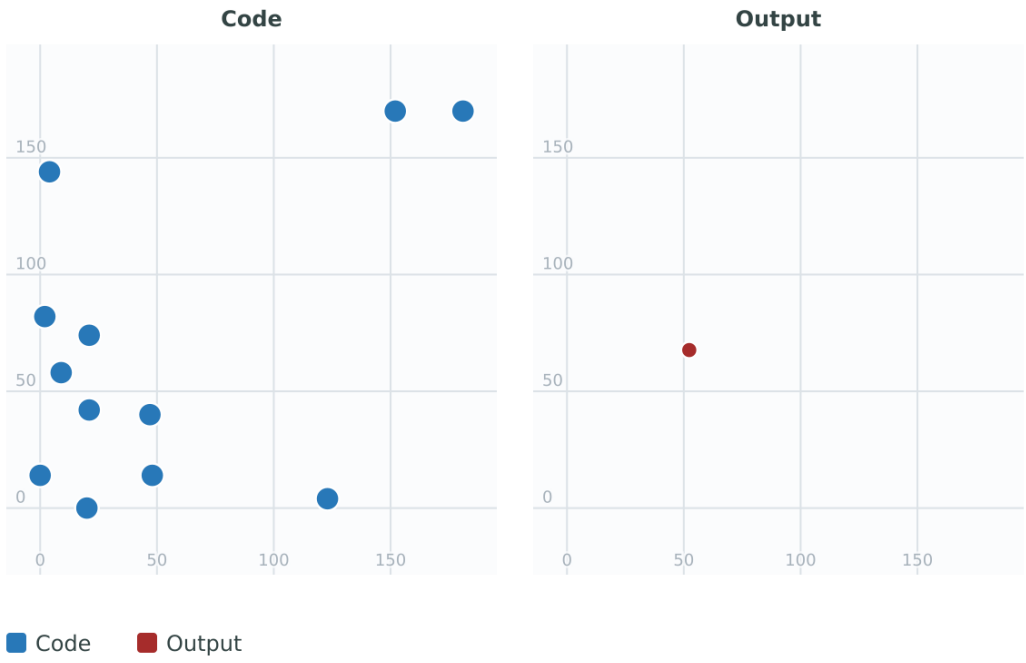
Code

```
SELECT ST_Centroid(
  'MULTIPOINT (8 24,10 92,12 154,17 68,28 10,29 52,29 84,55 50,56 24,131 14,160 180,189 180) ↩
    ');
```

Output

```
POINT(60.333 77.667)
```

Figure



Centroid of a LineString.

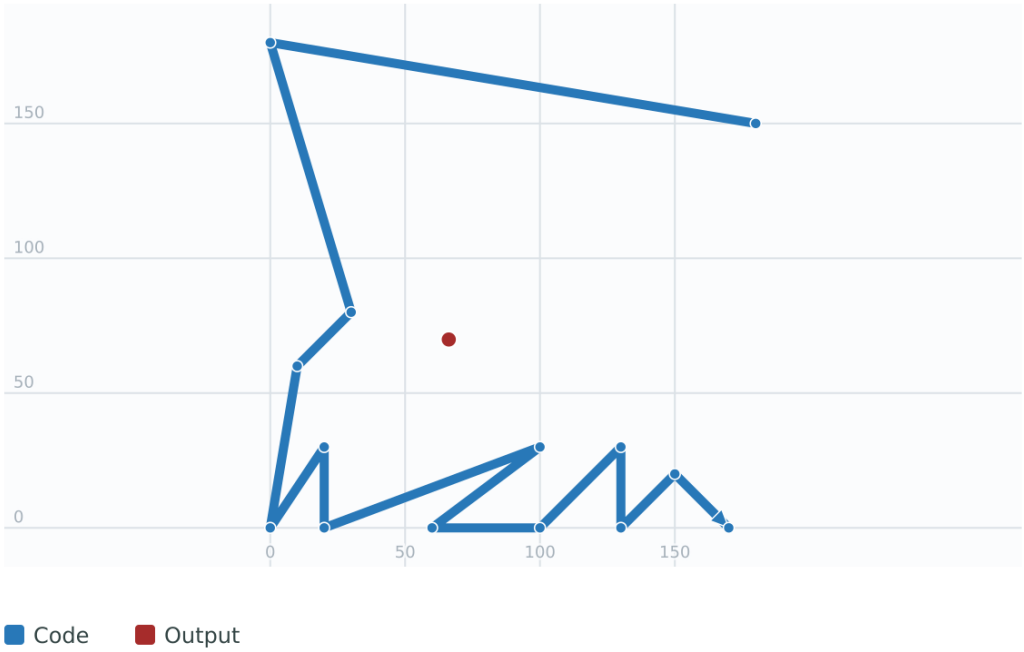
Code

```
SELECT ST_Centroid(
  'LINESTRING (190 160,10 190,40 90,20 70,10 10,30 40,30 10,110 40,70 10,110 10,140 40,140  ↩
    10,160 30,180 10)');
```

Output

```
POINT(76.191 79.876)
```

Figure



Centroid of a Polygon.

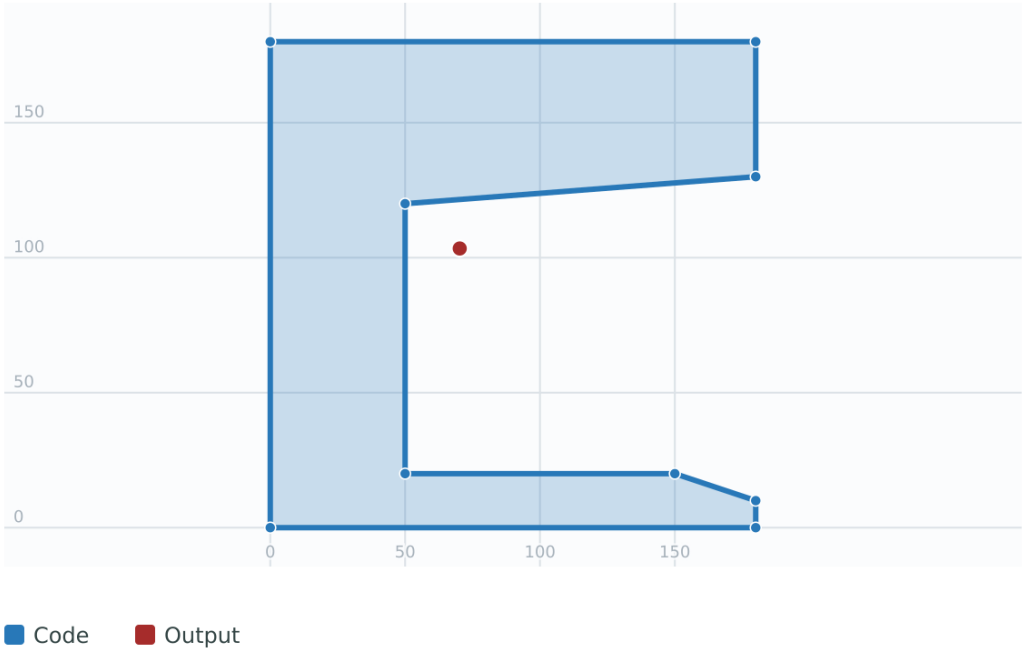
Code

```
SELECT ST_Centroid(
  'POLYGON ((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))');
```

Output

```
POINT(80.251 113.405)
```

Figure



Centroid of the highest-dimension components of a GeometryCollection.

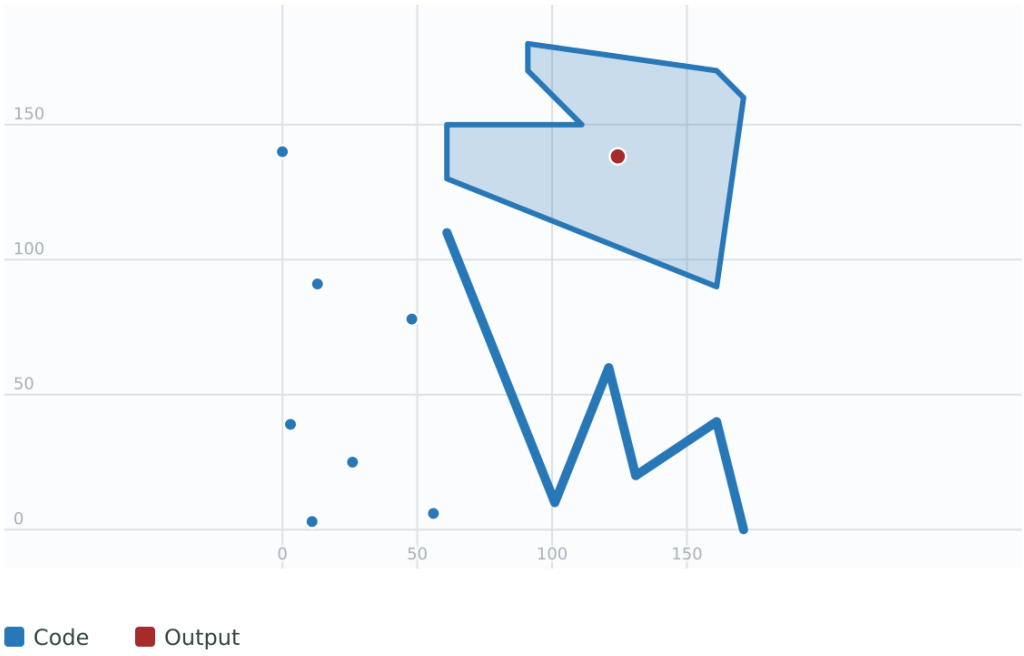
Code

```
SELECT ST_Centroid(
  'GEOMETRYCOLLECTION (POLYGON ((190 170,180 100,80 140,80 160,130 160,110 180,110 190,180 ↩
   180,190 170)),
    LINESTRING (80 120,120 20,140 70,150 30,180 50,190 10),
    MULTIPOINT (19 150,22 49,30 13,32 101,45 35,67 88,75 16))');
```

Output

```
POINT(143.361 148.263)
```

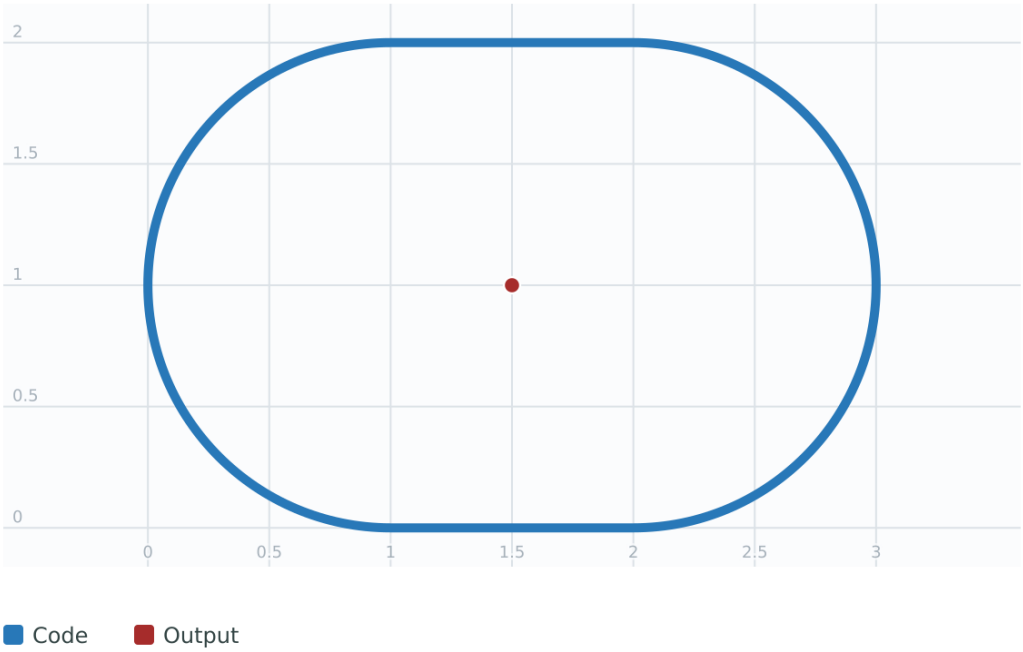
Figure



Code

```
SELECT ST_Centroid(g)
FROM  ST_GeomFromText('CIRCULARSTRING(0 2,-1 1,0 0,0.5 0,1 0,2 1,1 2,0.5 2,0 2)')  AS g ;
```

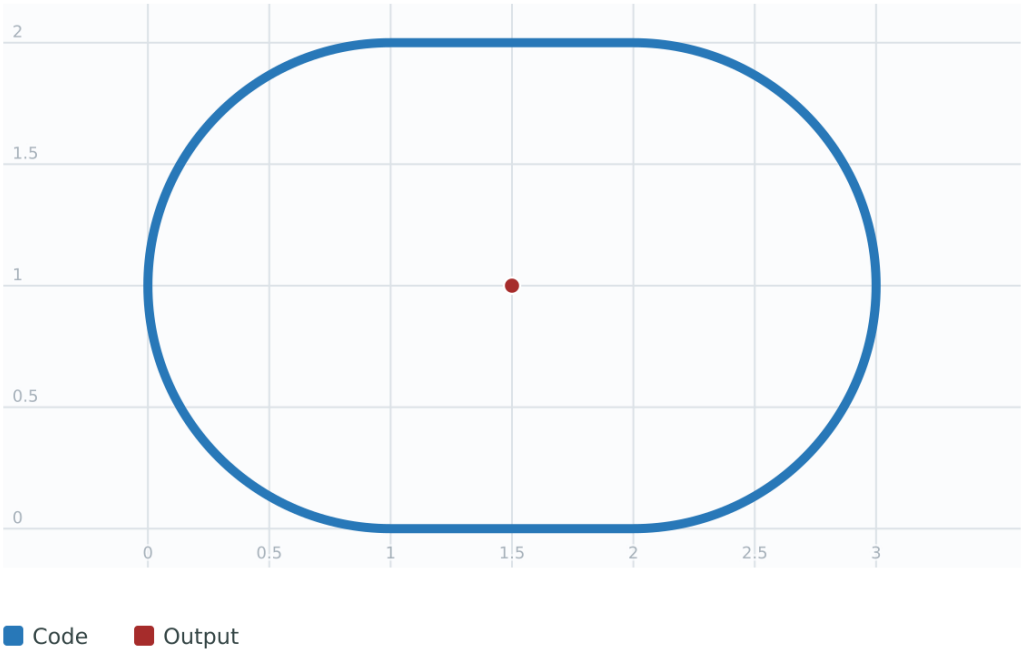
Figure



Code

```
SELECT ST_Centroid(g)
FROM  ST_GeomFromText('COMPOUNDCURVE(CIRCULARSTRING(0 2,-1 1,0 0),(0 0,0.5 0,1 0),  ↩
    CIRCULARSTRING( 1 0,2 1,1 2),(1 2,0.5 2,0 2))' ) AS g;
```

Figure



Centroids of the parts of a MultiPolygon.

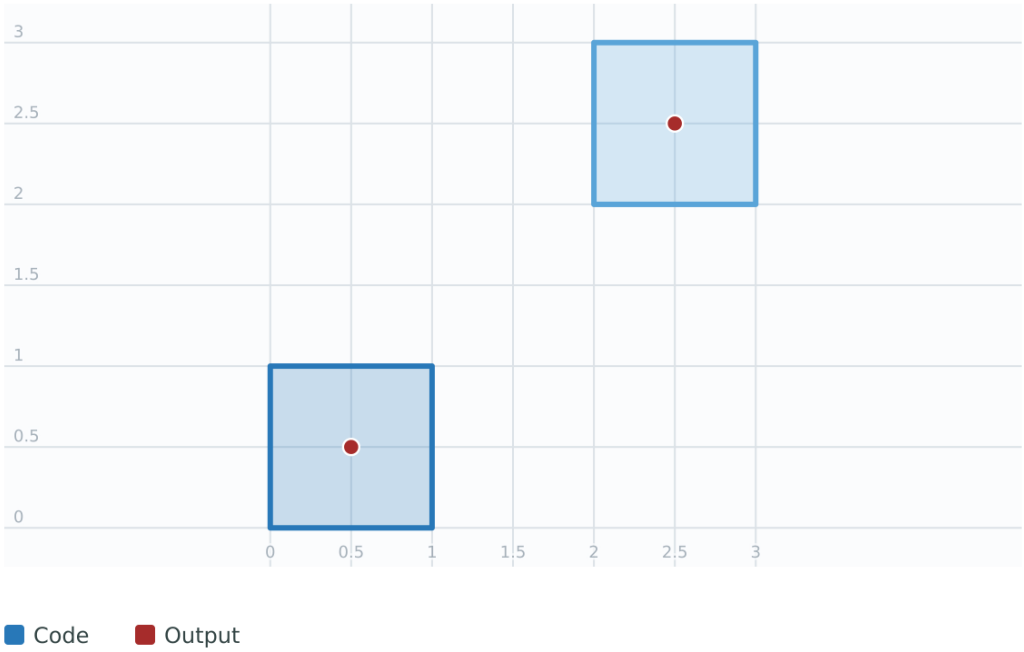
Code

```
SELECT ST_Collect(ST_Centroid(geom) ORDER BY path)
FROM ST_Dump('MULTIPOLYGON (
    ((0 0,0 1,1 1,1 0,0 0)),
    ((2 2,2 3,3 3,3 2,2 2))
  )'::geometry);
```

Output

```
MULTIPOINT((0.5 0.5),(2.5 2.5))
```

Figure



See Also

ST_Collect, ST_Dump, ST_PointOnSurface, ST_GeometricMedian

7.14.4 ST_ChaikinSmoothing

ST_ChaikinSmoothing — Returns a smoothed version of a geometry, using the Chaikin algorithm

Synopsis

geometry **ST_ChaikinSmoothing**(geometry geom, integer nIterations = 1, boolean preserveEndPoints = false);

Description

Smoothes a linear or polygonal geometry using Chaikin's algorithm. The degree of smoothing is controlled by the `nIterations` parameter. On each iteration, each interior vertex is replaced by two vertices located at 1/4 of the length of the line segments before and after the vertex. A reasonable degree of smoothing is provided by 3 iterations; the maximum is limited to 5.

If `preserveEndPoints` is true, the endpoints of Polygon rings are not smoothed. The endpoints of LineStrings are always preserved.



Note
The number of vertices doubles with each iteration, so the result geometry may have many more points than the input. To reduce the number of points use a simplification function on the result (see ST_Simplify, ST_SimplifyPreserveTopology and ST_SimplifyVW).

The result has interpolated values for the Z and M dimensions when present.

 This function supports 3d and will not drop the z-index.

Availability: 2.5.0

Examples

Smoothing a Polygon using 1, 2 and 3 iterations.

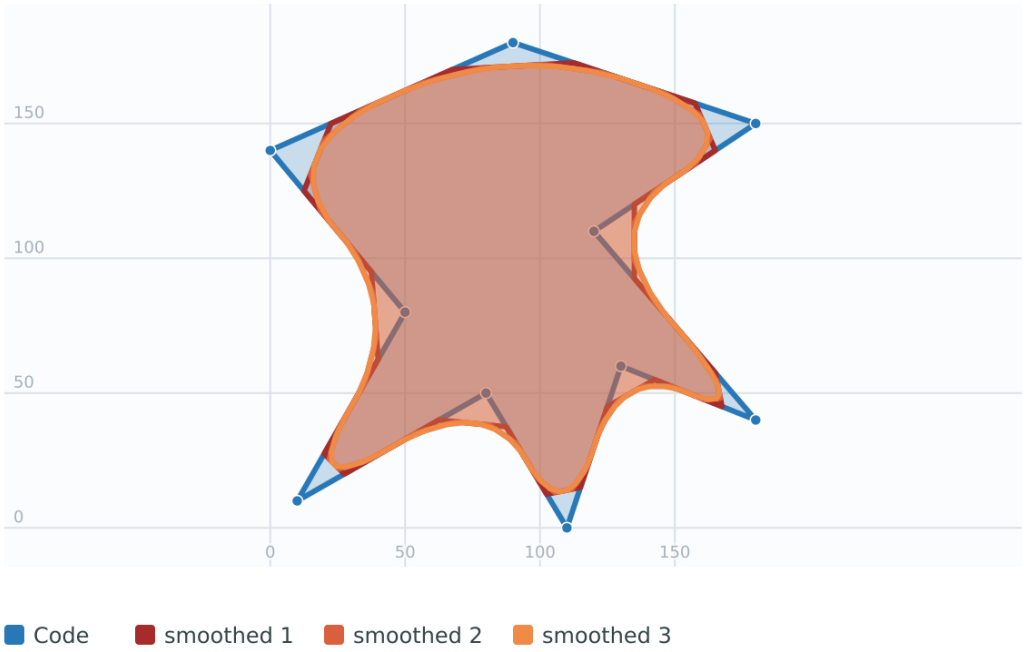
Code

```
SELECT n AS iterations, ST_ChaikinSmoothing(
    'POLYGON ((20 20,60 90,10 150,100 190,190 160,130 120,190 50,140 70,120 10,90 60,20 20))',
    n
  ) AS smoothed
FROM generate_series(1, 3) AS n;
```

Output

```
1 | POLYGON((30 37.5,50 72.5,47.5 105,22.5 135,32.5 160,77.5 180,122.5 182.5,167.5 167.5,175 150,145 130,145 102.5,175 67.5,177.5 55,152.5 65,135 55,125 25,112.5 22.5,97.5 47.5,72.5 50,37.5 30,30 37.5))
2 | POLYGON((35 46.25,45 63.75,49.375 80.625,48.125 96.875,41.25 112.5,28.75 127.5,25 141.25,30 153.75,43.75 165,66.25 175,88.75 180.625,111.25 181.875,133.75 178.75,156.25 171.25,169.375 163.125,173.125 154.375,167.5 145,152.5 135,145 123.125,145 109.375,152.5 93.75,167.5 76.25,175.625 64.375,176.875 58.125,171.25 57.5,158.75 62.5,148.125 62.5,139.375 57.5,132.5 47.5,127.5 32.5,121.875 24.375,115.625 23.125,108.75 28.75,101.25 41.25,91.25 48.125,78.75 49.375,63.75 45,46.25 35,35.625 31.875,31.875 35.625,35 46.25))
3 | POLYGON((37.5 50.625,42.5 59.375,46.09375 67.96875,48.28125 76.40625,49.0625 84.6875,48.4375 92.8125,46.40625 100.78125,42.96875 108.59375,38.125 116.25,31.875 123.75,27.8125 130.9375,25.9375 137.8125,26.25 144.375,28.75 150.625,33.4375 156.5625,40.3125 162.1875,49.375 167.5,60.625 172.5,71.875 176.40625,83.125 179.21875,94.375 180.9375,105.625 181.5625,116.875 181.09375,128.125 179.53125,139.375 176.875,150.625 173.125,159.53125 169.21875,166.09375 165.15625,170.3125 160.9375,172.1875 156.5625,171.71875 152.03125,168.90625 147.34375,163.75 142.5,156.25 137.5,150.625 132.03125,146.875 126.09375,145 119.6875,145 112.8125,146.875 105.46875,150.625 97.65625,156.25 89.375,163.75 80.625,169.53125 73.28125,173.59375 67.34375,175.9375 62.8125,176.5625 59.6875,175.46875 57.96875,172.65625 57.65625,168.125 58.75,161.875 61.25,156.09375 62.5,150.78125 62.5,145.9375 61.25,141.5625 58.75,137.65625 55,134.21875 50,131.25 43.75,128.75 36.25,126.09375 30.46875,123.28125 26.40625,120.3125 24.0625,117.1875 23.4375,113.90625 24.53125,110.46875 27.34375,106.875 31.875,103.125 38.125,98.75 42.96875,93.75 46.40625,88.125 48.4375,81.875 49.0625,75 48.28125,67.5 46.09375,59.375 42.5,50.625 37.5,43.59375 34.21875,38.28125 32.65625,34.6875 32.8125,32.8125 34.6875,32.65625 38.28125,34.21875 43.59375,37.5 50.625))
```

Figure



Smoothing a LineString using 1, 2 and 3 iterations.

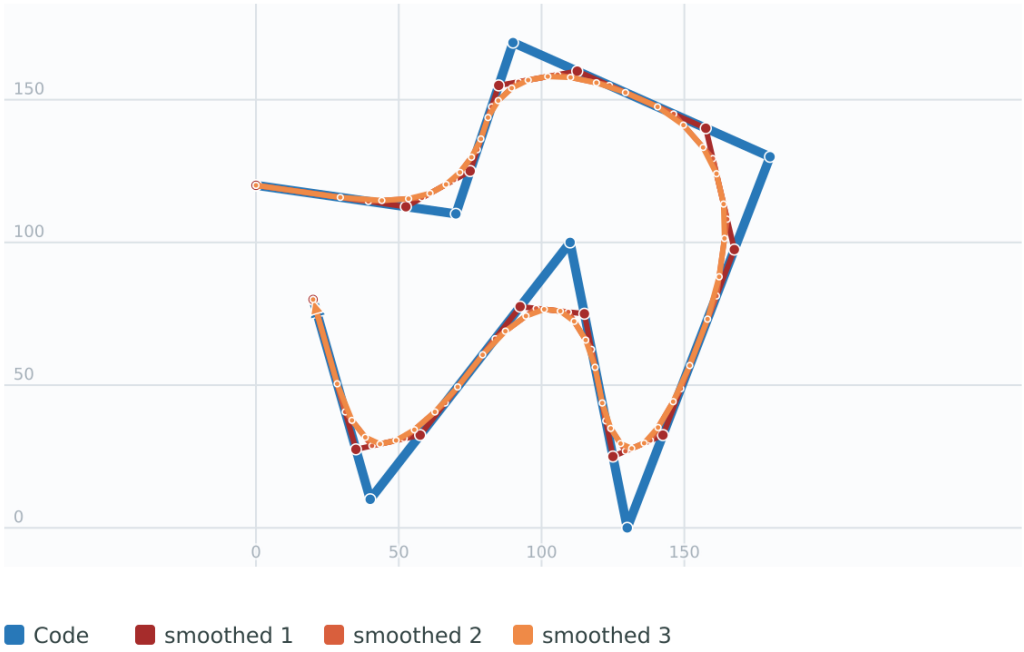
Code

```
SELECT n AS iterations, ST_ChaikinSmoothing(
    'LINESTRING (10 140,80 130,100 190,190 150,140 20,120 120,50 30,30 100)', n
  ) AS smoothed
FROM generate_series(1, 3) AS n;
```

Output

```
1 | LINESTRING(10 140,62.5 132.5,85 145,95 175,122.5 180,167.5 160,177.5 117.5,152.5 52.5,135 45,125 95,102.5 97.5,67.5 ↩
     52.5,45 47.5,30 100)
2 | LINESTRING(10 140,49.375 134.375,68.125 135.625,79.375 141.875,87.5 152.5,92.5 167.5,101.875 176.25,115.625  ↩
    178.75,133.75 175,156.25 165,170 149.375,175 128.125,171.25 101.25,158.75 68.75,148.125 50.625,139.375  ↩
    46.875,132.5 57.5,127.5 82.5,119.375 95.625,108.125 96.875,93.75 86.25,76.25 63.75,61.875 51.25,50.625 48.75,41.25 ↩
     60.625,30 100)
3 | LINESTRING(10 140,39.53125 135.78125,54.0625 134.6875,63.4375 135.3125,70.9375 137.1875,76.5625 140.3125,81.40625  ↩
    144.53125,85.46875 149.84375,88.75 156.25,91.25 163.75,94.84375 169.6875,99.53125 174.0625,105.3125  ↩
    176.875,112.1875 178.125,120.15625 177.8125,129.21875 175.9375,139.375 172.5,150.625 167.5,159.6875  ↩
    161.09375,166.5625 153.28125,171.25 144.0625,173.75 133.4375,174.0625 121.40625,172.1875 107.96875,168.125  ↩
    93.125,161.875 76.875,156.09375 64.21875,150.78125 55.15625,145.9375 49.6875,141.5625 47.8125,137.65625  ↩
    49.53125,134.21875 54.84375,131.25 63.75,128.75 76.25,125.46875 85.78125,121.40625 92.34375,116.5625  ↩
    95.9375,110.9375 96.5625,104.53125 94.21875,97.34375 88.90625,89.375 80.625,80.625 69.375,72.65625 60.625,65.46875 ↩
     54.375,59.0625 50.625,53.4375 49.375,48.28125 51.71875,43.59375 57.65625,38.4375 70.46875,30 100)
```

Figure



Smoothing a triangle shows the weakest visual change, so it is kept last.

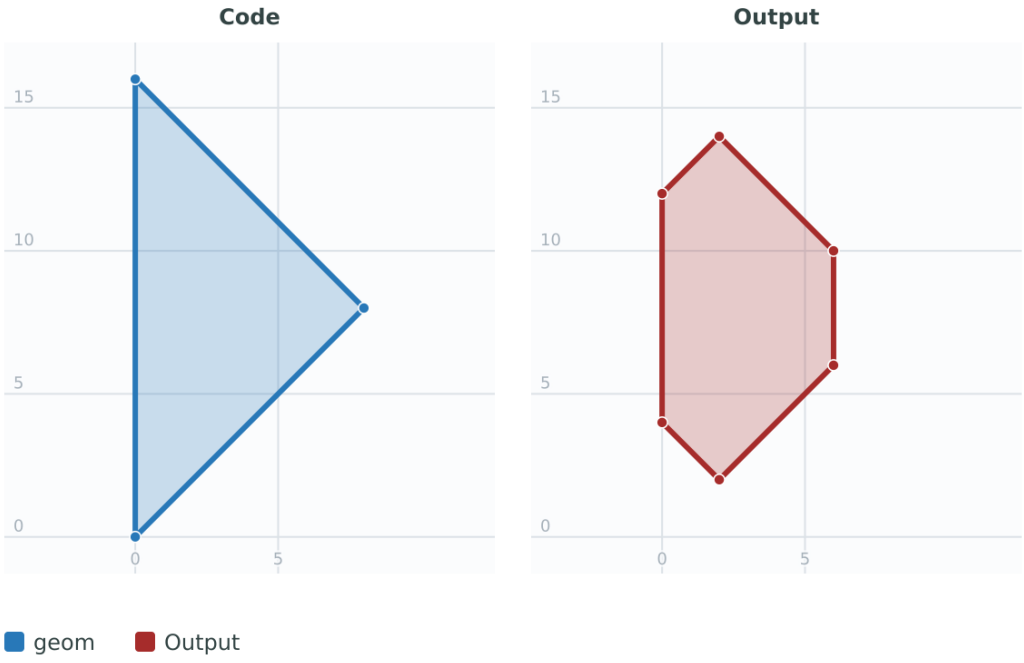
Code

```
SELECT ST_ChaikinSmoothing(geom) smoothed
FROM (SELECT  'POLYGON((0 0,8 8,0 16,0 0))'::geometry geom) AS foo;
```

Output

```
POLYGON((2 2,6 6,6 10,2 14,0 12,0 4,2 2))
```

Figure



See Also

ST_Simplify, ST_SimplifyPreserveTopology, ST_SimplifyVW

7.14.5 ST_CatmullRomSmoothing

ST_CatmullRomSmoothing — Returns a smoothed version of a geometry, using the Catmull-Rom spline algorithm

Synopsis

geometry **ST_CatmullRomSmoothing**(geometry geom, integer nSegments = 5);

Description

Smoothes a linear or polygonal geometry using the Catmull-Rom spline algorithm. Unlike ST_ChaikinSmoothing, this is an *interpolating* spline: the output curve passes through every original vertex. Between each pair of consecutive original vertices, `nSegments` - 1 new intermediate points are inserted.

The `nSegments` parameter controls the density of the output. With `nSegments` = 5 (the default), each original span is divided into 5 sub-segments (inserting 4 new points per span). The minimum value is 2.

At least 4 input vertices are required to apply smoothing; geometries with fewer vertices are returned unchanged. Points and multipoints are always returned unchanged.



Note
The output vertex count grows as `1 + (N-1) * nSegments` for open lines, and `1 + N * nSegments` for closed rings, where N is the number of input vertices. For large geometries, use a simplification function on the result to reduce the number of points (see ST_Simplify, ST_SimplifyPreserveTopology and ST_SimplifyVW).

The result has interpolated values for the Z and M dimensions when present.

 This function supports 3d and will not drop the z-index.

Availability: 3.7.0

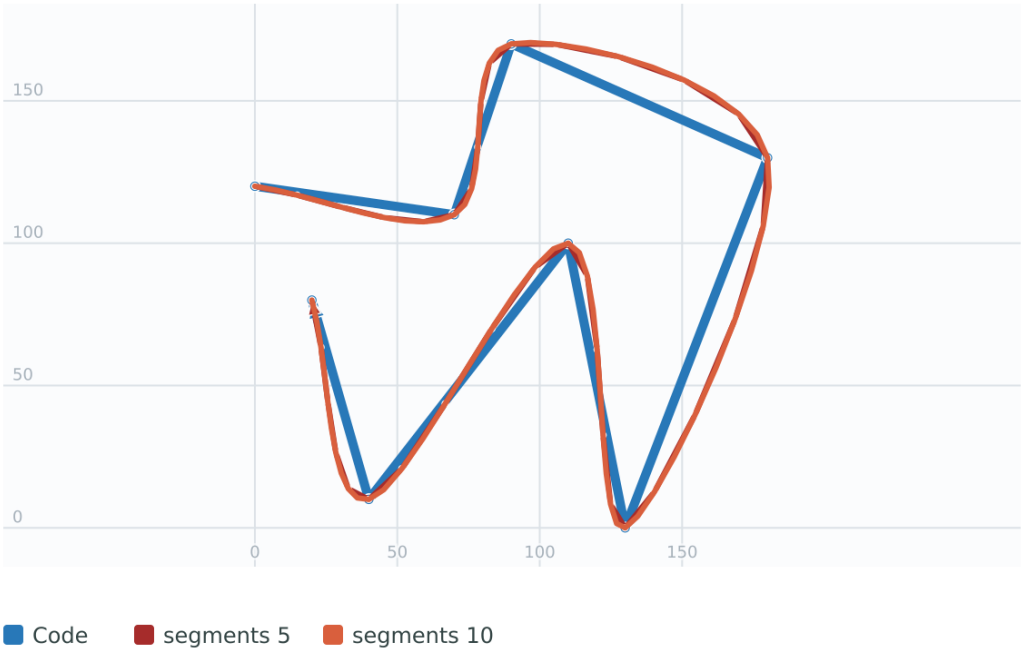
Examples

Smoothing a LineString using nSegments = 5 and 10.

Code

```
WITH input(geom) AS (VALUES
  ('LINESTRING (10 140,80 130,100 190,190 150,140 20,120 120,50 30,30 100)'::geometry)
)
SELECT ST_CatmullRomSmoothing(geom, 5) AS segments_5,
  ST_CatmullRomSmoothing(geom, 10) AS segments_10
FROM input;
```

Figure



Smoothing a 4-point collinear line with default nSegments=5 shows the minimal change case.

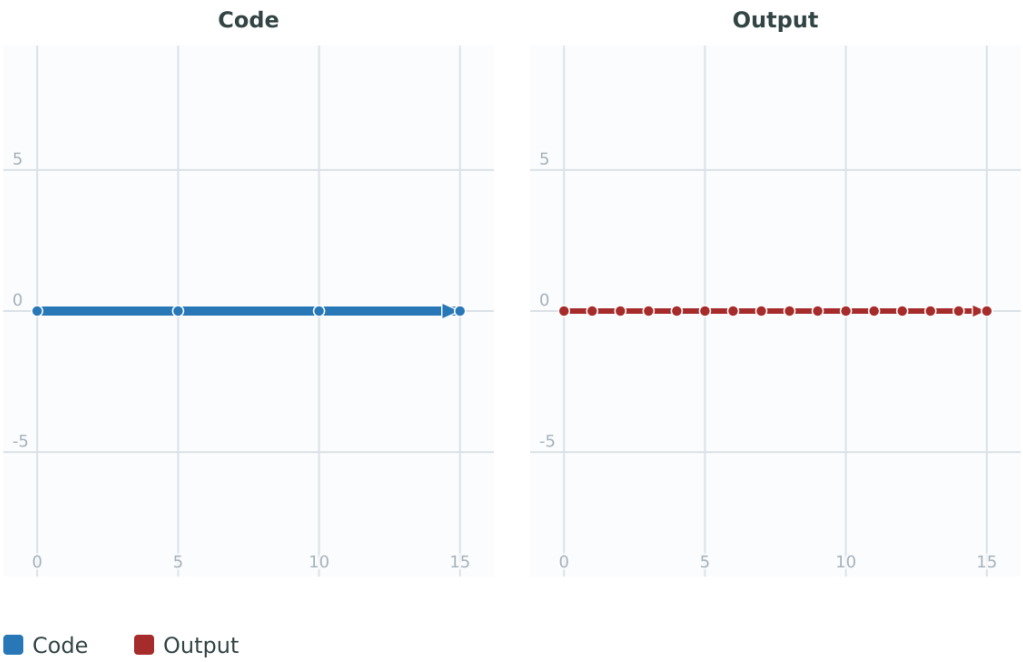
Code

```
SELECT ST_CatmullRomSmoothing('LINESTRING(0 0,5 0,10 0,15 0)');
```

Output

```
LINESTRING(0 0,1 0,2 0,3 0,4 0,5 0,6 0,7 0,8 0,9 0,10 0,11 0,12 0,13 0,14 0,15 0)
```

Figure

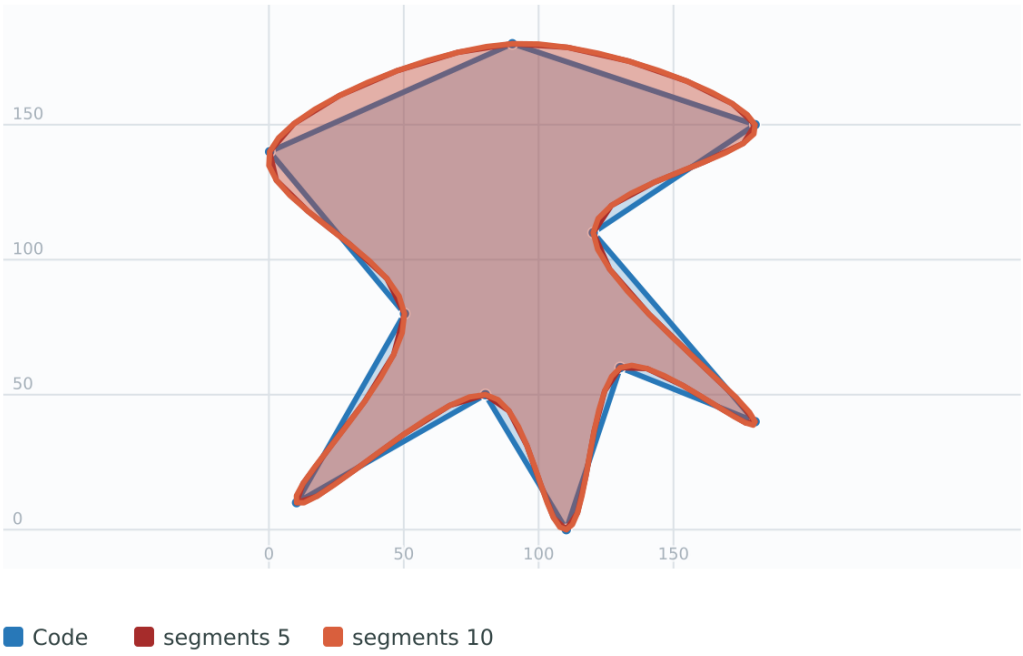


Smoothing a Polygon using nSegments = 5 and 10. This larger example is kept last.

Code

```
WITH input(geom) AS (VALUES
  ('POLYGON ((20 20,60 90,10 150,100 190,190 160,130 120,190 50,140 70,120 10,90 60,20 20))' ↩
    ::geometry)
)
SELECT ST_CatmullRomSmoothing(geom, 5) AS segments_5,
  ST_CatmullRomSmoothing(geom, 10) AS segments_10
FROM input;
```

Figure



See Also

ST_ChaikinSmoothing, ST_Simplify, ST_SimplifyPreserveTopology, ST_SimplifyVW

7.14.6 ST_ConcaveHull

ST_ConcaveHull — Computes a possibly concave geometry that contains all input geometry vertices

Synopsis

geometry **ST_ConcaveHull**(geometry param_geom, float param_pctconvex, boolean param_allow_holes = false);

Description

A concave hull is a (usually) concave geometry which contains the input, and whose vertices are a subset of the input vertices. In the general case the concave hull is a Polygon. The concave hull of two or more collinear points is a two-point LineString. The concave hull of one or more identical points is a Point. The polygon will not contain holes unless the optional `param_allow_holes` argument is specified as true.

One can think of a concave hull as "shrink-wrapping" a set of points. This is different to the convex hull, which is more like wrapping a rubber band around the points. A concave hull generally has a smaller area and represents a more natural boundary for the input points.

The `param_pctconvex` controls the concaveness of the computed hull. A value of 1 produces the convex hull. Values between 1 and 0 produce hulls of increasing concaveness. A value of 0 produces a hull with maximum concaveness (but still a single polygon). Choosing a suitable value depends on the nature of the input data, but often values between 0.3 and 0.1 produce reasonable results.



Note
Technically, the `param_pctconvex` determines a length as a fraction of the difference between the longest and shortest edges in the Delaunay Triangulation of the input points. Edges longer than this length are "eroded" from the triangulation. The triangles remaining form the concave hull.

For point and linear inputs, the hull will enclose all the points of the inputs. For polygonal inputs, the hull will enclose all the points of the input *and also* all the areas covered by the input. If you want a point-wise hull of a polygonal input, convert it to points first using ST_Points.

This is not an aggregate function. To compute the concave hull of a set of geometries use ST_Collect (e.g. `ST_ConcaveHull(ST_Col ), 0.80)`.

Availability: 2.0.0

Enhanced: 3.3.0, GEOS native implementation enabled for GEOS 3.11+

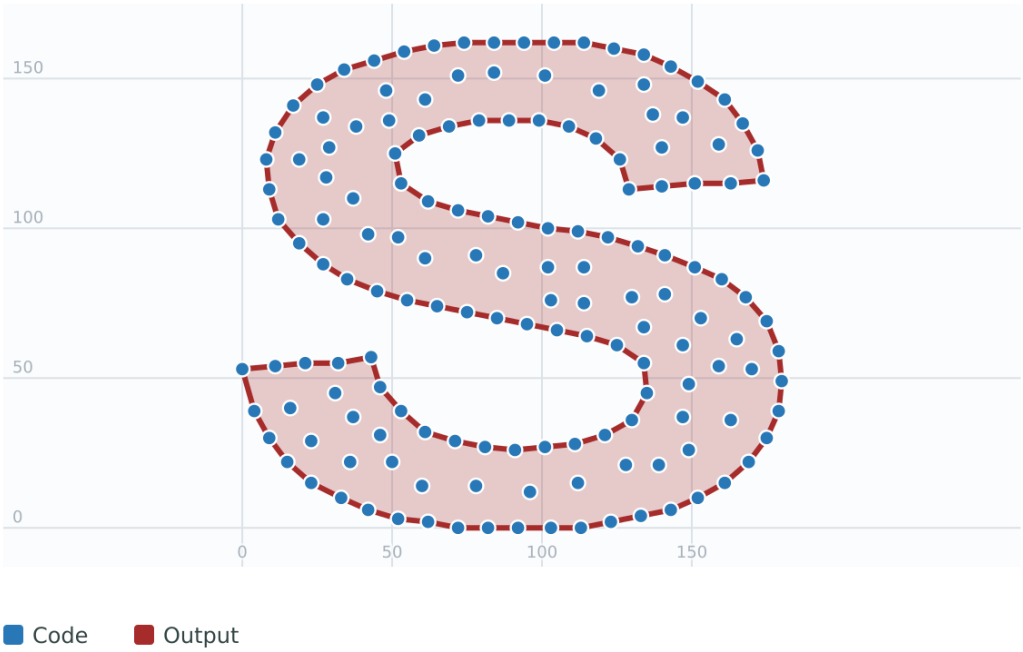
Examples

Concave hull of a MultiPoint.

Code

```
SELECT ST_ConcaveHull('MULTIPOINT ((10 72),(53 76),(56 66),(63 58),(71 51),(81 48),(91 46) ↩
    ,(101 45),(111 46),(121 47),(131 50),(140 55),(145 64),(144 74),(135 80),(125 83),(115  ↩
    85),(105 87),(95 89),(85 91),(75 93),(65 95),(55 98),(45 102),(37 107),(29 114),(22 122) ↩
    ,(19 132),(18 142),(21 151),(27 160),(35 167),(44 172),(54 175),(64 178),(74 180),(84  ↩
    181),(94 181),(104 181),(114 181),(124 181),(134 179),(144 177),(153 173),(162 168),(171 ↩
     162),(177 154),(182 145),(184 135),(139 132),(136 142),(128 149),(119 153),(109 155) ↩
    ,(99 155),(89 155),(79 153),(69 150),(61 144),(63 134),(72 128),(82 125),(92 123),(102  ↩
    121),(112 119),(122 118),(132 116),(142 113),(151 110),(161 106),(170 102),(178 96),(185 ↩
     88),(189 78),(190 68),(189 58),(185 49),(179 41),(171 34),(162 29),(153 25),(143 23) ↩
    ,(133 21),(123 19),(113 19),(102 19),(92 19),(82 19),(72 21),(62 22),(52 25),(43 29),(33 ↩
     34),(25 41),(19 49),(14 58),(21 73),(31 74),(42 74),(173 134),(161 134),(150 133),(97  ↩
    104),(52 117),(157 156),(94 171),(112 106),(169 73),(58 165),(149 40),(70 33),(147 157) ↩
    ,(48 153),(140 96),(47 129),(173 55),(144 86),(159 67),(150 146),(38 136),(111 170),(124 ↩
     94),(26 59),(60 41),(71 162),(41 64),(88 110),(122 34),(151 97),(157 56),(39 146),(88  ↩
    33),(159 45),(47 56),(138 40),(129 165),(33 48),(106 31),(169 147),(37 122),(71 109) ↩
    ,(163 89),(37 156),(82 170),(180 72),(29 142),(46 41),(59 155),(124 106),(157 80),(175  ↩
    82),(56 50),(62 116),(113 95),(144 167))',
         0.1 );
```

Figure

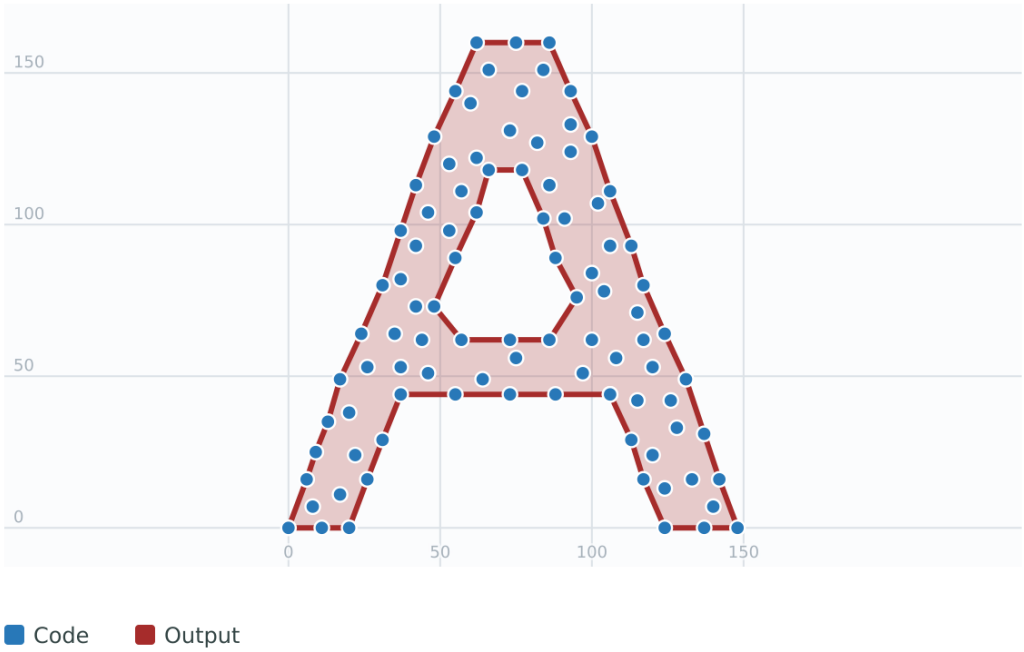


Concave hull of a MultiPoint, allowing holes.

Code

```
SELECT ST_ConcaveHull('MULTIPOINT ((132 64),(114 64),(99 64),(81 64),(63 64),(57 49),(52 36) ↩
    ,(46 20),(37 20),(26 20),(32 36),(39 55),(43 69),(50 84),(57 100),(63 118),(68 133),(74  ↩
    149),(81 164),(88 180),(101 180),(112 180),(119 164),(126 149),(132 131),(139 113),(143  ↩
    100),(150 84),(157 69),(163 51),(168 36),(174 20),(163 20),(150 20),(143 36),(139 49) ↩
    ,(132 64),(99 151),(92 138),(88 124),(81 109),(74 93),(70 82),(83 82),(99 82),(112 82) ↩
    ,(126 82),(121 96),(114 109),(110 122),(103 138),(99 151),(34 27),(43 31),(48 44),(46  ↩
    58),(52 73),(63 73),(61 84),(72 71),(90 69),(101 76),(123 71),(141 62),(166 27),(150 33) ↩
    ,(159 36),(146 44),(154 53),(152 62),(146 73),(134 76),(143 82),(141 91),(130 98),(126  ↩
    104),(132 113),(128 127),(117 122),(112 133),(119 144),(108 147),(119 153),(110 171) ↩
    ,(103 164),(92 171),(86 160),(88 142),(79 140),(72 124),(83 131),(79 118),(68 113),(63  ↩
    102),(68 93),(35 45))',
         0.15, true );
```

Figure



Comparing a concave hull of a Polygon to the concave hull of the constituent points. The hull respects the boundary of the polygon, whereas the points-based hull does not.

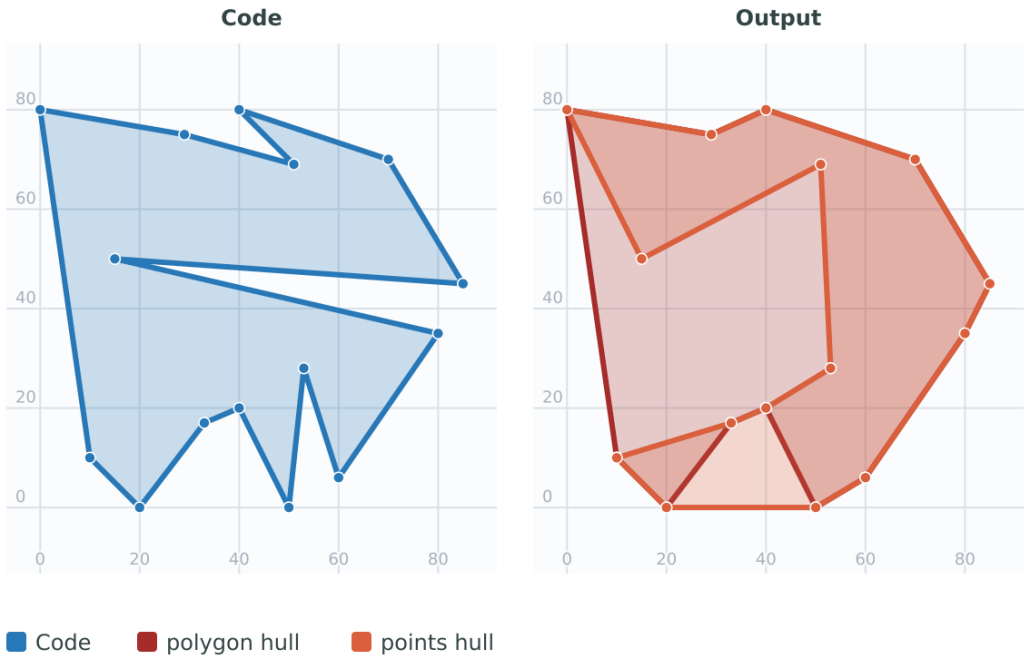
Code

```
WITH data(geom) AS (VALUES
    ('POLYGON ((10 90,39 85,61 79,50 90,80 80,95 55,25 60,90 45,70 16,63 38,60 10,50 30,43 ↩
     27,30 10,20 20,10 90))'::geometry)
)
SELECT ST_ConcaveHull(geom, 0.1) AS polygon_hull,
       ST_ConcaveHull(ST_Points(geom), 0.1) AS points_hull
    FROM data;
```

Output

```
POLYGON((60 10,50 30,43 27,30 10,20 20,10 90,39 85,50 90,80 80,95 55,90 45,70 16,60 10)) | POLYGON((25 60,10 90,39 ↩
     85,50 90,80 80,95 55,90 45,70 16,60 10,30 10,20 20,43 27,50 30,63 38,61 79,25 60))
```

Figure



Using with ST_Collect to compute the concave hull of a geometry set.

This example estimates an infected area from point observations.

Code

```
SELECT disease_type,
    ST_ConcaveHull(ST_Collect(obs_pnt), 0.3 ) AS geom
  FROM disease_obs
  GROUP BY disease_type;
```

See Also

ST_ConvexHull, ST_Collect, CG_AlphaShape, CG_OptimalAlphaShape

7.14.7 ST_ConvexHull

ST_ConvexHull — Computes the convex hull of a geometry.

Synopsis

geometry **ST_ConvexHull**(geometry geomA);

Description

Computes the convex hull of a geometry. The convex hull is the smallest convex geometry that encloses all geometries in the input.

One can think of the convex hull as the geometry obtained by wrapping an rubber band around a set of geometries. This is different from a concave hull which is analogous to "shrink-wrapping" the geometries. A convex hull is often used to determine an affected area based on a set of point observations.

In the general case the convex hull is a Polygon. The convex hull of two or more collinear points is a two-point LineString. The convex hull of one or more identical points is a Point.

This is not an aggregate function. To compute the convex hull of a set of geometries, use ST_Collect to aggregate them into a geometry collection (e.g. `ST_ConvexHull(ST_Collect(geom))`.

Performed by the GEOS module

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.16

This function supports 3d and will not drop the z-index.

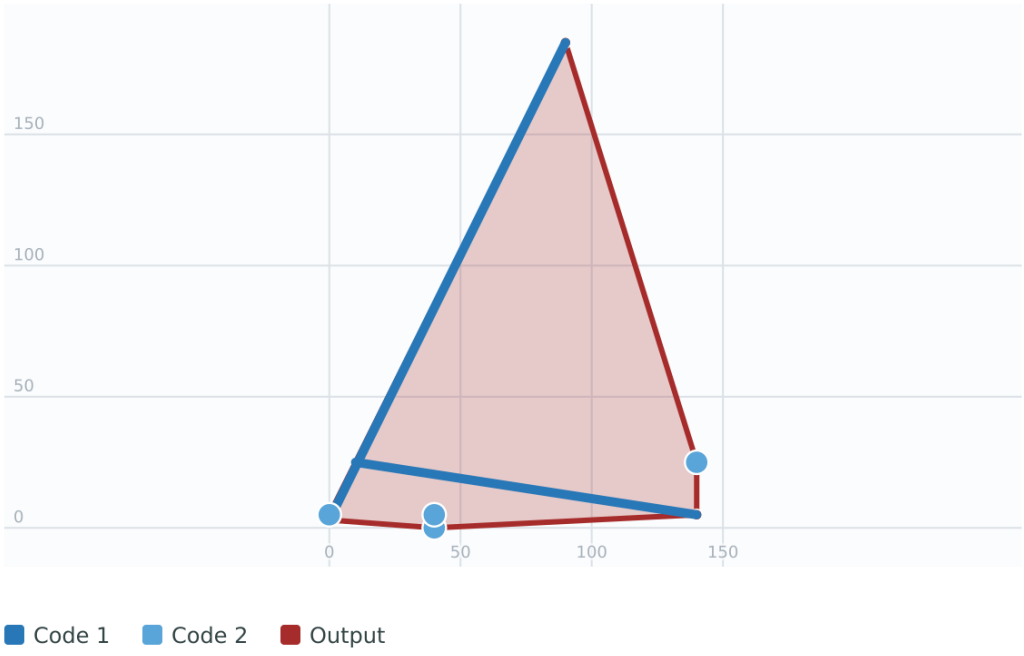
Examples

Compute the convex hull of a MultiLineString and a MultiPoint.

Code

```
SELECT ST_ConvexHull(ST_Collect(ST_GeomFromText('MULTILINESTRING((100 190,10 8),(150 10,20 ↩
    30))'),
            ST_GeomFromText('MULTIPOINT(50 5,150 30,50 10,10 10)')
            ));
```

Figure



Using with ST_Collect to compute the convex hulls of geometry sets.

This example returns an estimated infected area from point observations.

Code

```
SELECT d.disease_type,
    ST_ConvexHull(ST_Collect(d.geom)) As geom
    FROM disease_obs As d
    GROUP BY d.disease_type;
```

See Also

ST_Collect, ST_ConcaveHull, ST_MinimumBoundingCircle

7.14.8 ST_DelaunayTriangles

ST_DelaunayTriangles — Returns the Delaunay triangulation of the vertices of a geometry.

Synopsis

geometry **ST_DelaunayTriangles**(geometry g1, float tolerance = 0.0, int4 flags = 0);

Description

Computes the Delaunay triangulation of the vertices of the input geometry. The optional `tolerance` can be used to snap nearby input vertices together, which improves robustness in some situations. The result geometry is bounded by the convex hull of the input vertices. The result geometry representation is determined by the `flags` code:

- `0` - a GEOMETRYCOLLECTION of triangular POLYGONs (default)
- `1` - a MULTILINESTRING of the edges of the triangulation
- `2` - A TIN of the triangulation

Performed by the GEOS module.

Availability: 2.1.0

 This function supports 3d and will not drop the z-index.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

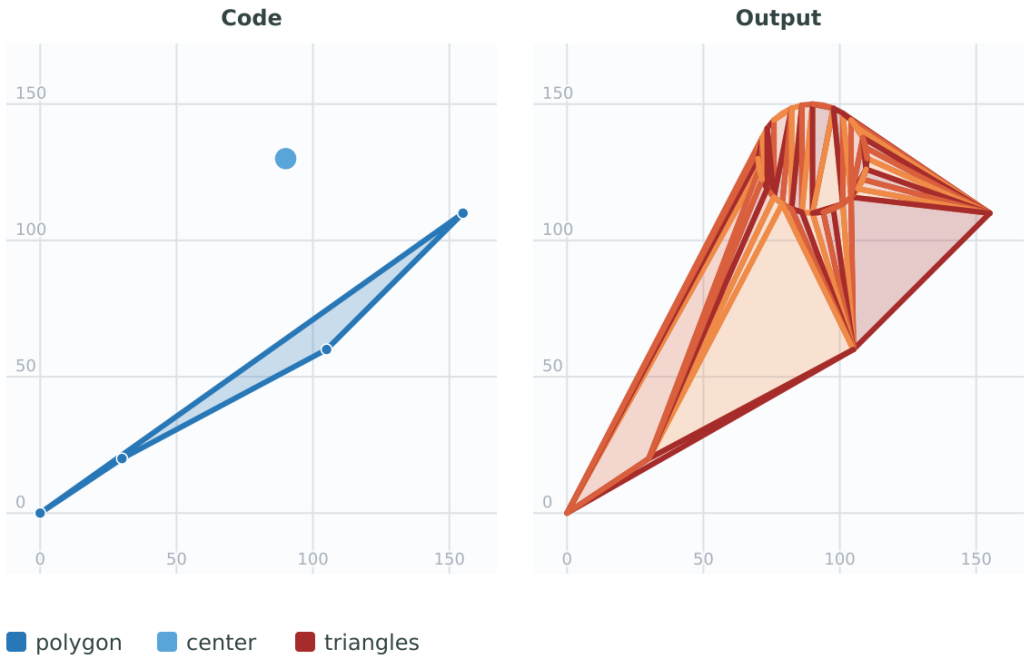
Examples

Triangulation of the vertices of a polygon and a buffered point.

Code

```
WITH data AS (
  SELECT polygon, ST_Buffer(center, 20) AS point_buffer
  FROM (SELECT
    'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry AS polygon,
    'POINT(110 170)'::geometry AS center
  ) AS input
)
SELECT ST_DelaunayTriangles(ST_Union(polygon, point_buffer)) AS triangles
FROM data;
```

Figure

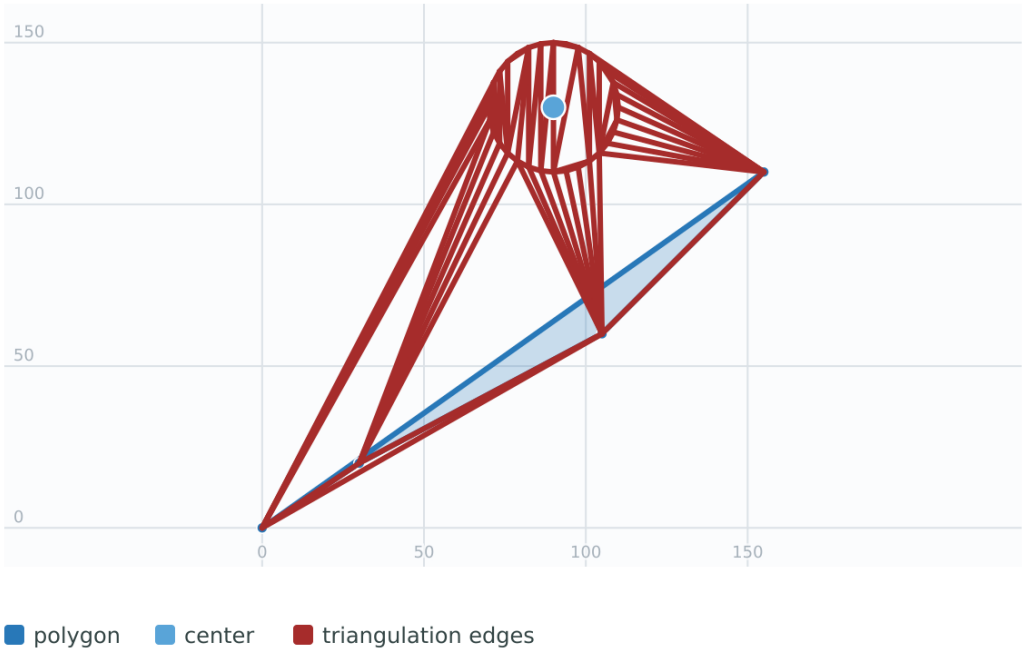


Triangulation edges returned as a `MULTILINESTRING`.

Code

```
WITH data AS (
  SELECT polygon, ST_Buffer(center, 20) AS point_buffer
  FROM (SELECT
    'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry AS polygon,
    'POINT(110 170)'::geometry AS center
  ) AS input
)
SELECT ST_DelaunayTriangles(ST_Union(polygon, point_buffer), 0.001, 1) AS ↩
    triangulation_edges
FROM data;
```

Figure

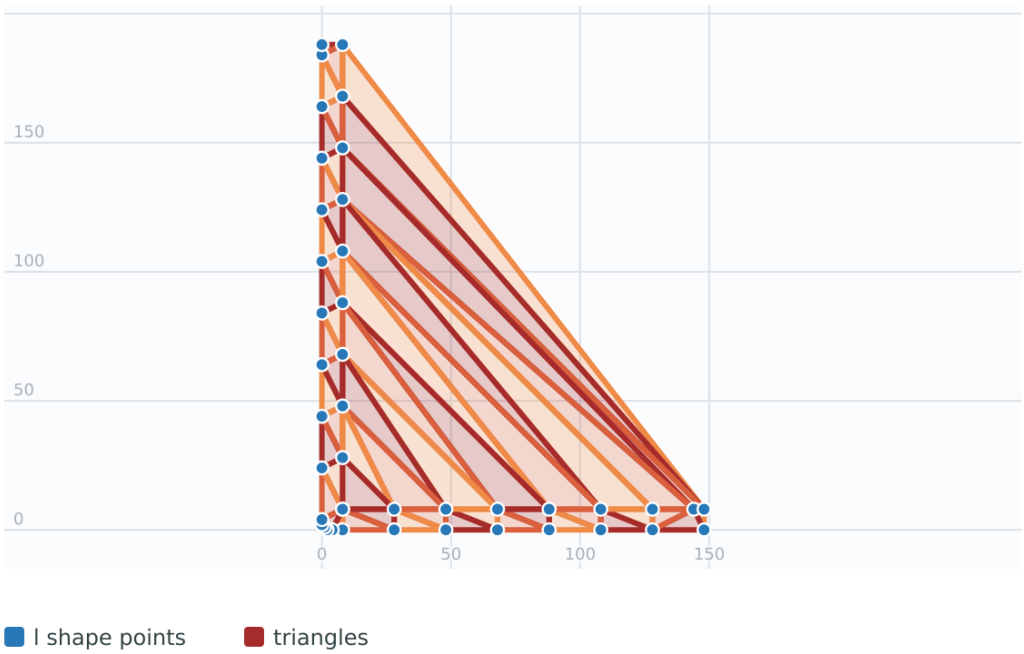


Triangulation of 42 points forming an L shape.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT(14 14,34 14,54 14,74 14,94 14,114 14,134 14,
    150 14,154 14,154 6,134 6,114 6,94 6,74 6,54 6,34 6,
    14 6,10 6,8 6,7 7,6 8,6 10,6 30,6 50,6 70,6 90,6 110,6 130,
    6 150,6 170,6 190,6 194,14 194,14 174,14 154,14 134,14 114,
    14 94,14 74,14 54,14 34,14 14)'::geometry AS l_shape_points
)
SELECT ST_DelaunayTriangles(l_shape_points) AS triangles
FROM data;
```

Figure



Example using vertices with Z values.

Code

```
SELECT ST_DelaunayTriangles(
  'MULTIPOINT Z(14 14 10,150 14 100,34 6 25,20 10 150)'::geometry);
```

See Also

ST_VoronoiPolygons, ST_TriangulatePolygon, CG_ConstrainedDelaunayTriangles, ST_VoronoiLines, ST_ConvexHull

7.14.9 ST_FilterByM

ST_FilterByM — Removes vertices based on their M value

Synopsis

geometry **ST_FilterByM**(geometry geom, double precision min, double precision max = null, boolean returnM = false);

Description

Filters out vertex points based on their M-value. Returns a geometry with only vertex points that have a M-value larger or equal to the min value and smaller or equal to the max value. If max-value argument is left out only min value is considered. If fourth argument is left out the m-value will not be in the resulting geometry. If resulting geometry have too few vertex points left for its geometry type an empty geometry will be returned. In a geometry collection geometries without enough points will just be left out silently.

This function is mainly intended to be used in conjunction with ST_SetEffectiveArea. ST_EffectiveArea sets the effective area of a vertex in its m-value. With ST_FilterByM it then is possible to get a simplified version of the geometry without any calculations, just by filtering



Note
There is a difference in what ST_SimplifyVW returns when not enough points meet the criteria compared to ST_FilterByM. ST_SimplifyVW returns the geometry with enough points while ST_FilterByM returns an empty geometry



Note
Note that the returned geometry might be invalid



Note
This function returns all dimensions, including the Z and M values

Availability: 2.5.0

Examples

A linestring is filtered

Code

```
SELECT ST_FilterByM(geom, 30) simplified
FROM (SELECT  ST_SetEffectiveArea('LINESTRING(5 2,3 8,6 20,7 25,10 10)'::geometry) geom) As ↩
    foo;
```

Output

```
LINESTRING(5 2,7 25,10 10)
```

Figure



See Also

ST_SetEffectiveArea, ST_SimplifyVW

7.14.10 ST_GeneratePoints

ST_GeneratePoints — Generates a multipoint of random points contained in a Polygon or MultiPolygon.

Synopsis

geometry **ST_GeneratePoints**(geometry g, integer npoints, integer seed = 0);

Description

ST_GeneratePoints generates a multipoint consisting of a given number of pseudo-random points which lie within the input area. The optional `seed` is used to regenerate a deterministic sequence of points, and must be greater than zero.

Availability: 2.3.0

Enhanced: 3.0.0, added seed parameter

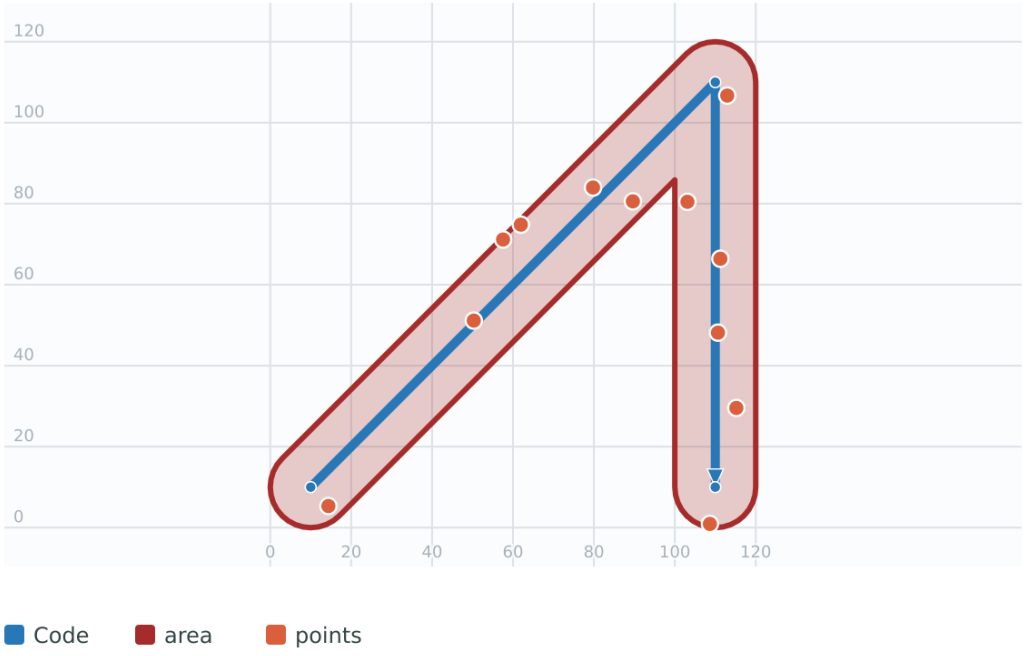
Examples

Generate 12 points inside a buffered LineString using the deterministic seed 1996.

Code

```
WITH source AS (
  SELECT ST_Buffer(ST_GeomFromText('LINESTRING(50 50,150 150,150 50)'),
           10, 'endcap=round join=round') AS geom
)
SELECT geom AS area,
       ST_GeneratePoints(geom, 12, 1996) AS points
FROM source;
```

Figure



Given a table of polygons s, return 12 individual points per polygon. Results will be different each time you run.

Code

```
SELECT s.id, dp.path[1] AS pt_id, dp.geom
FROM s, ST_DumpPoints(ST_GeneratePoints(s.geom, 12)) AS dp;
```

Generate dot-density points from a polygon table, using the absolute value of an attribute as the number of points and keeping the sign of the attribute for styling.

Code

```
SELECT z.id,
       CASE WHEN z.count_delta < 0 THEN -1 ELSE 1 END AS sign,
       dp.path[1] AS pt_id,
       dp.geom
FROM zones AS z
CROSS JOIN LATERAL ST_DumpPoints(
    ST_GeneratePoints(z.geom, abs(z.count_delta)::integer, z.id + 1)
) AS dp
WHERE z.count_delta <> 0;
```

See Also

ST_DumpPoints

7.14.11 ST_GeometricMedian

ST_GeometricMedian — Returns the geometric median of a MultiPoint.

Synopsis

geometry **ST_GeometricMedian** (geometry geom, float8 tolerance = NULL, int max_iter = 10000, boolean fail_if_not_converged = false);

Description

Computes the approximate geometric median of a MultiPoint geometry using the Weiszfeld algorithm. The geometric median is the point minimizing the sum of distances to the input points. It provides a centrality measure that is less sensitive to outlier points than the centroid (center of mass).

The algorithm iterates until the distance change between successive iterations is less than the supplied `tolerance` parameter. If this condition has not been met after `max_iterations` iterations, the function produces an error and exits, unless `fail_if_not_converged` is set to `false` (the default).

If a `tolerance` argument is not provided, the tolerance value is calculated based on the extent of the input geometry.

If present, the input point M values are interpreted as their relative weights.

Availability: 2.3.0

Enhanced: 2.5.0 Added support for M as weight of points.

 This function supports 3d and will not drop the z-index.

 This function supports M coordinates.

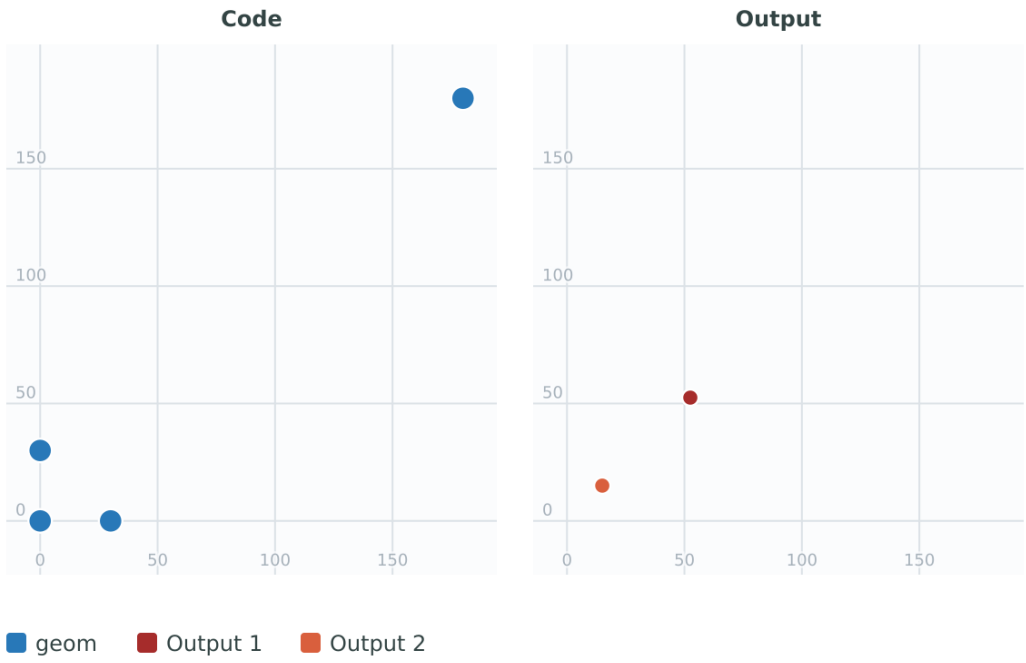
Examples

Compare the geometric median and centroid of a MultiPoint.

Code

```
WITH test AS (
SELECT 'MULTIPOINT((10 10),(10 40),(40 10),(190 190))'::geometry geom)
SELECT
  ST_Centroid(geom) centroid,
  ST_GeometricMedian(geom) median
FROM test;
```

Figure



See Also

ST_Centroid

7.14.12 ST_LineMerge

ST_LineMerge — Return the lines formed by sewing together a MultiLineString.

Synopsis

geometry **ST_LineMerge**(geometry amultilinestring);
geometry **ST_LineMerge**(geometry amultilinestring, boolean directed);

Description

Returns a LineString or MultiLineString formed by joining together the line elements of a MultiLineString. Lines are joined at their endpoints at 2-way intersections. Lines are not joined across intersections of 3-way or greater degree.

If **directed** is TRUE, then ST_LineMerge will not change point order within LineStrings, so lines with opposite directions will not be merged



Note
Only use with MultiLineString/LineStrings. Other geometry types return an empty GeometryCollection

Performed by the GEOS module.

Enhanced: 3.3.0 accept a directed parameter.

Requires GEOS >= 3.11.0 to use the directed parameter.

Availability: 1.1.0



Warning
This function strips the M dimension.

Examples

Cardinality 2

Nodes with cardinality 2 are merged away.

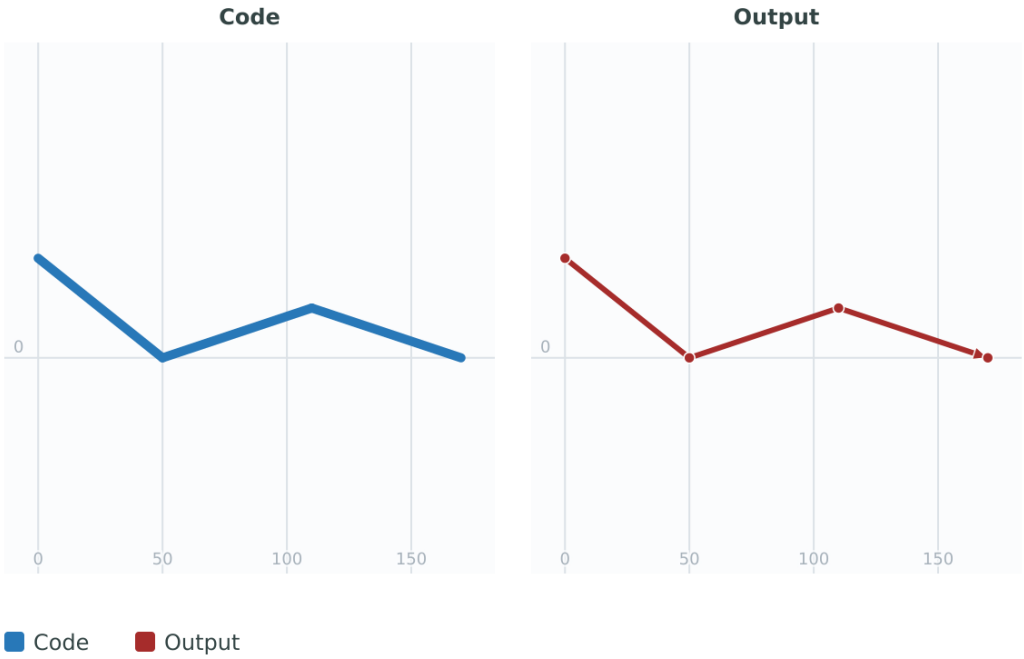
Code

```
SELECT ST_LineMerge('MULTILINESTRING((10 160,60 120),(120 140,60 120),(120 140,180 120))'
);
```

Output

```
LINESTRING(10 160,60 120,120 140,180 120)
```

Figure



Cardinality 3 or More

Lines are not merged across intersections with degree > 2.

Code

```
SELECT ST_LineMerge('MULTILINESTRING((10 160,60 120),(120 140,60 120),(120 140,180 120),(100 ↩
    180,120 140))'
);
```

Output

```
MULTILINESTRING((10 160,60 120,120 140),(100 180,120 140),(120 140,180 120))
```

Figure



Non-Touching Lines

If merging is not possible due to non-touching lines, the original MultiLineString is returned.

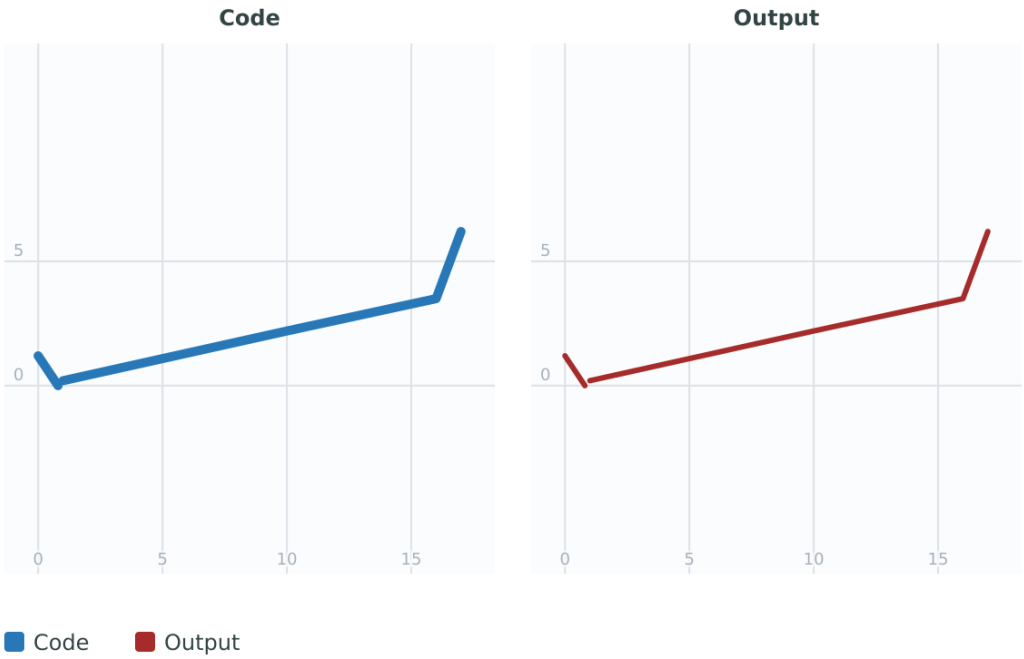
Code

```
SELECT ST_LineMerge('MULTILINESTRING((-29 -27,-30 -29.7,-36 -31,-45 -33),(-45.2 -33.2,-46 ↩
    -32))'
);
```

Output

```
MULTILINESTRING((-45.2 -33.2,-46 -32),(-29 -27,-30 -29.7,-36 -31,-45 -33))
```

Figure



Directed Parameter

Lines with opposite directions are not merged if *directed* is TRUE.

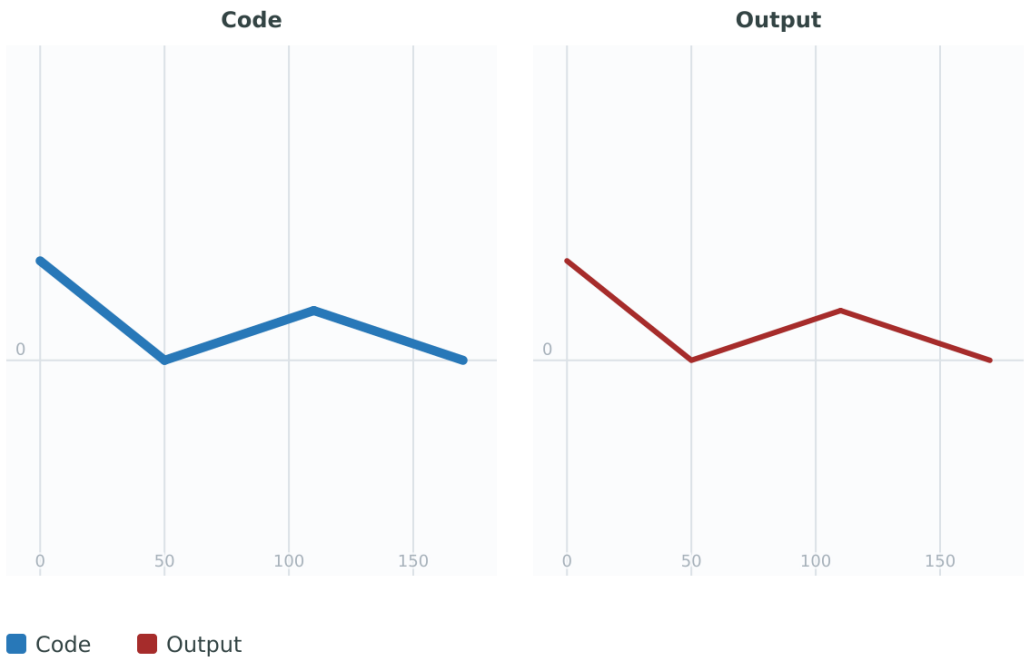
Code

```
SELECT ST_LineMerge('MULTILINESTRING((60 30,10 70),(120 50,60 30),(120 50,180 30))',
TRUE);
```

Output

```
MULTILINESTRING((120 50,60 30,10 70),(120 50,180 30))
```

Figure



Z-dimension Handling

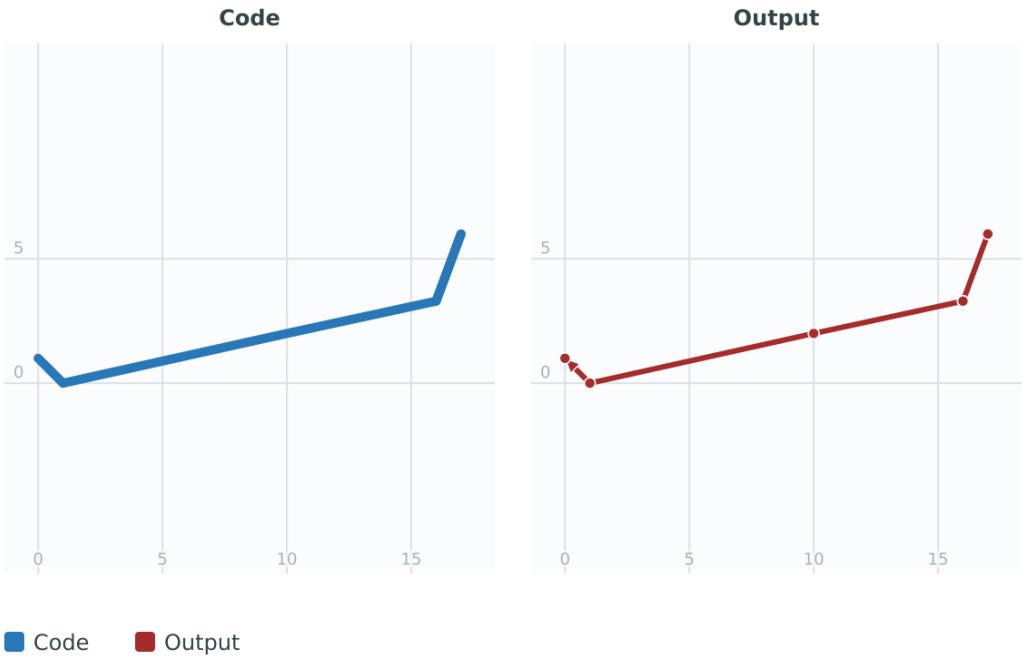
Code

```
SELECT ST_LineMerge('MULTILINESTRING((-29 -27 11,-30 -29.7 10,-36 -31 5,-45 -33 6),(-29 -27 ↩
    12,-30 -29.7 5),(-45 -33 1,-46 -32 11))'
        );
```

Output

```
LINESTRING Z (-30 -29.7 5,-29 -27 11,-30 -29.7 10,-36 -31 5,-45 -33 1,-46 -32 11)
```

Figure



See Also

ST_Segmentize, ST_LineSubstring

7.14.13 ST_MaximumInscribedCircle

ST_MaximumInscribedCircle — Computes the largest circle contained within a geometry.

Synopsis

(geometry, geometry, double precision) **ST_MaximumInscribedCircle**(geometry geom);

Description

Finds the largest circle that is contained within a (multi)polygon, or which does not overlap any lines and points. Returns a record with fields:

- `center` - center point of the circle
- `nearest` - a point on the geometry nearest to the center
- `radius` - radius of the circle

For polygonal inputs, the circle is inscribed within the boundary rings, using the internal rings as boundaries. For linear and point inputs, the circle is inscribed within the convex hull of the input, using the input lines and points as further boundaries.

Availability: 3.1.0.

Requires GEOS >= 3.9.0.

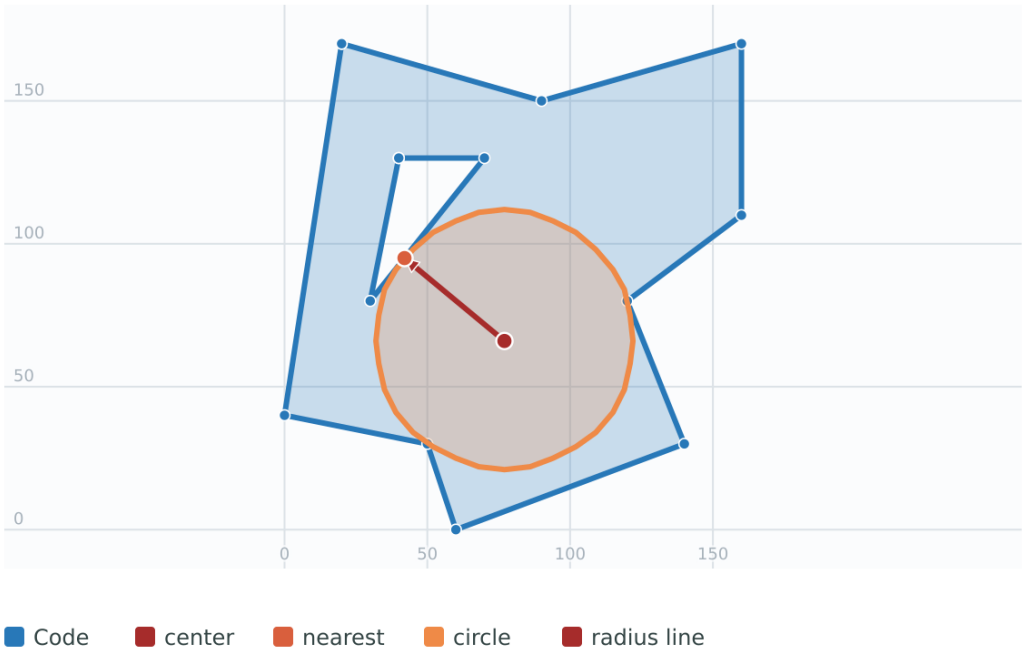
Examples

Maximum inscribed circle of a polygon. Center, nearest point, radius, circle, and radius line are returned.

Code

```
WITH mic AS (
  SELECT * FROM ST_MaximumInscribedCircle(
    'POLYGON ((40 180,110 160,180 180,180 120,140 90,160 40,80 10,70 40,20 50,40 180),
              (60 140,50 90,90 140,60 140))')
)
SELECT round(radius::numeric, 3) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line
FROM mic;
```

Figure

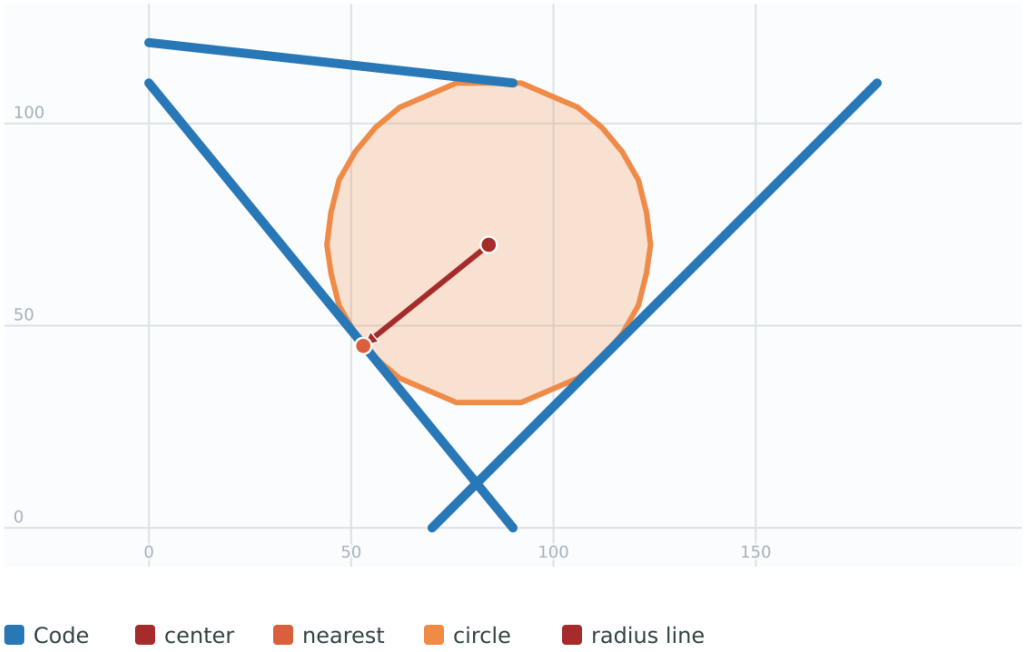


Maximum inscribed circle of a MultiLineString.

Code

```
WITH mic AS (
  SELECT * FROM ST_MaximumInscribedCircle(
    'MULTILINESTRING((100 40,10 150),(10 160,100 150),(190 150,80 40))')
)
SELECT round(radius::numeric, 3) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line
FROM mic;
```

Figure



See Also

ST_MinimumBoundingRadius, ST_LargestEmptyCircle

7.14.14 ST_LargestEmptyCircle

ST_LargestEmptyCircle — Computes the largest circle not overlapping a geometry.

Synopsis

(geometry, geometry, double precision) **ST_LargestEmptyCircle**(geometry geom, double precision tolerance=0.0, geometry boundary=POINT EMPTY);

Description

Finds the largest circle which does not overlap a set of point and line obstacles. (Polygonal geometries may be included as obstacles, but only their boundary lines are used.) The center of the circle is constrained to lie inside a polygonal boundary, which by default is the convex hull of the input geometry. The circle center is the point in the interior of the boundary which has the farthest distance from the obstacles. The circle itself is provided by the center point and a nearest point lying on an obstacle determining the circle radius.

If a boundary is supplied it must be a valid polygonal geometry with positive area.

The circle center is determined to a given accuracy specified by a distance tolerance, using an iterative algorithm. If the accuracy distance is not specified a reasonable default is used.

Returns a record with fields:

- `center` - center point of the circle
- `nearest` - a point on the geometry nearest to the center

- `radius` - radius of the circle

To find the largest empty circle in the interior of a polygon, see ST_MaximumInscribedCircle.

Availability: 3.4.0.

Requires GEOS >= 3.9.0.

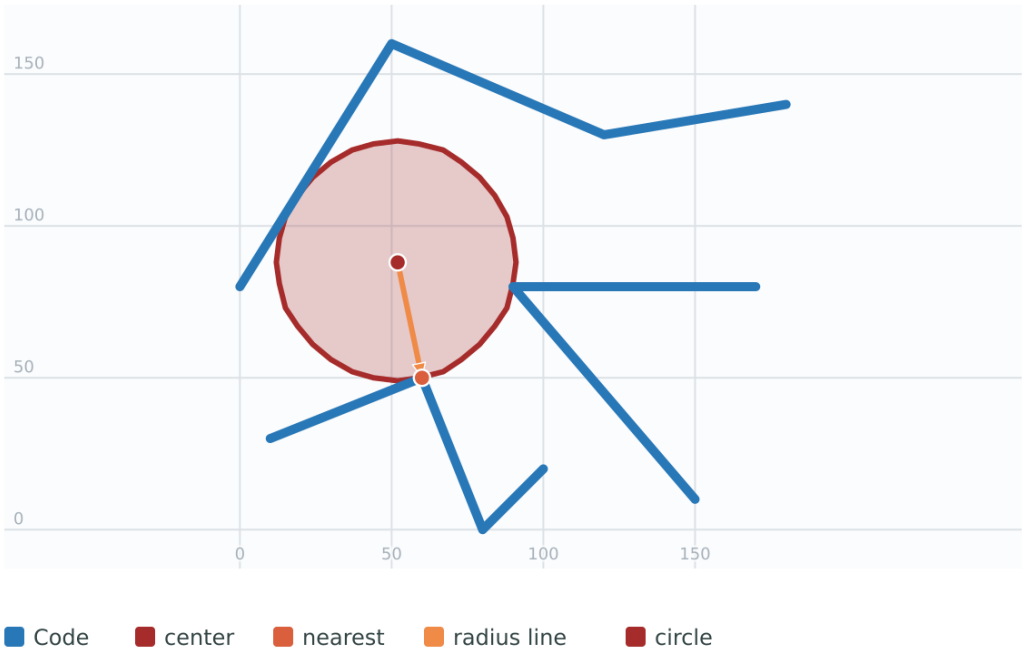
Examples

Largest empty circle within a set of lines.

Code

```
WITH obstacles(geom) AS (VALUES ('MULTILINESTRING (
        (10 100,60 180,130 150,190 160),
        (20 50,70 70,90 20,110 40),
        (160 30,100 100,180 100))'::geometry)),
circle AS (
  SELECT result.*
  FROM obstacles
  CROSS JOIN LATERAL ST_LargestEmptyCircle(obstacles.geom) AS result
)
SELECT round(radius::numeric, 6) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle
FROM circle;
```

Figure

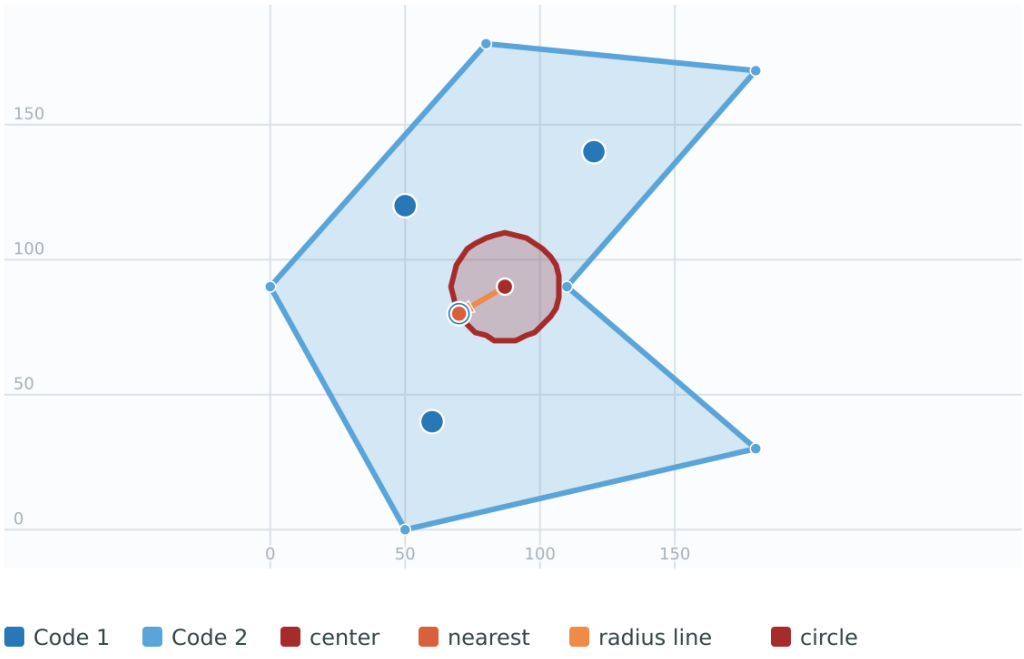


Largest empty circle within a set of points, constrained to lie in a polygon. The constraint polygon boundary is included both as an obstacle and as the constraint for the circle center.

Code

```
WITH inputs(points, boundary) AS (VALUES (
  'MULTIPOINT ((70 50),(60 130),(130 150),(80 90))'::geometry,
  'POLYGON ((90 190,10 100,60 10,190 40,120 100,190 180,90 190))'::geometry
)),
circle AS (
  SELECT result.*
  FROM inputs
  CROSS JOIN LATERAL ST_LargestEmptyCircle(
    ST_Collect(points, boundary), 0, boundary
  ) AS result
)
SELECT round(radius::numeric, 6) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle
FROM circle;
```

Figure



See Also

ST_MinimumBoundingRadius

7.14.15 ST_MinimumBoundingCircle

ST_MinimumBoundingCircle — Returns the smallest circle polygon that contains a geometry.

Synopsis

geometry **ST_MinimumBoundingCircle**(geometry geomA, integer num_segs_per_qt_circ=48);

Description

Returns the smallest circle polygon that contains a geometry.



Note
The bounding circle is approximated by a polygon with a default of 48 segments per quarter circle. Because the polygon is an approximation of the minimum bounding circle, some points in the input geometry may not be contained within the polygon. The approximation can be improved by increasing the number of segments. For applications where an approximation is not suitable ST_MinimumBoundingRadius may be used.

Use with ST_Collect to get the minimum bounding circle of a set of geometries.

To compute two points lying on the minimum circle (the "maximum diameter") use ST_LongestLine.

The ratio of the area of a polygon divided by the area of its Minimum Bounding Circle is referred to as the *Reock compactness score*.

Performed by the GEOS module.

Availability: 1.4.0

Examples

Code

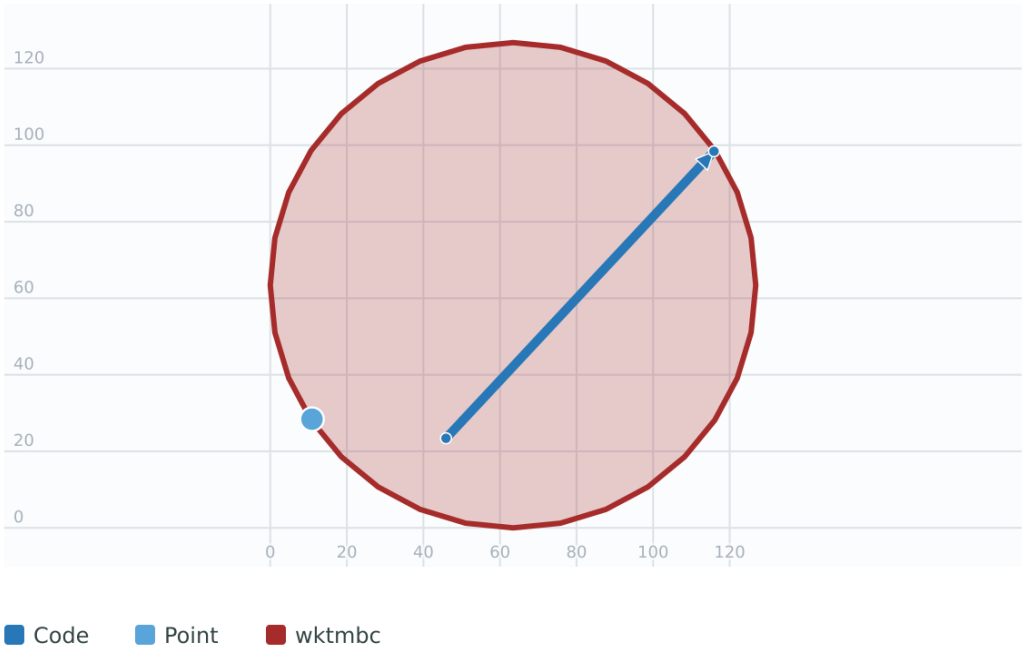
```
SELECT d.disease_type,
ST_MinimumBoundingCircle(ST_Collect(d.geom)) As geom
FROM disease_obs As d
GROUP BY d.disease_type;
```

Minimum bounding circle of a point and linestring, using 8 segments to approximate a quarter circle.

Code

```
SELECT ST_MinimumBoundingCircle(ST_Collect('LINESTRING(55 75,125 150)',
ST_Point(20, 80)), 8
) As wktmbc;
```

Figure



See Also

ST_Collect, ST_MinimumBoundingRadius, ST_LargestEmptyCircle, ST_LongestLine

7.14.16 ST_MinimumBoundingRadius

ST_MinimumBoundingRadius — Returns the center point and radius of the smallest circle that contains a geometry.

Synopsis

(geometry, double precision) **ST_MinimumBoundingRadius**(geometry geom);

Description

Computes the center point and radius of the smallest circle that contains a geometry. Returns a record with fields:

- `center` - center point of the circle
- `radius` - radius of the circle

Use with ST_Collect to get the minimum bounding circle of a set of geometries.

To compute two points lying on the minimum circle (the "maximum diameter") use ST_LongestLine.

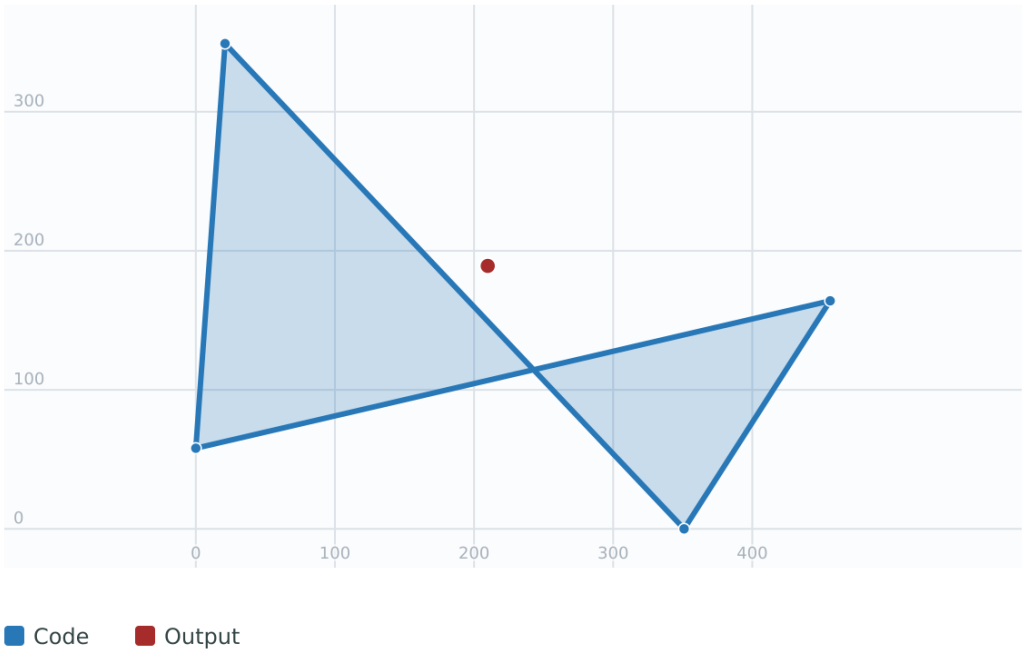
Availability - 2.3.0

Examples

Code

```
SELECT center, radius FROM ST_MinimumBoundingRadius('POLYGON((26426 65078,26531 65242,26075 ↩
    65136,26096 65427,26426 65078))');
```

Figure



See Also

ST_Collect, ST_MinimumBoundingCircle, ST_LongestLine

7.14.17 ST_OrientedEnvelope

ST_OrientedEnvelope — Returns a minimum-area rectangle containing a geometry.

Synopsis

geometry **ST_OrientedEnvelope**(geometry geom);

Description

Returns the minimum-area rotated rectangle enclosing a geometry. Note that more than one such rectangle may exist. May return a Point or LineString in the case of degenerate inputs.

Availability: 2.5.0.

Requires GEOS >= 3.6.0.

Examples

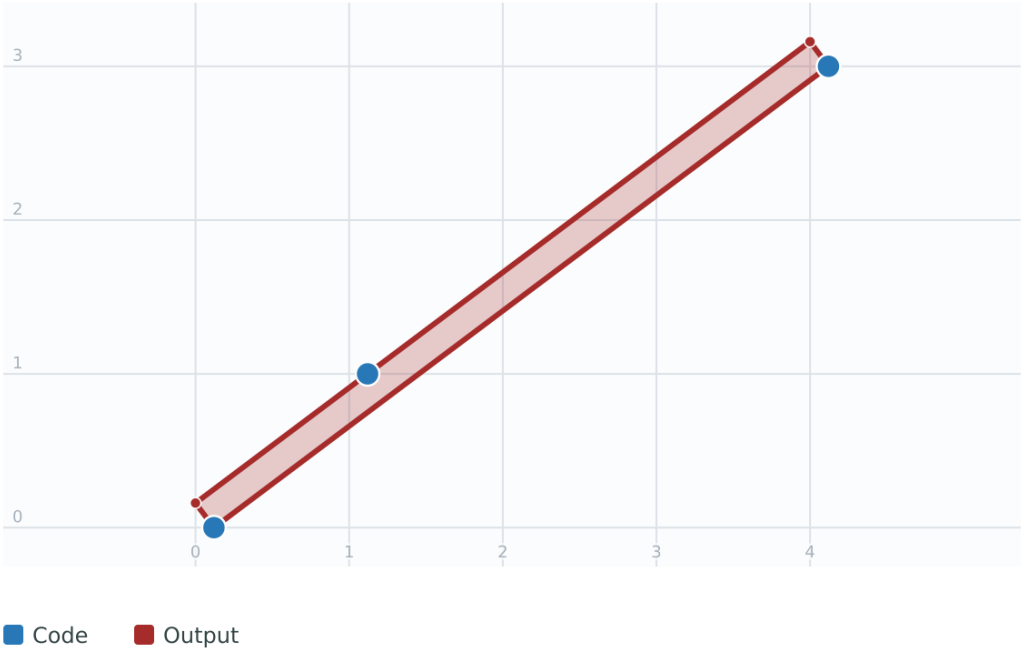
Code

```
SELECT ST_OrientedEnvelope('MULTIPOINT ((0 0),(-1 -1),(3 2))');
```

Output

```
POLYGON((3 2,-1 -1,-1.12 -0.84,2.88 2.16,3 2))
```

Figure



Oriented envelope of a Point and LineString.

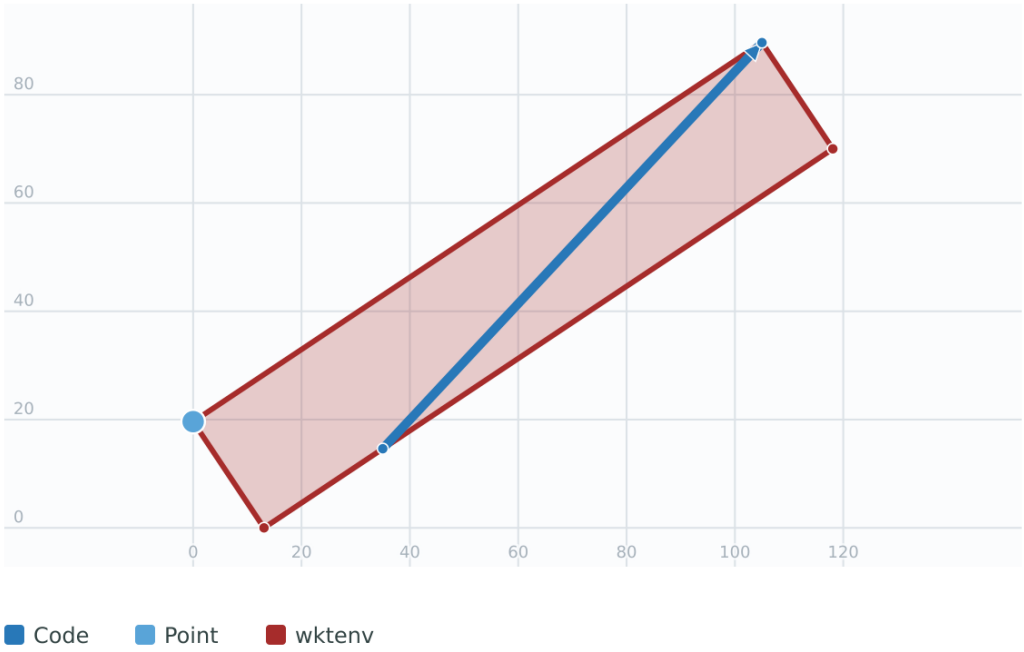
Code

```
SELECT ST_OrientedEnvelope(ST_Collect('LINESTRING(55 75,125 150)',
                ST_Point(20, 80))
                ) As wktenv;
```

Output

```
POLYGON((20 80,125 150,138.0769230769241 130.38461538461573,33.07692307692291 60.384615384615195,20 80))
```

Figure



See Also

ST_Envelope ST_MinimumBoundingCircle

7.14.18 ST_OffsetCurve

ST_OffsetCurve — Returns an offset line at a given distance and side from an input line.

Synopsis

geometry **ST_OffsetCurve**(geometry line, float signed_distance, text style_parameters='');

Description

Return an offset line at a given distance and side from an input line. All points of the returned geometries are not further than the given distance from the input geometry. Useful for computing parallel lines about a center line.

For positive distance the offset is on the left side of the input line and retains the same direction. For a negative distance it is on the right side and in the opposite direction.

Units of distance are measured in units of the spatial reference system.

Note that output may be a MULTILINESTRING or EMPTY for some jigsaw-shaped input geometries.

The optional third parameter allows specifying a list of blank-separated key=value pairs to tweak operations as follows:

- 'quad_segs=#' : number of segments used to approximate a quarter circle (defaults to 8).
- 'join=round|mitre|bevel' : join style (defaults to "round"). 'miter' is also accepted as a synonym for 'mitre'.
- 'mitre_limit=#.#' : mitre ratio limit (only affects mitred join style). 'miter_limit' is also accepted as a synonym for 'mitre_limit'.

Performed by the GEOS module.

Behavior changed in GEOS 3.11 so offset curves now have the same direction as the input line, for both positive and negative offsets.

Availability: 2.0

Enhanced: 2.5 - added support for GEOMETRYCOLLECTION and MULTILINESTRING



Note

This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

Examples

Compute an open buffer around roads.

Code

```
SELECT ST_Union(
 ST_OffsetCurve(f.geom, f.width/2, 'quad_segs=4 join=round'),
 ST_OffsetCurve(f.geom, -f.width/2, 'quad_segs=4 join=round')
) as track
FROM someroadstable;
```

Offset 15 units with `quad_segs=4 join=round`.

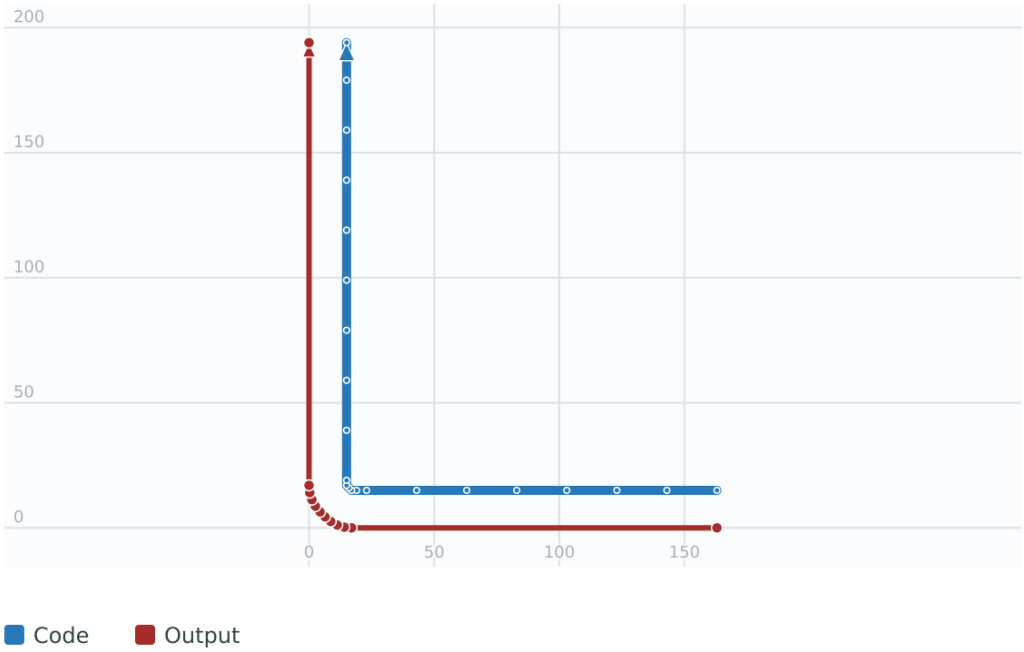
Code

```
SELECT ST_OffsetCurve(
    ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)'),
    15, 'quad_segs=4 join=round');
```

Output

```
LINESTRING(164 1,18 1,15.073645169758077 1.288220793951544,
    12.259748514523654 2.141807012330698,9.66644650470597 3.527955815461818,
    7.393398282201788 5.393398282201788,5.393398282201788 7.393398282201788,
    3.527955815461818 9.66644650470597,2.141807012330696 12.259748514523654,
    1.288220793951544 15.073645169758075,1 18,1 195)
```

Figure



Offset -15 units with `quad_segs=4 join=round`.

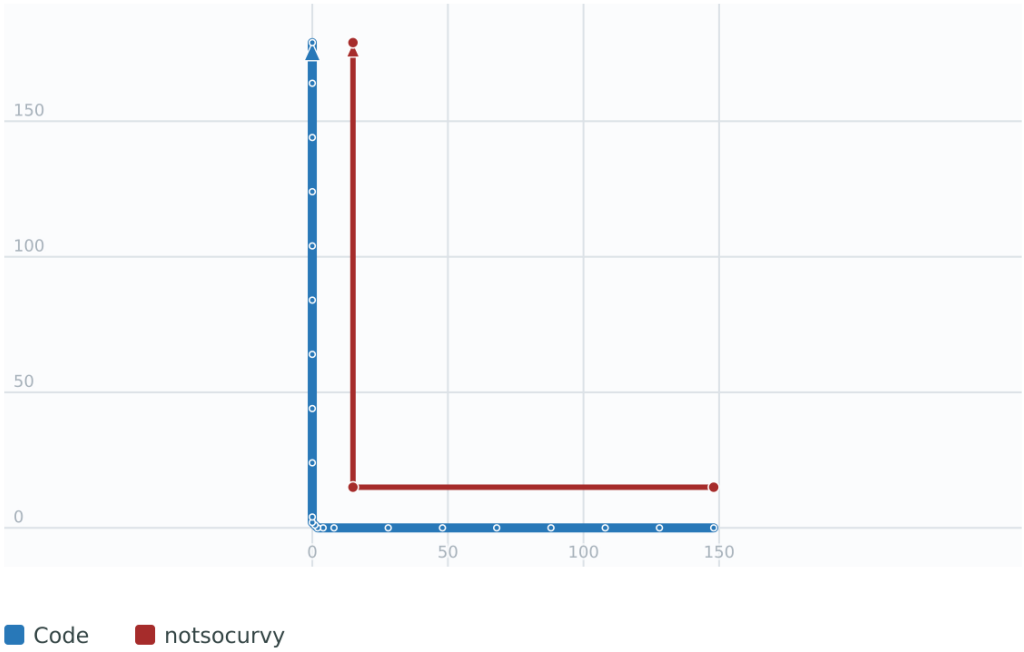
Code

```
SELECT ST_OffsetCurve(
    geom,
    -15,
    'quad_segs=4 join=round') As notsocurvy
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
LINESTRING(164 31,31 31,31 195)
```

Figure



Apply two negative offsets to obtain a smoother curve; the distances combine to -45 units.

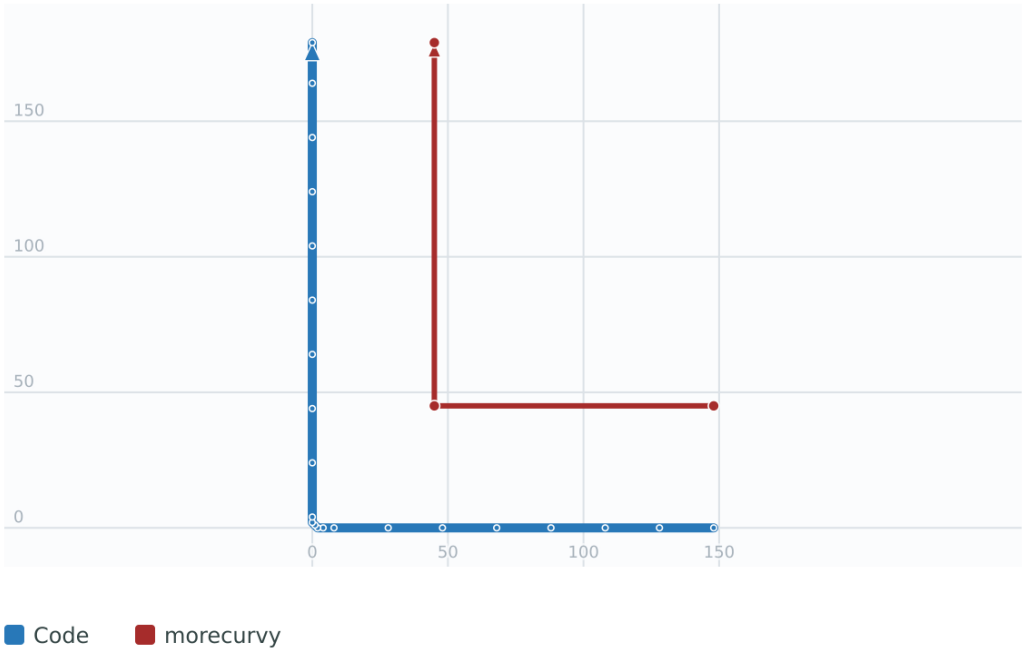
Code

```
SELECT ST_OffsetCurve(
    ST_OffsetCurve(
        geom,
        -30,
        'quad_segs=4 join=round'),
    -15,
    'quad_segs=4 join=round') As morecurvy
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
LINESTRING(164 61,61 61,61 195)
```

Figure



Collect the positive offset and the double-offset to produce parallel curves.

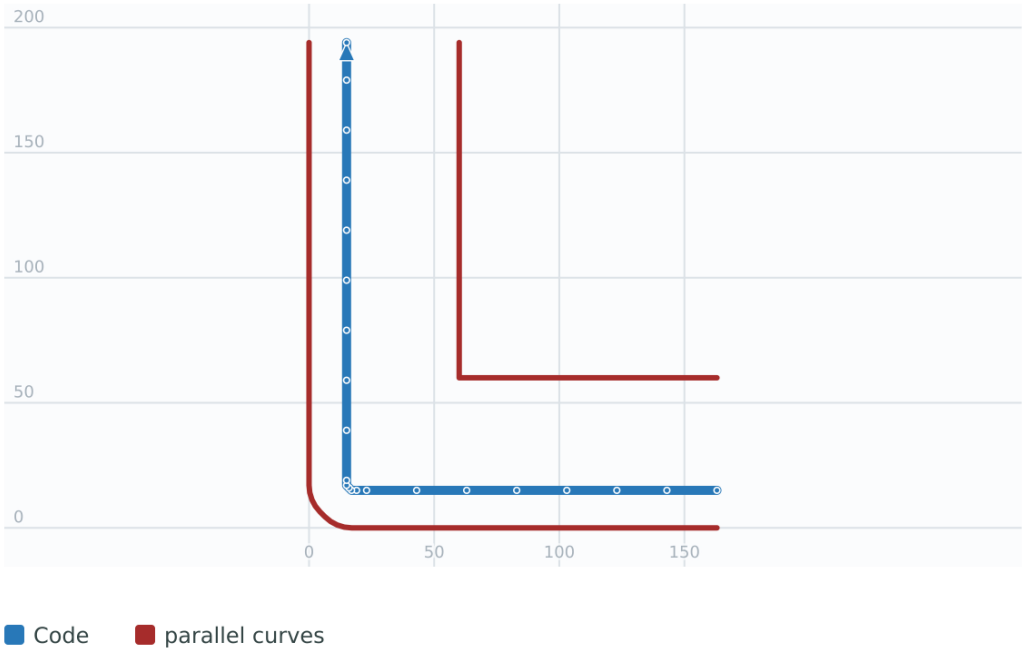
Code

```
SELECT ST_Collect(
    ST_OffsetCurve(geom, 15, 'quad_segs=4 join=round'),
    ST_OffsetCurve(
        ST_OffsetCurve(
            geom,
            -30,
            'quad_segs=4 join=round'),
        -15,
        'quad_segs=4 join=round')
) As parallel_curves
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
MULTILINESTRING((164 1,18 1,15.073645169758077 1.288220793951544,
12.259748514523654 2.141807012330698,9.66644650470597 3.527955815461818,
7.393398282201788 5.393398282201788,5.393398282201788 7.393398282201788,
3.527955815461818 9.66644650470597,2.141807012330696 12.259748514523654,
1.288220793951544 15.073645169758075,1 18,1 195),(164 61,61 61,61 195))
```

Figure



Offset 15 units with a bevel join.

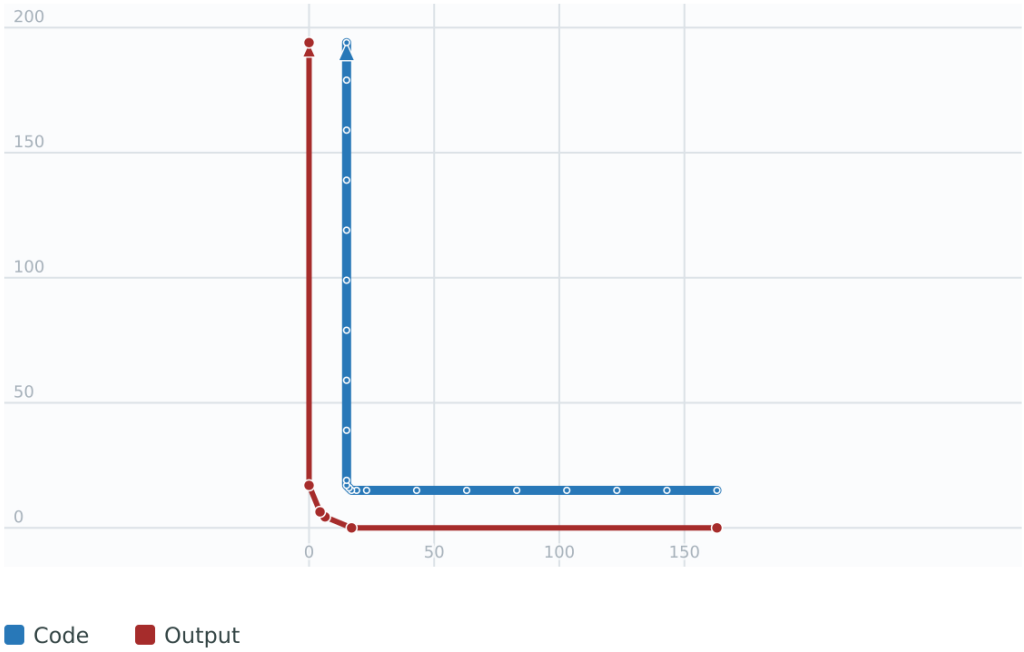
Code

```
SELECT ST_OffsetCurve(
    ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)'),
        15, 'quad_segs=4 join=bevel');
```

Output

```
LINESTRING(164 1,18 1,7.393398282201788 5.393398282201788,
    5.393398282201788 7.393398282201788,1 18,1 195)
```

Figure



Collect offsets on both sides using a mitre join and a mitre limit of 2.2.

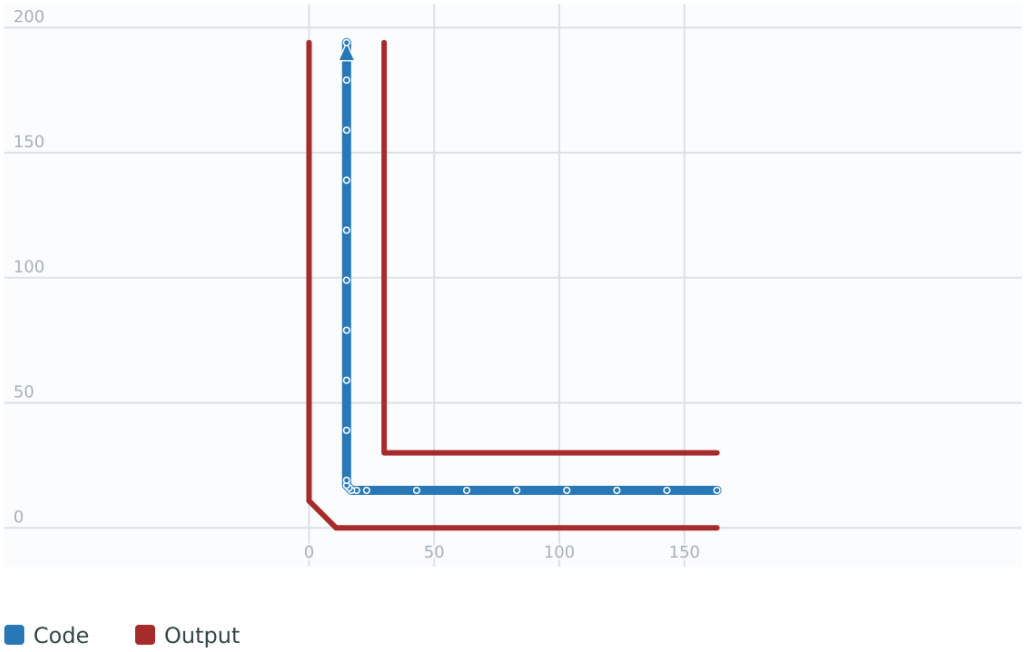
Code

```
SELECT ST_Collect(
    ST_OffsetCurve(geom, 15, 'quad_segs=4 join=mitre mitre_limit=2.2'),
    ST_OffsetCurve(geom, -15, 'quad_segs=4 join=mitre mitre_limit=2.2')
    )
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
MULTILINESTRING((164 1,11.786796564403577 1,1 11.786796564403577,1 195),
    (164 31,31 31,31 195))
```

Figure



See Also

ST_Buffer

7.14.19 ST_PointOnSurface

ST_PointOnSurface — Computes a point guaranteed to lie in a polygon, or on a geometry.

Synopsis

geometry **ST_PointOnSurface**(geometry g1);

Description

Returns a `POINT` which is guaranteed to lie in the interior of a surface (`POLYGON`, `MULTIPOLYGON`, and `CURVEPOLYGON`). In PostGIS this function also works on line and point geometries.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.14.2 // s3.2.18.2

This method implements the SQL/MM specification. SQL-MM 3: 8.1.5, 9.5.6. The specifications define ST_PointOnSurface for surface geometries only. PostGIS extends the function to support all common geometry types. Other databases (Oracle, DB2, ArcSDE) seem to support this function only for surfaces. SQL Server 2008 supports all common geometry types.

This function supports 3d and will not drop the z-index.

Examples

Point on surface of a MultiPoint.

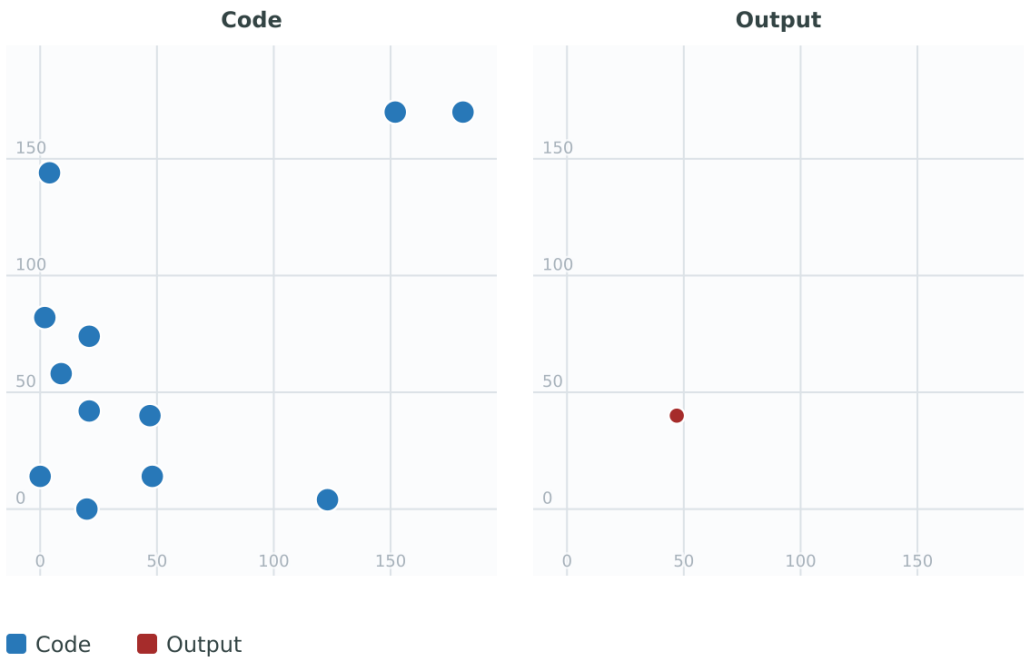
Code

```
SELECT ST_PointOnSurface(
  'MULTIPOINT (8 24,10 92,12 154,17 68,28 10,29 52,29 84,55 50,56 24,131 14,160 180,189 180) ↩
    ');
```

Output

```
POINT(55 50)
```

Figure



Point on surface of a LineString.

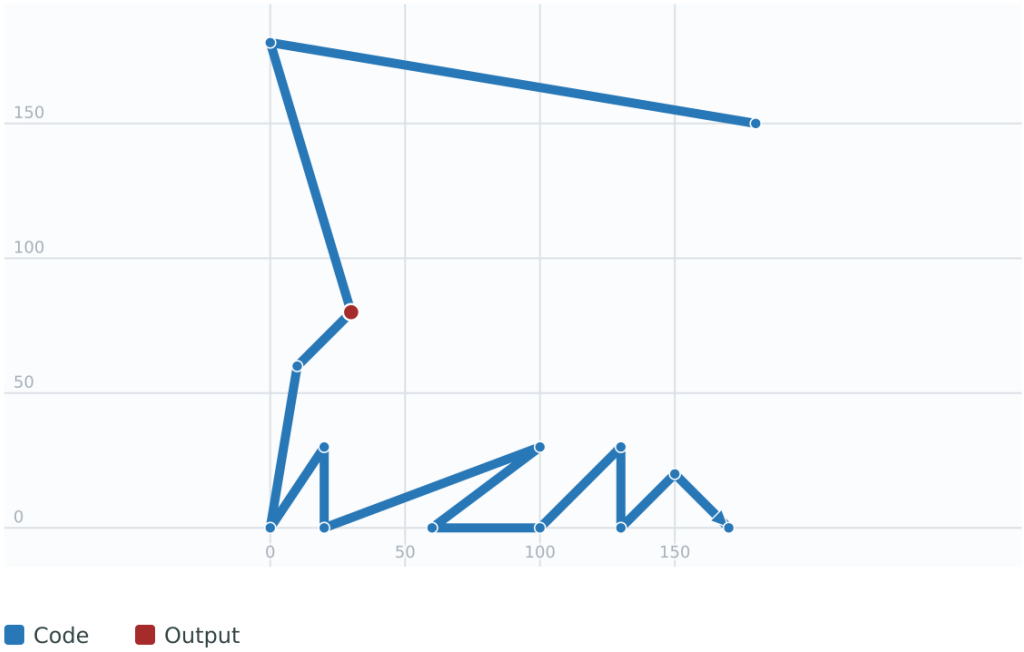
Code

```
SELECT ST_PointOnSurface(
  'LINESTRING (190 160,10 190,40 90,20 70,10 10,30 40,30 10,110 40,70 10,110 10,140 40,140  ↩
    10,160 30,180 10)');
```

Output

```
POINT(40 90)
```

Figure



Point on surface of a Polygon.

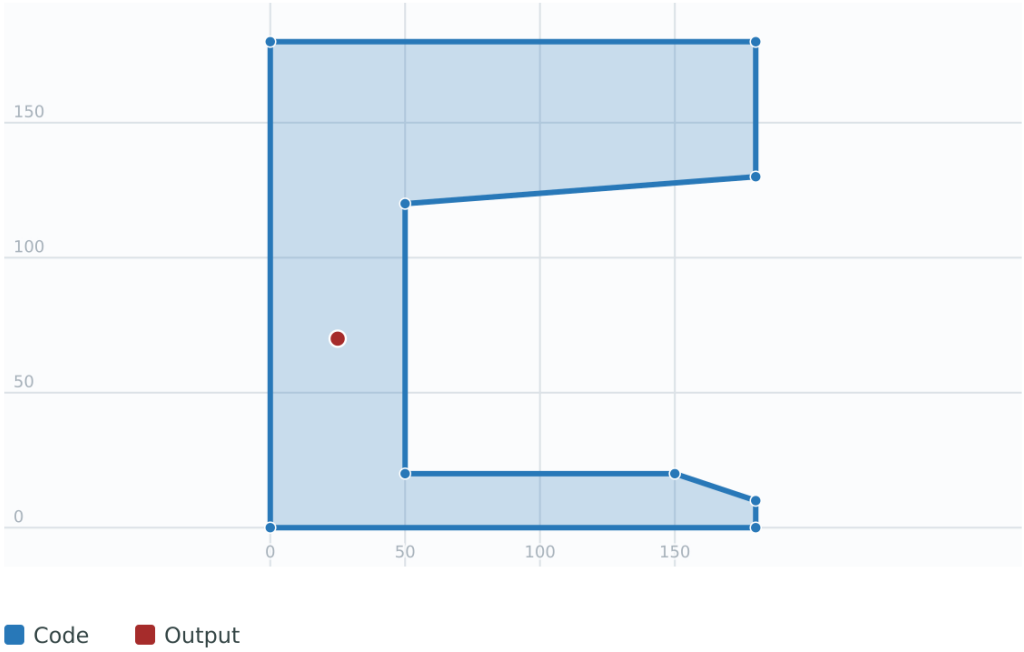
Code

```
SELECT ST_PointOnSurface(
  'POLYGON ((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))');
```

Output

```
POINT(35 80)
```

Figure



Point on surface of the highest-dimension components of a GeometryCollection.

Code

```
SELECT ST_PointOnSurface(
  'GEOMETRYCOLLECTION (POLYGON ((190 170,180 100,80 140,80 160,130 160,110 180,110 190,180 ↩
   180,190 170)),
    LINESTRING (80 120,120 20,140 70,150 30,180 50,190 10),
    MULTIPOINT (19 150,22 49,30 13,32 101,45 35,67 88,75 16))');
```

Output

```
POINT(133.571 150)
```

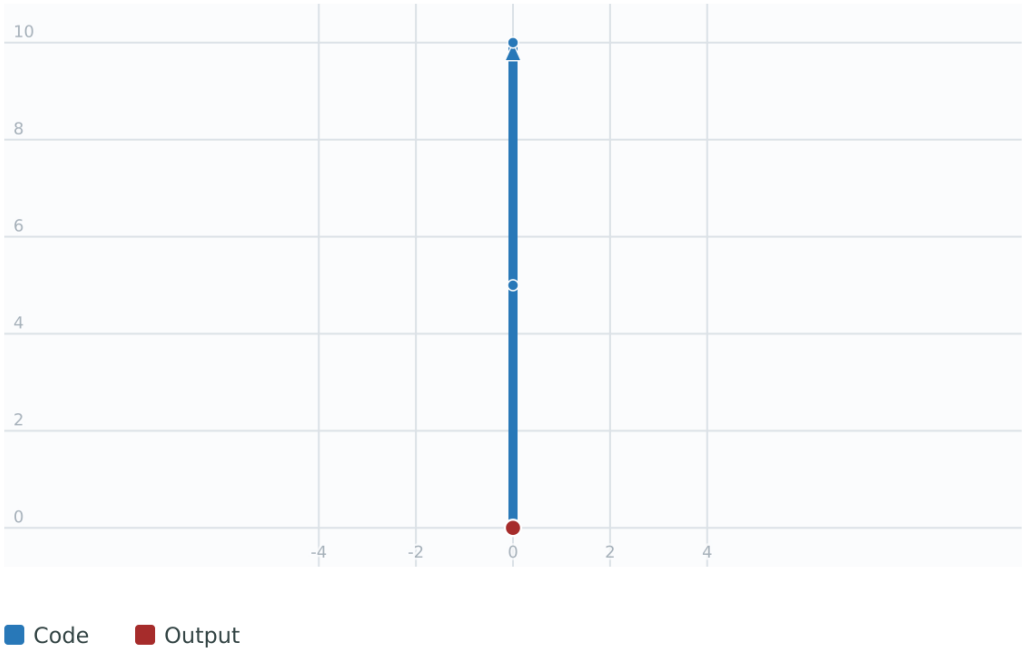
Figure



Code

```
SELECT ST_PointOnSurface('LINESTRING(0 5 1,0 0 1,0 10 2)');
```

Figure

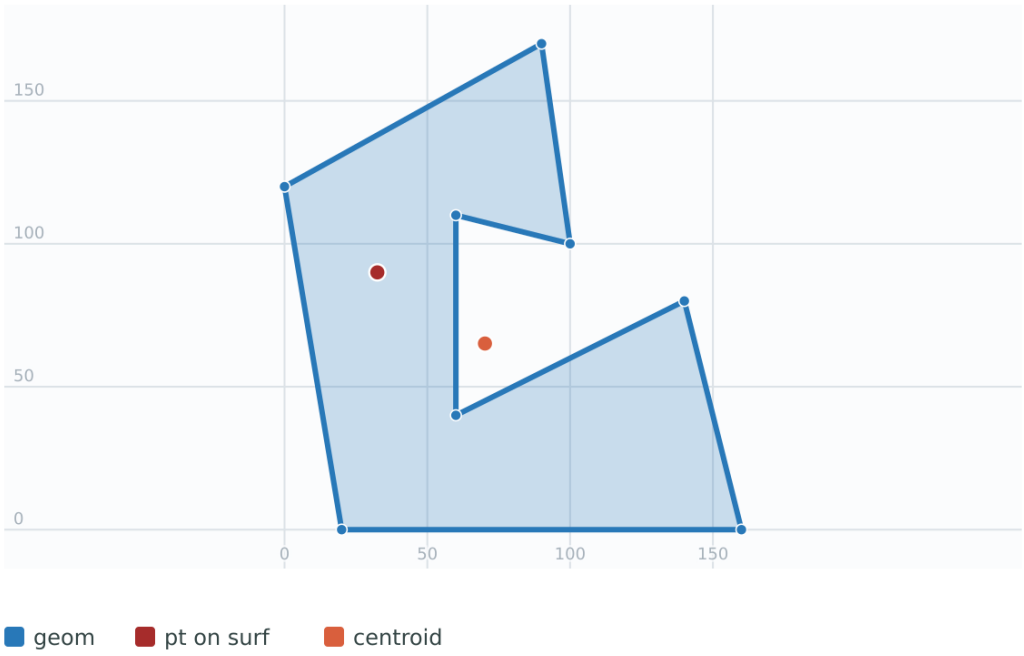


Example: The result of ST_PointOnSurface is guaranteed to lie within polygons, whereas the point computed by ST_Centroid may be outside.

Code

```
SELECT ST_PointOnSurface(geom) AS pt_on_surf, ST_Centroid(geom) AS centroid
    FROM (SELECT 'POLYGON ((130 120,120 190,30 140,50 20,190 20,
                      170 100,90 60,90 130,130 120))'::geometry AS geom) AS t;
```

Figure



See Also

ST_Centroid, ST_MaximumInscribedCircle

7.14.20 ST_Polygonize

ST_Polygonize — Computes a collection of polygons formed from the linework of a set of geometries.

Synopsis

geometry **ST_Polygonize**(geometry set geomfield);
geometry **ST_Polygonize**(geometry[] geom_array);

Description

Creates a GeometryCollection containing the polygons formed by the linework of a set of geometries. If the input linework does not form any polygons, an empty GeometryCollection is returned.

This function creates polygons covering all delimited areas. If the result is intended to form a valid polygonal geometry, use ST_BuildArea to prevent holes being filled.



Note
The input linework must be correctly noded for this function to work properly. To ensure input is noded use ST_Node on the input geometry before polygonizing.



Note
GeometryCollections can be difficult to handle with external tools. Use ST_Dump to convert the polygonized result into separate polygons.

Performed by the GEOS module.

Availability: 1.0.0RC1

Examples

Polygonize a set of input lines.

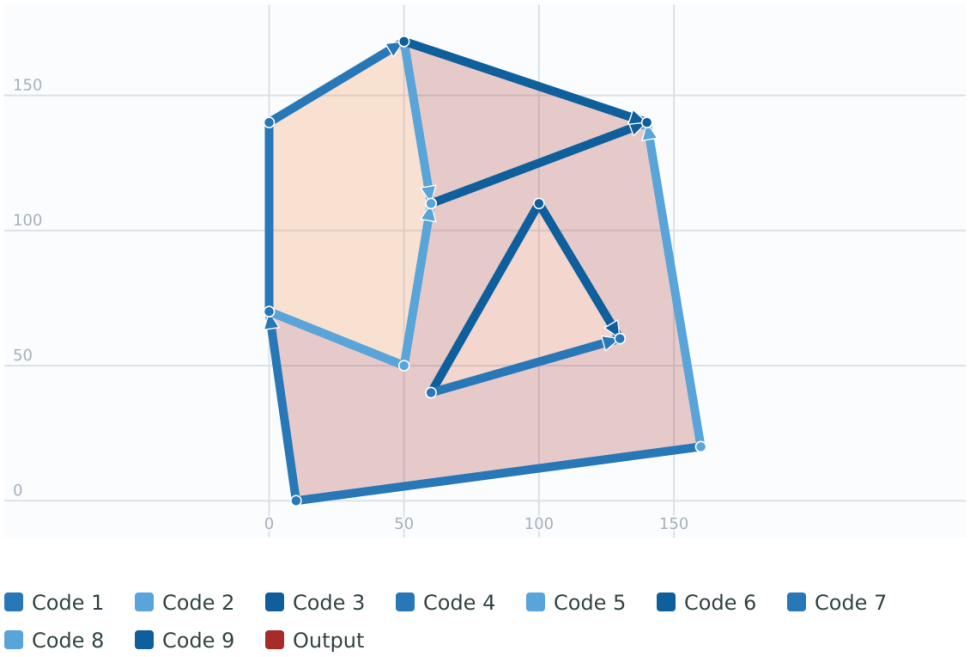
Code

```
WITH data(geom) AS (
  VALUES
    ('LINESTRING (180 40,30 20,20 90)'::geometry),
    ('LINESTRING (180 40,160 160)'::geometry),
    ('LINESTRING (80 60,120 130,150 80)'::geometry),
    ('LINESTRING (80 60,150 80)'::geometry),
    ('LINESTRING (20 90,70 70,80 130)'::geometry),
    ('LINESTRING (80 130,160 160)'::geometry),
    ('LINESTRING (20 90,20 160,70 190)'::geometry),
    ('LINESTRING (70 190,80 130)'::geometry),
    ('LINESTRING (70 190,160 160)'::geometry)
)
SELECT ST_Polygonize(geom)
FROM data;
```

Output

```
GEOMETRYCOLLECTION(POLYGON((180 40,30 20,20 90,70 70,80 130,160 160,180 40),(150 80,120 130,80 60,150 80)), POLYGON((80 ↩
      60,120 130,150 80,80 60)), POLYGON((80 130,70 70,20 90,20 160,70 190,80 130)), POLYGON((160 160,80 130,70 190,160 ↩
      160)))
```

Figure

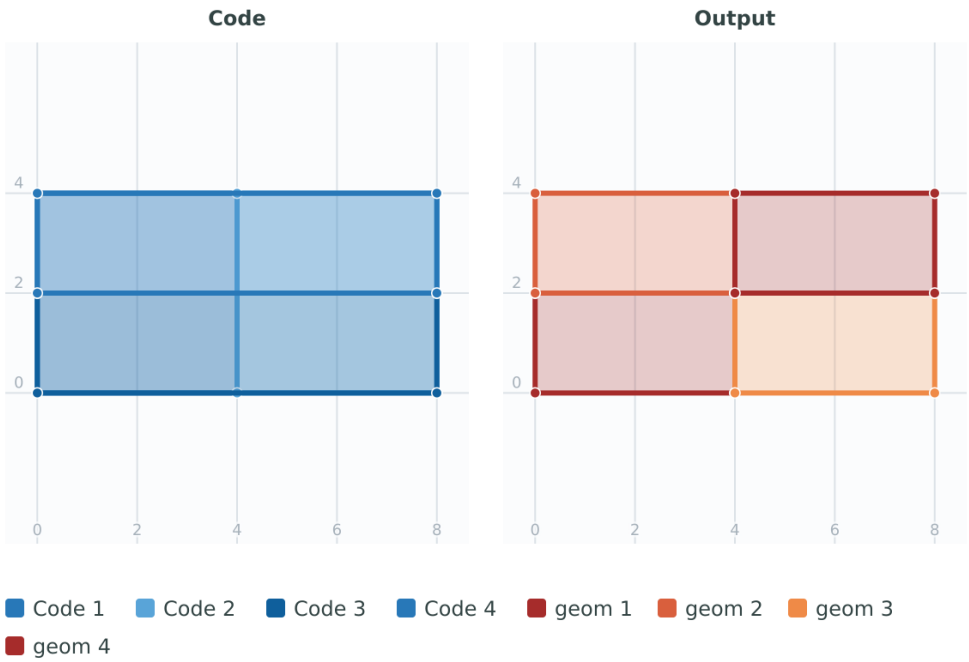


Polygonizing two polygon coverages and transferring attributes to the resultants:

Code

```
WITH poly_a(id, geom) AS (VALUES
  ('a1', 'POLYGON ((0 0, 4 0, 4 4, 0 4, 0 0))'::geometry),
  ('a2', 'POLYGON ((4 0, 8 0, 8 4, 4 4, 4 0))'::geometry)
),
poly_b(id, geom) AS (VALUES
  ('b1', 'POLYGON ((0 0, 8 0, 8 2, 0 2, 0 0))'::geometry),
  ('b2', 'POLYGON ((0 2, 8 2, 8 4, 0 4, 0 2))'::geometry)
),
lines AS (
  SELECT ST_Boundary(geom) AS geom FROM poly_a
  UNION ALL
  SELECT ST_Boundary(geom) FROM poly_b
),
noded AS (
  SELECT ST_UnaryUnion(ST_Collect(geom)) AS geom FROM lines
),
cells AS (
  SELECT (ST_Dump(ST_Polygonize(geom))).geom AS geom FROM noded
)
SELECT a.id AS id_a,
       b.id AS id_b,
       c.geom AS geom
FROM cells AS c
LEFT JOIN poly_a AS a ON ST_Within(ST_PointOnSurface(c.geom), a.geom)
LEFT JOIN poly_b AS b ON ST_Within(ST_PointOnSurface(c.geom), b.geom)
WHERE a.id IS NOT NULL OR b.id IS NOT NULL
ORDER BY id_a, id_b;
```

Figure



Polygonizing a table of linestrings:

Code

```
SELECT ST_Polygonize(geom_4269) As geomtextrep
FROM (SELECT geom_4269 FROM ma.suffolk_edges) As foo;
```

Output

```
 geomtextrep
-------------
 SRID=4269;GEOMETRYCOLLECTION(POLYGON((-71.040878 42.285678,-71.040943 42.2856,-71.04096  ↩
    42.285752,-71.040878 42.285678)),
 POLYGON((-71.17166 42.353675,-71.172026 42.354044,-71.17239 42.354358,-71.171794  ↩
    42.354971,-71.170511 42.354855,
 -71.17112 42.354238,-71.17166 42.353675)))
(1 row)
```

Use ST_Dump to dump out the polygonize geoms into individual polygons.

Code

```
SELECT (ST_Dump(t.polycoll)).geom AS geomtextrep
FROM (SELECT ST_Polygonize(geom_4269) AS polycoll
    FROM (SELECT geom_4269 FROM ma.suffolk_edges)
        As foo) AS t;
```

Output

```
 geomtextrep
-------------
 SRID=4269;POLYGON((-71.040878 42.285678,-71.040943 42.2856,-71.04096 42.285752,
-71.040878 42.285678))
 SRID=4269;POLYGON((-71.17166 42.353675,-71.172026 42.354044,-71.17239 42.354358,
-71.171794 42.354971,-71.170511 42.354855,-71.17112 42.354238,-71.17166 42.353675))
(2 rows)
```

See Also

ST_BuildArea, ST_Boundary, ST_Collect, ST_Dump, ST_Node, ST_PointOnSurface, ST_UnaryUnion, ST_Within

7.14.21 ST_ReducePrecision

ST_ReducePrecision — Returns a valid geometry with points rounded to a grid tolerance.

Synopsis

geometry **ST_ReducePrecision**(geometry g, float8 gridsize);

Description

Returns a valid geometry with all points rounded to the provided grid tolerance, and features below the tolerance removed.

Unlike ST_SnapToGrid the returned geometry will be valid, with no ring self-intersections or collapsed components.

Precision reduction can be used to:

- match coordinate precision to the data accuracy
- reduce the number of coordinates needed to represent a geometry
- ensure valid geometry output to formats which use lower precision (e.g. text formats such as WKT, GeoJSON or KML when the number of output decimal places is limited).
- export valid geometry to systems which use lower or limited precision (e.g. SDE, Oracle tolerance value)

Availability: 3.1.0.

Requires GEOS >= 3.9.0.

Examples

Code

```
SELECT ST_ReducePrecision('POINT(1.412 19.323)', 0.1);
```

Output

```
POINT(1.4 19.3)
```

Figure



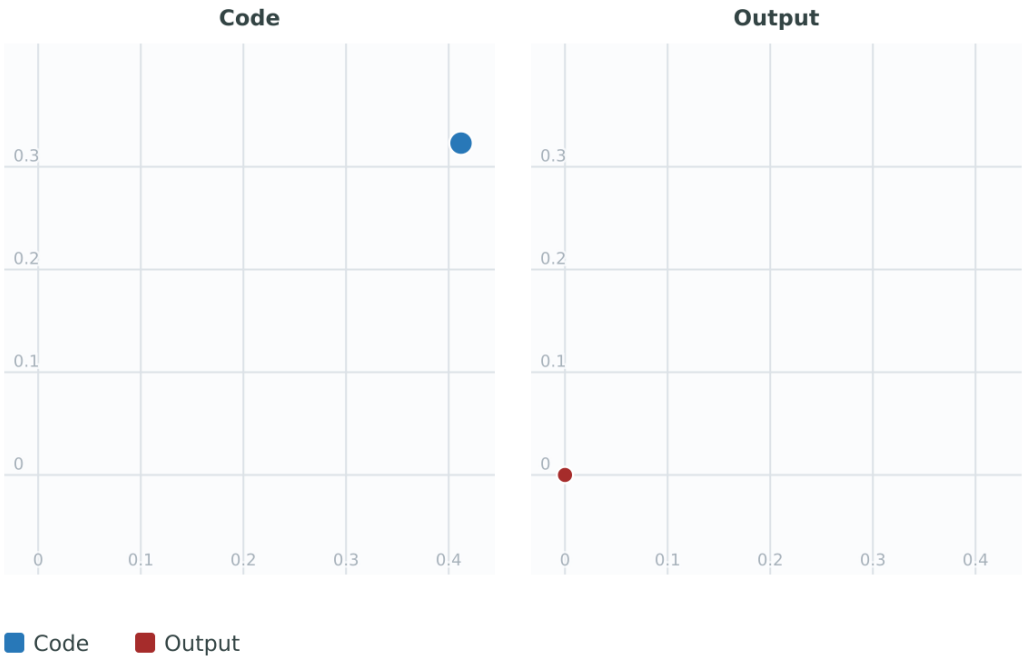
Code

```
SELECT ST_ReducePrecision('POINT(1.412 19.323)', 1.0);
```

Output

```
POINT(1 19)
```

Figure



Code

```
SELECT ST_ReducePrecision('POINT(1.412 19.323)', 10);
```

Output

```
POINT(0 20)
```

Figure



Precision reduction can reduce number of vertices

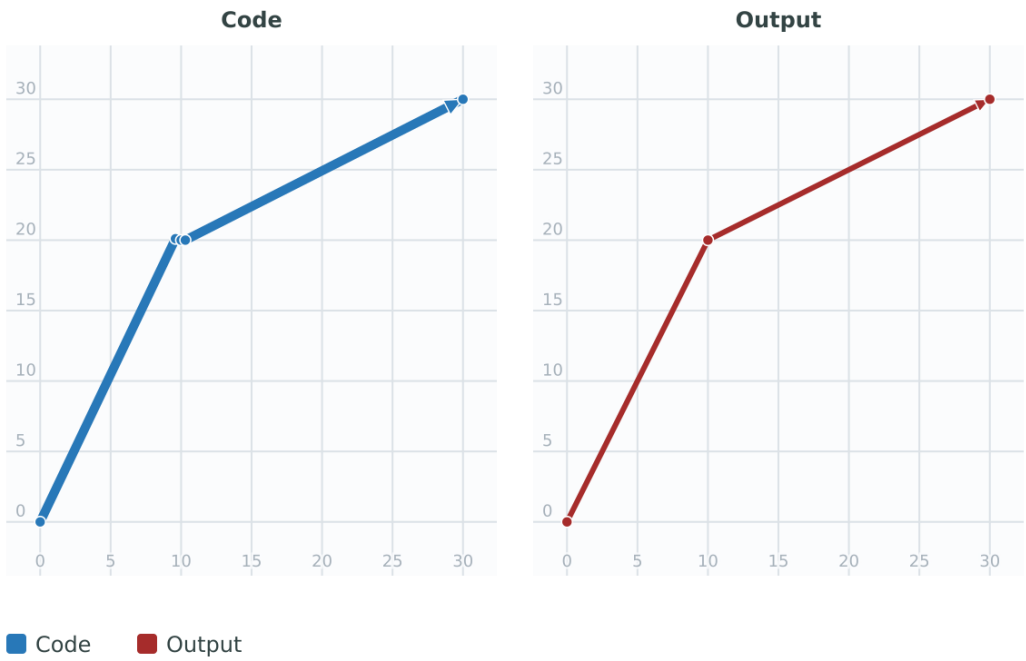
Code

```
SELECT ST_ReducePrecision('LINESTRING (10 10,19.6 30.1,20 30,20.3 30,40 40)', 1);
```

Output

```
LINESTRING (10 10, 20 30, 40 40)
```

Figure



Precision reduction splits polygons if needed to ensure validity

Code

```
WITH data(geom) AS (
  VALUES ('POLYGON ((10 10,60 60.1,70 30,40 40,50 10,10 10))'::geometry)
)
SELECT geom AS input,
       ST_ReducePrecision(geom, 10) AS reduced
FROM data;
```

Output

```
-[ RECORD 1 ]-------------------------------------------------
input   | POLYGON((10 10,60 60.1,70 30,40 40,50 10,10 10))
reduced | MULTIPOLYGON(((10 10,40 40,50 10,10 10)),((40 40,60 60,70 30,40 40)))
```

Figure



See Also

ST_SnapToGrid, ST_Simplify, ST_SimplifyVW

7.14.22 ST_SharedPaths

ST_SharedPaths — Returns a collection containing paths shared by the two input linestrings/multilinestrings.

Synopsis

geometry **ST_SharedPaths**(geometry lineal1, geometry lineal2);

Description

Returns a collection containing paths shared by the two input geometries. Those going in the same direction are in the first element of the collection, those going in the opposite direction are in the second element. The paths themselves are given in the direction of the first geometry.

Performed by the GEOS module.

Availability: 2.0.0

Examples: Finding shared paths

Adjacent shared path segments may be returned as separate or merged LineStrings depending on the GEOS version.

Find the shared paths of a MultiLineString and a LineString.

Code

```
SELECT ST_SharedPaths(ST_GeomFromText('MULTILINESTRING((26 125,26 200,126 200,126 125,26 ↩
    125),
       (51 150,101 150,76 175,51 150))'),
   ST_GeomFromText('LINESTRING(151 100,126 156.25,126 125,101 150,76 175)')
   )
```

Output

```
GEOMETRYCOLLECTION(MULTILINESTRING((126 156.25,126 125),
(101 150,76 175)),MULTILINESTRING EMPTY)
```

Figure



Flip the LineString orientation to return the same paths in the opposite-direction member.

Code

```
SELECT ST_SharedPaths(ST_GeomFromText('LINESTRING(76 175,101 150,126 125,126 156.25,151 100) ↩
    '),
   ST_GeomFromText('MULTILINESTRING((26 125,26 200,126 200,126 125,26 125),
       (51 150,101 150,76 175,51 150))')
    )
```

Output

```
GEOMETRYCOLLECTION(MULTILINESTRING EMPTY,
MULTILINESTRING((76 175,101 150),(126 125,126 156.25)))
```

Figure



See Also

ST_Dump, ST_GeometryN, ST_NumGeometries

7.14.23 ST_Simplify

ST_Simplify — Returns a simplified representation of a geometry, using the Douglas-Peucker algorithm.

Synopsis

geometry **ST_Simplify**(geometry geom, float tolerance);
geometry **ST_Simplify**(geometry geom, float tolerance, boolean preserveCollapsed);

Description

Computes a simplified representation of a geometry using the Douglas-Peucker algorithm. The simplification `tolerance` is a distance value, in the units of the input SRS. Simplification removes vertices which are within the tolerance distance of the simplified linework. The result may not be valid even if the input is.

The function can be called with any kind of geometry (including GeometryCollections), but only line and polygon elements are simplified. Endpoints of linear geometry are preserved.

The `preserveCollapsed` flag retains small geometries that would otherwise be removed at the given tolerance. For example, if a 1m long line is simplified with a 10m tolerance, when `preserveCollapsed` is true the line will not disappear. This flag is useful for rendering purposes, to prevent very small features disappearing from a map.



Note
The returned geometry may lose its simplicity (see ST_IsSimple), topology may not be preserved, and polygonal results may be invalid (see ST_IsValid). Use ST_SimplifyPreserveTopology to preserve topology and ensure validity.



Note
This function does not preserve boundaries shared between polygons. Use ST_CoverageSimplify if this is required.

Availability: 1.2.2

Examples

A circle simplified with a medium tolerance becomes an octagon, and with a large tolerance disappears.

Code

```
SELECT ST_NPoints(geom) AS np_before,
     ST_NPoints(ST_Simplify(geom, 0.1)) AS np01_notbadcircle,
     ST_NPoints(ST_Simplify(geom, 0.5)) AS np05_notquitecircle,
     ST_NPoints(ST_Simplify(geom, 1)) AS np1_octagon,
     ST_NPoints(ST_Simplify(geom, 10)) AS np10_gone,
     (ST_Simplify(geom, 100) is null) AS  np100_geometrygoesaway
FROM (SELECT ST_Buffer('POINT(1 3)', 10, 12) As geom) AS t;
```

Output

```
-[ RECORD 1 ]------------
np_before                | 49
np01_notbadcircle        | 33
np05_notquitecircle      | 17
np1_octagon              | 9
np10_gone                | null
np100_geometrygoesaway   | t
```

Simplifying a set of lines. Lines may intersect after simplification.

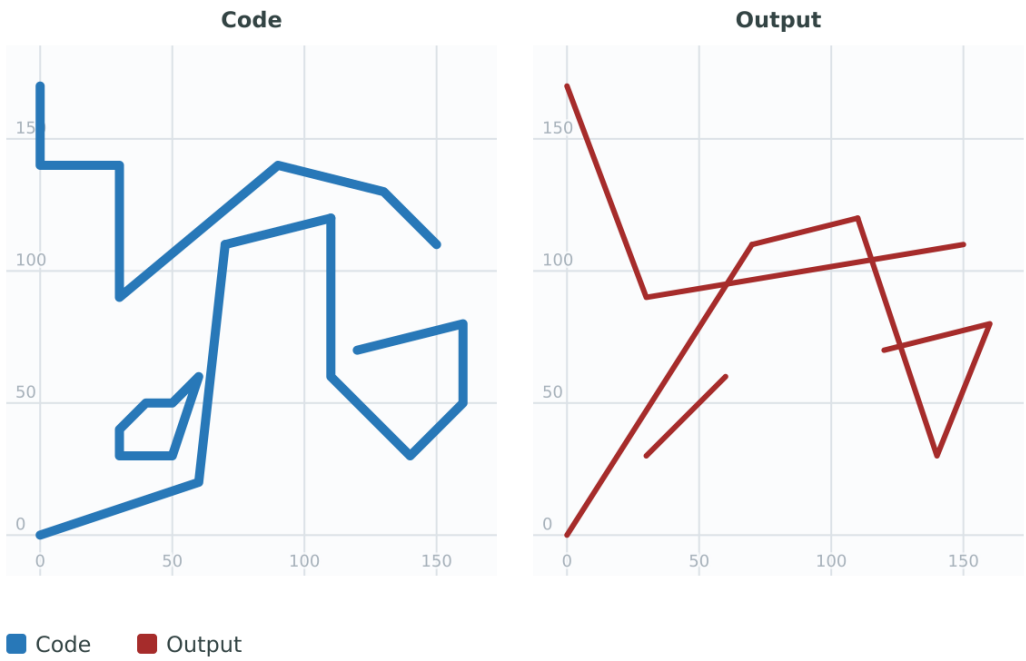
Code

```
SELECT ST_Simplify('MULTILINESTRING ((20 180,20 150,50 150,50 100,110 150,150 140,170 120) ↩
    ,(20 10,80 30,90 120),(90 120,130 130),(130 130,130 70,160 40,180 60,180 90,140 80),(50  ↩
    40,70 40,80 70,70 60,60 60,50 50,50 40))'::geometry,
40);
```

Output

```
MULTILINESTRING((20 180,50 100,170 120),(20 10,90 120),(90 120,130 130),(130 130,160 40,180 90,140 80),(50 40,80 70,50  ↩
    40))
```

Figure



Simplifying a MultiPolygon. Polygonal results may be invalid.

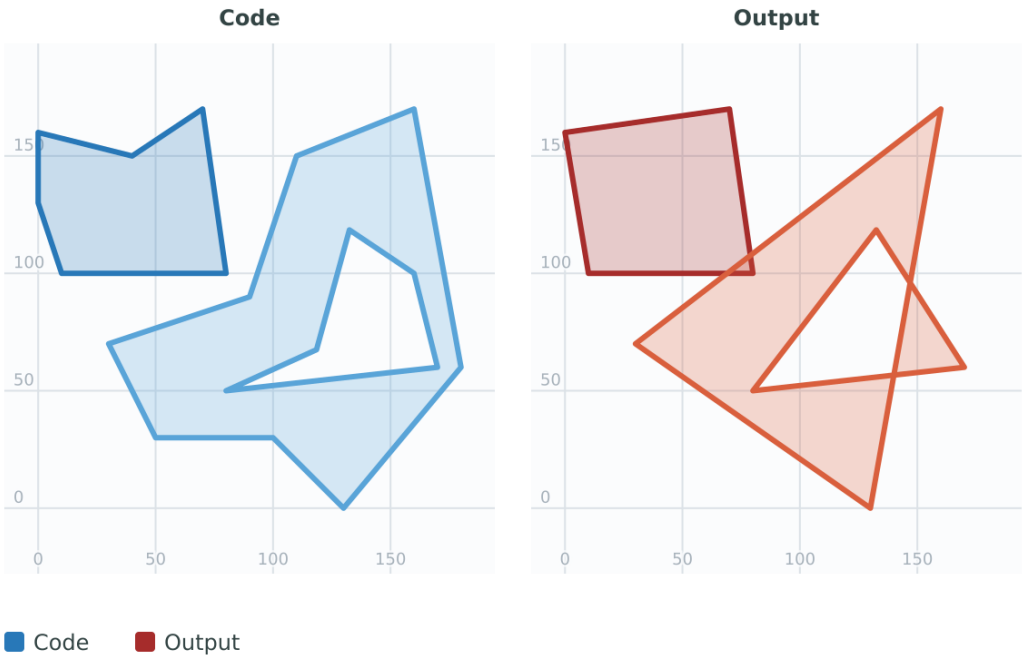
Code

```
SELECT ST_Simplify('MULTIPOLYGON (((90 110,80 180,50 160,10 170,10 140,20 110,90 110)),((40 ↩
    80,100 100,120 160,170 180,190 70,140 10,110 40,60 40,40 80),(180 70,170 110,142.5 ↩
    128.5,128.5 77.5,90 60,180 70)))'::geometry,
40);
```

Output

```
MULTIPOLYGON(((90 110,80 180,10 170,20 110,90 110)),((40 80,170 180,140 10,40 80),(180 70,142.5 128.5,90 60,180 70)))
```

Figure



See Also

ST_IsSimple, ST_SimplifyPreserveTopology, ST_SimplifyVW, ST_CoverageSimplify, Topology ST_Simplify

7.14.24 ST_SimplifyPreserveTopology

ST_SimplifyPreserveTopology — Returns a simplified and valid representation of a geometry, using the Douglas-Peucker algorithm.

Synopsis

geometry **ST_SimplifyPreserveTopology**(geometry geom, float tolerance);

Description

Computes a simplified representation of a geometry using a variant of the Douglas-Peucker algorithm which limits simplification to ensure the result has the same topology as the input. The simplification `tolerance` is a distance value, in the units of the input SRS. Simplification removes vertices which are within the tolerance distance of the simplified linework, as long as topology is preserved. The result will be valid and simple if the input is.

The function can be called with any kind of geometry (including GeometryCollections), but only line and polygon elements are simplified. For polygonal inputs, the result will have the same number of rings (shells and holes), and the rings will not cross. Ring endpoints may be simplified. For linear inputs, the result will have the same number of lines, and lines will not intersect if they did not do so in the original geometry. Endpoints of linear geometry are preserved.



Note
This function does not preserve boundaries shared between polygons. Use ST_CoverageSimplify if this is required.

Performed by the GEOS module.

Availability: 1.3.3

Examples

For the same example as ST_Simplify, ST_SimplifyPreserveTopology prevents the geometry from disappearing.

Code

```
SELECT  ST_NPoints(geom) AS np_before,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 0.1)) AS np01_notbadcircle,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 0.5)) AS np05_notquitecircle,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 1))   AS np1_octagon,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 10))  AS np10_minimal,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 100)) AS np100_minimal
FROM (SELECT ST_Buffer('POINT(1 3)', 10, 12) AS geom) AS t;
```

Output

```
 np_before | np01_notbadcircle | np05_notquitecircle | np1_octagon | np10_minimal |  ↩
   np100_minimal
-----------+-------------------+---------------------+-------------+--------------+--------------- ←

        49 |                33 |                  17 |           9 |            4 |          ↩
        4
```

Simplifying a set of lines, preserving topology of non-intersecting lines.

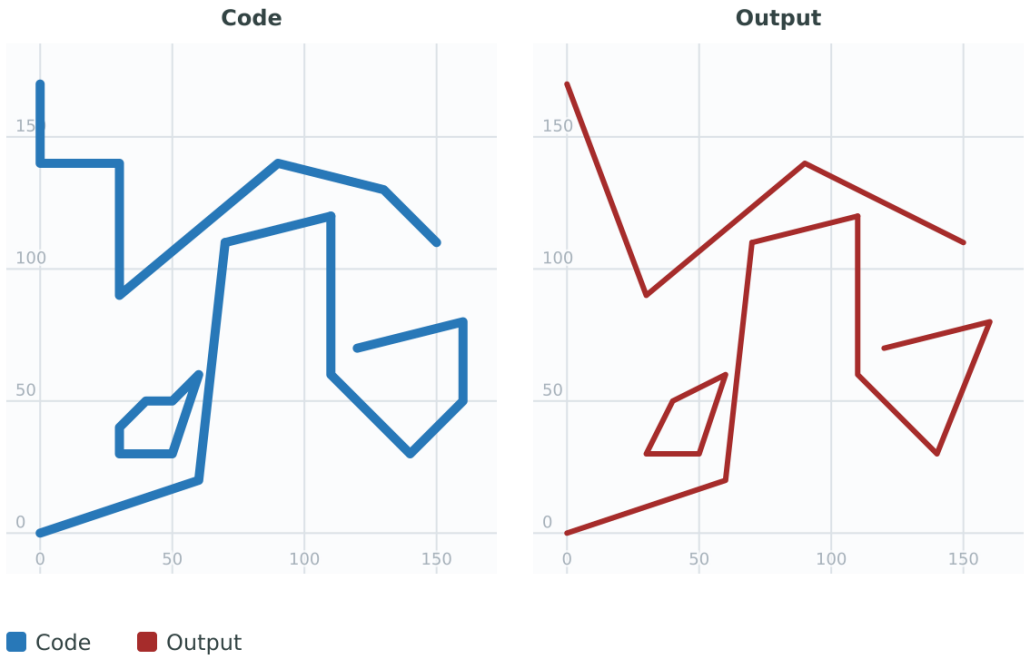
Code

```
SELECT ST_SimplifyPreserveTopology('MULTILINESTRING ((20 180,20 150,50 150,50 100,110  ↩
    150,150 140,170 120),(20 10,80 30,90 120),(90 120,130 130),(130 130,130 70,160 40,180  ↩
    60,180 90,140 80),(50 40,70 40,80 70,70 60,60 60,50 50,50 40))'::geometry,
40);
```

Output

```
MULTILINESTRING((20 180,50 100,110 150,170 120),(20 10,80 30,90 120),(90 120,130 130),(130 130,130 70,160 40,180 90,140 ↩
     80),(50 40,70 40,80 70,60 60,50 40))
```

Figure



Simplifying a MultiPolygon, preserving topology of shells and holes.

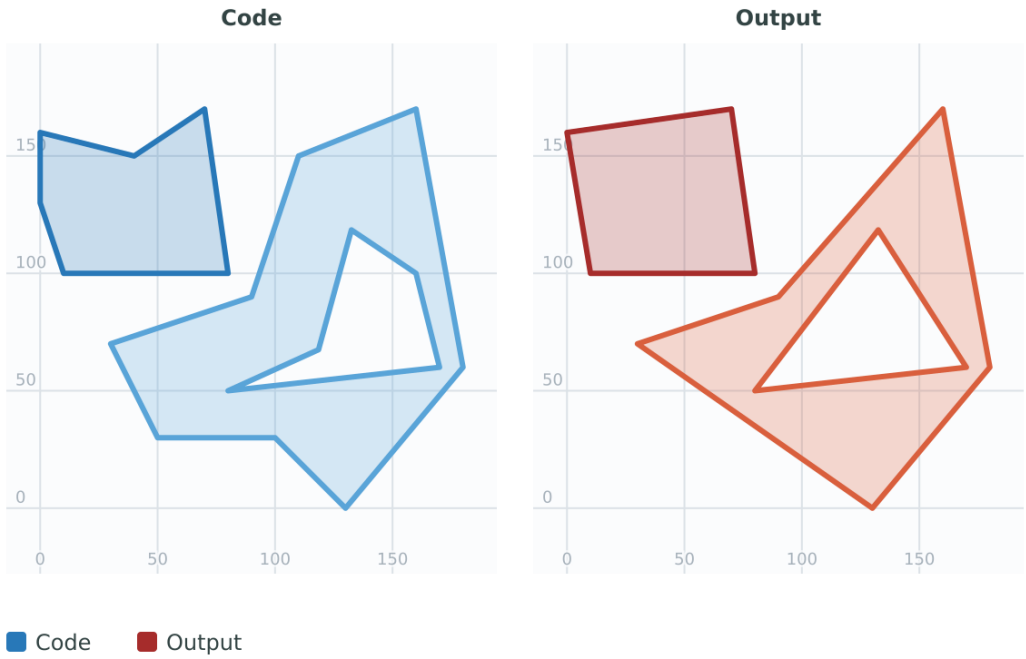
Code

```
SELECT ST_SimplifyPreserveTopology('MULTIPOLYGON (((90 110,80 180,50 160,10 170,10 140,20 ↩
    110,90 110)),((40 80,100 100,120 160,170 180,190 70,140 10,110 40,60 40,40 80),(180 ↩
    70,170 110,142.5 128.5,128.5 77.5,90 60,180 70)))'::geometry,
40);
```

Output

```
MULTIPOLYGON(((90 110,80 180,10 170,20 110,90 110)),((40 80,100 100,170 180,190 70,140 10,40 80),(180 70,142.5 128.5,90 ↩
      60,180 70)))
```

Figure



See Also

ST_Simplify, ST_SimplifyVW, ST_CoverageSimplify

7.14.25 ST_SimplifyPolygonHull

ST_SimplifyPolygonHull — Computes a simplified topology-preserving outer or inner hull of a polygonal geometry.

Synopsis

geometry **ST_SimplifyPolygonHull**(geometry param_geom, float vertex_fraction, boolean is_outer = true);

Description

Computes a simplified topology-preserving outer or inner hull of a polygonal geometry. An outer hull completely covers the input geometry. An inner hull is completely covered by the input geometry. The result is a polygonal geometry formed by a subset of the input vertices. MultiPolygons and holes are handled and produce a result with the same structure as the input.

The reduction in vertex count is controlled by the `vertex_fraction` parameter, which is a number in the range 0 to 1. Lower values produce simpler results, with smaller vertex count and less concaveness. For both outer and inner hulls a vertex fraction of 1.0 produces the original geometry. For outer hulls a value of 0.0 produces the convex hull (for a single polygon); for inner hulls it produces a triangle.

The simplification process operates by progressively removing concave corners that contain the least amount of area, until the vertex count target is reached. It prevents edges from crossing, so the result is always a valid polygonal geometry.

To get better results with geometries that contain relatively long line segments, it might be necessary to "segmentize" the input, as shown below.

Performed by the GEOS module.

Availability: 3.3.0.

Requires GEOS >= 3.11.0.

Examples

Outer hull of a Polygon.

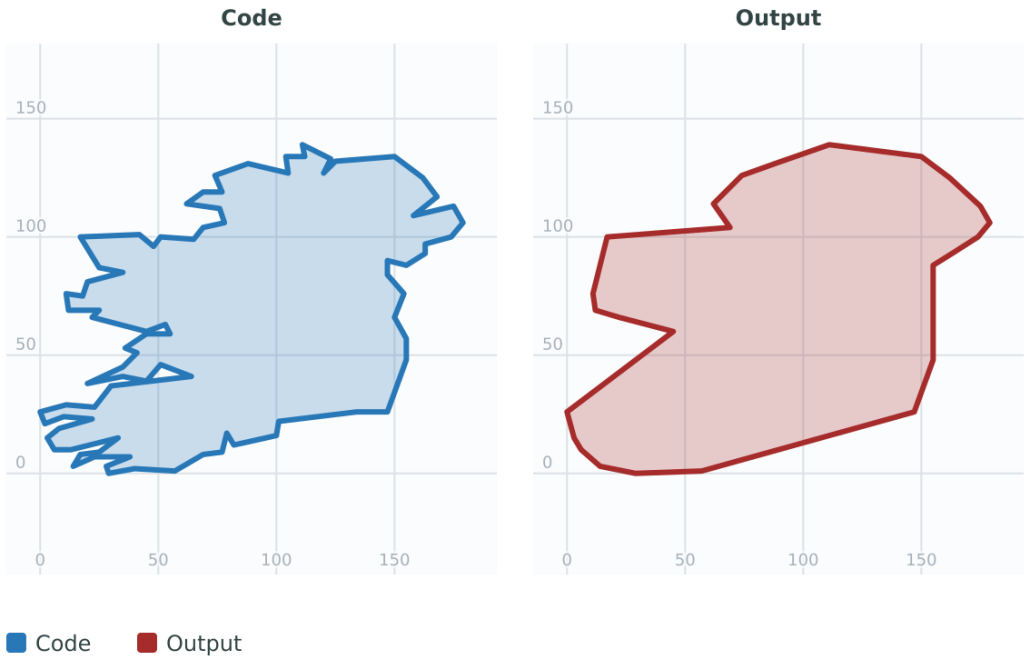
Code

```
SELECT ST_SimplifyPolygonHull(
  'POLYGON ((131 158,136 163,161 165,173 156,179 148,169 140,186 144,190 137,185 131,174 ↩
    128,174 124,166 119,158 121,158 115,165 107,161 97,166 88,166 79,158 57,145 57,112 ↩
    53,111 47,93 43,90 48,88 40,80 39,68 32,51 33,40 31,39 34,49 38,34 38,25 34,28 39,36 ↩
    40,44 46,24 41,17 41,14 46,19 50,33 54,21 55,13 52,11 57,22 60,34 59,41 68,75 72,62 ↩
    77,56 70,46 72,31 69,46 76,52 82,47 84,56 90,66 90,64 94,56 91,33 97,36 100,23 100,22 ↩
    107,29 106,31 112,46 116,36 118,28 131,53 132,59 127,62 131,76 130,80 135,89 137,87 ↩
    143,73 145,80 150,88 150,85 157,99 162,116 158,115 165,123 165,122 170,134 164,131 158)) ↩
    '::geometry,
  0.3
);
```

Output

```
POLYGON((161 165,173 156,186 144,190 137,185 131,174 124,166 119,166 79,158 57,68 32,40 31,25 34,17 41,14 46,11 57,56 ↩
    91,33 97,23 100,22 107,28 131,80 135,73 145,85 157,99 162,122 170,161 165))
```

Figure



Inner hull of a Polygon.

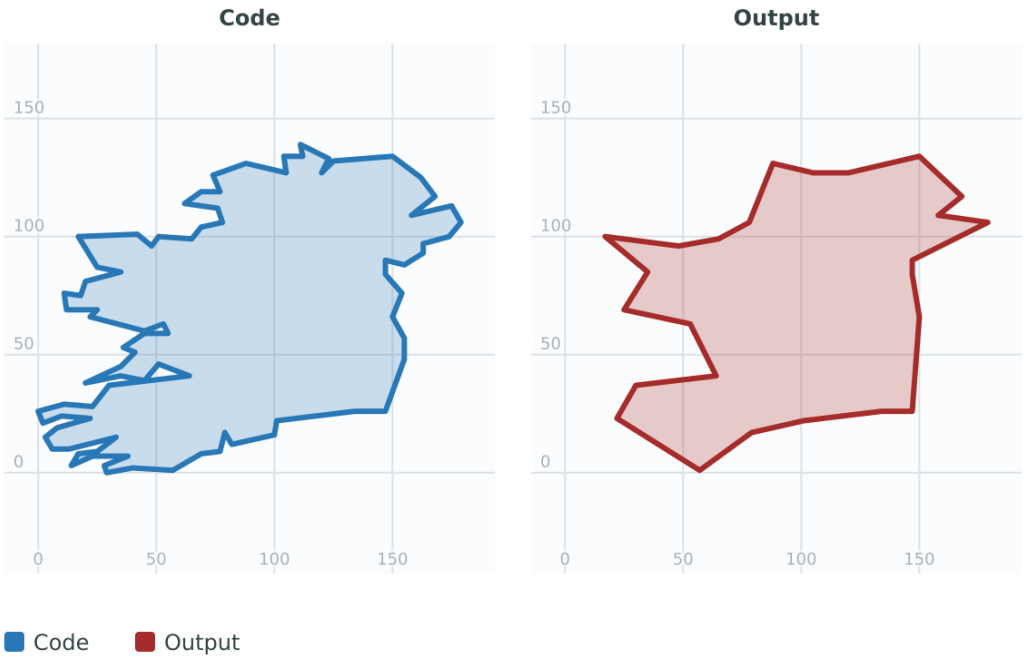
Code

```
SELECT ST_SimplifyPolygonHull(
  'POLYGON ((131 158,136 163,161 165,173 156,179 148,169 140,186 144,190 137,185 131,174 ↩
    128,174 124,166 119,158 121,158 115,165 107,161 97,166 88,166 79,158 57,145 57,112 ↩
    53,111 47,93 43,90 48,88 40,80 39,68 32,51 33,40 31,39 34,49 38,34 38,25 34,28 39,36 ↩
    40,44 46,24 41,17 41,14 46,19 50,33 54,21 55,13 52,11 57,22 60,34 59,41 68,75 72,62 ↩
    77,56 70,46 72,31 69,46 76,52 82,47 84,56 90,66 90,64 94,56 91,33 97,36 100,23 100,22 ↩
    107,29 106,31 112,46 116,36 118,28 131,53 132,59 127,62 131,76 130,80 135,89 137,87 ↩
    143,73 145,80 150,88 150,85 157,99 162,116 158,115 165,123 165,122 170,134 164,131 158)) ↩
    '::geometry,
  0.3, false
);
```

Output

```
POLYGON((131 158,116 158,99 162,89 137,76 130,59 127,28 131,46 116,36 100,64 94,75 72,41 68,33 54,68 32,90 48,112 ↩
    53,145 57,158 57,161 97,158 115,158 121,190 137,169 140,179 148,161 165,131 158))
```

Figure

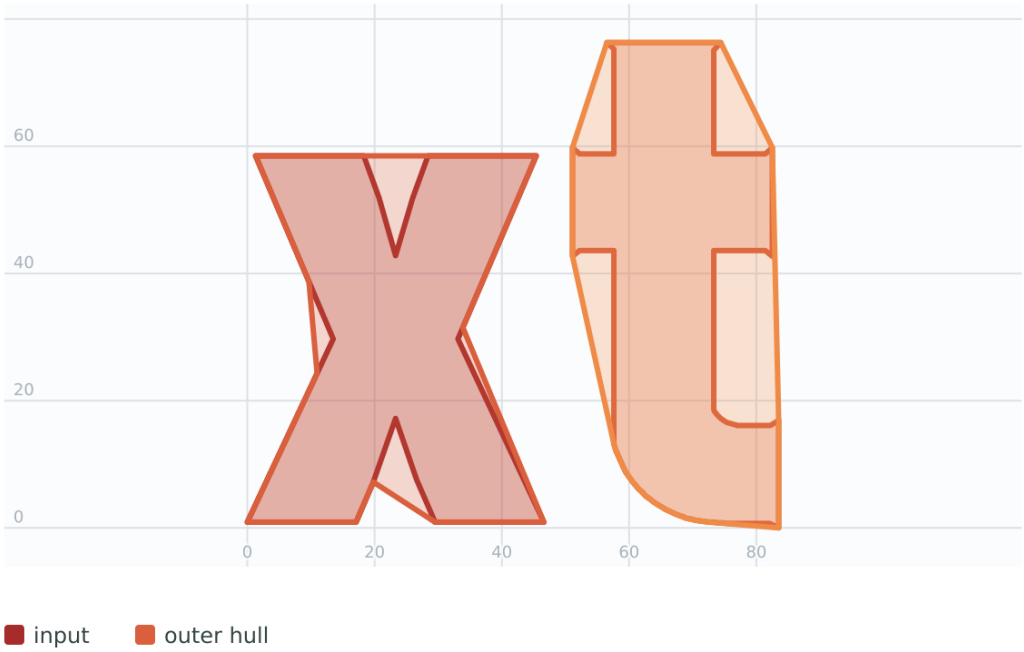


Outer hull simplification of a MultiPolygon, with segmentization.

Code

```
WITH letters AS (
  SELECT ST_Letters('xt') AS geom
)
SELECT geom AS input,
  ST_SimplifyPolygonHull(ST_Segmentize(geom, 2.0), 0.1) AS outer_hull
FROM letters;
```

Figure



See Also

ST_ConvexHull, ST_SimplifyVW, ST_ConcaveHull, ST_Segmentize

7.14.26 ST_SimplifyVW

ST_SimplifyVW — Returns a simplified representation of a geometry, using the Visvalingam-Whyatt algorithm

Synopsis

geometry **ST_SimplifyVW**(geometry geom, float tolerance);

Description

Returns a simplified representation of a geometry using the Visvalingam-Whyatt algorithm. The simplification `tolerance` is an area value, in squared units of the input SRS. Simplification removes vertices which form "corners" with area less than the tolerance. The result may not be valid even if the input is.

The function can be called with any kind of geometry (including GeometryCollections), but only line and polygon elements are simplified. Endpoints of linear geometry are preserved.



Note
The returned geometry may lose its simplicity (see ST_IsSimple), topology may not be preserved, and polygonal results may be invalid (see ST_IsValid). Use ST_SimplifyPreserveTopology to preserve topology and ensure validity. ST_CoverageSimplify also preserves topology and validity.

This is a general-purpose simplification function. It may remove narrow polygon spikes when their effective area is below the tolerance, but it is not a dedicated spike-removal or polygon-repair operation. Use ST_IsValid to check the result and ST_MakeValid to repair invalid polygonal output if needed.

This function does not preserve boundaries shared between polygons. Use ST_CoverageSimplify if this is required.

This function handles 3D and the third dimension will affect the result.

Availability: 2.2.0

Examples

This polygon has a narrow spike whose effective area is small enough to be removed by a tolerance of 20 square units.

Code

```
SELECT geom AS original, ST_SimplifyVW(geom, 20) AS simplified
FROM (SELECT 'POLYGON((0 0,10 0,10 10,5.2 10,5 16,4.8 10,0 10,0 0))'::geometry AS geom) AS t ↩
    ;
```

Output

```
POLYGON((0 0,10 0,10 10,5.2 10,5 16,4.8 10,0 10,0 0)) | POLYGON((0 0,10 0,10 10,0 10,0 0))
```

Figure



A LineString is simplified with a minimum-area tolerance of 30. For a linear simplification using ST_Simplify, a comparable tolerance would be a length in input-SRS units, so the value here is roughly that linear tolerance squared.

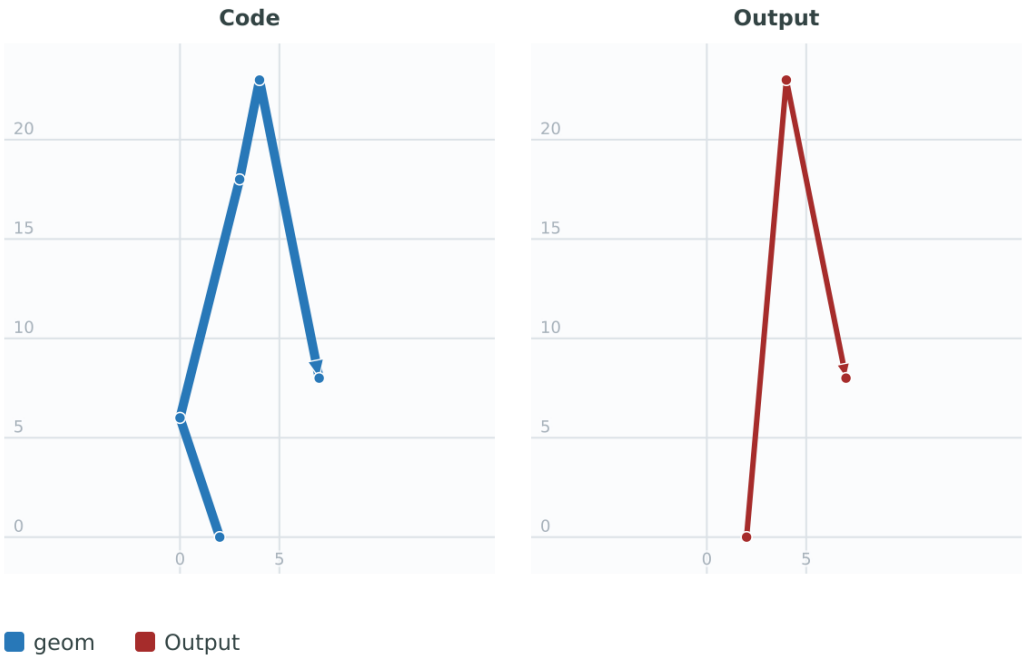
Code

```
SELECT ST_SimplifyVW(geom, 30) simplified
  FROM (SELECT 'LINESTRING(5 2,3 8,6 20,7 25,10 10)'::geometry AS geom) AS t;
```

Output

```
LINESTRING(5 2,7 25,10 10)
```

Figure



Simplifying a line.

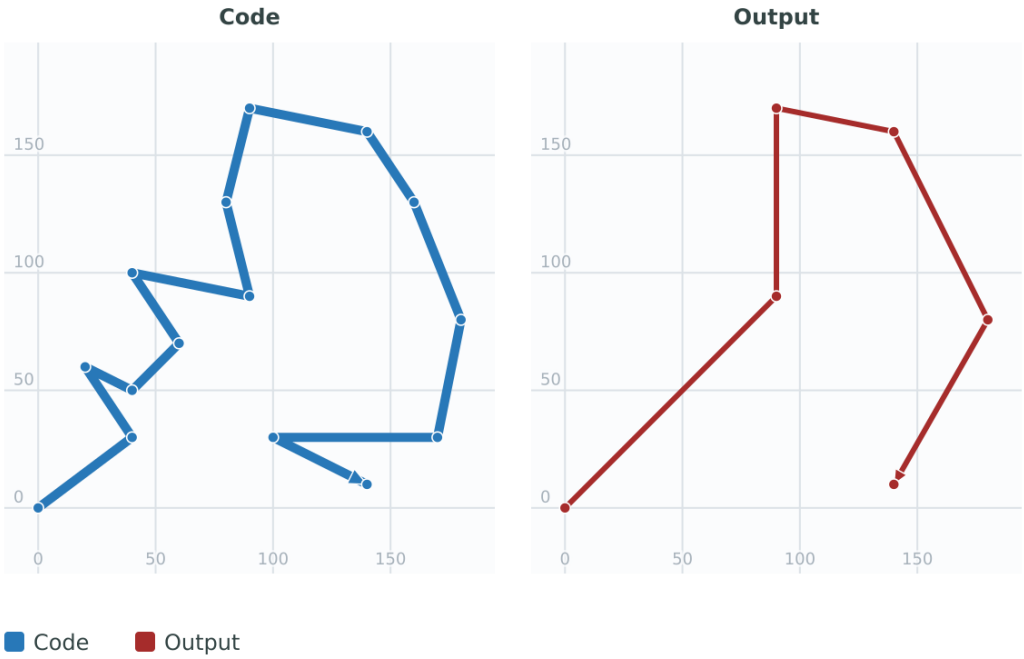
Code

```
SELECT ST_SimplifyVW('LINESTRING (10 10,50 40,30 70,50 60,70 80,50 110,100 100,90 140,100 ↩
    180,150 170,170 140,190 90,180 40,110 40,150 20)'::geometry,
1600);
```

Output

```
LINESTRING(10 10,100 100,100 180,150 170,190 90,150 20)
```

Figure



Simplifying a polygon.

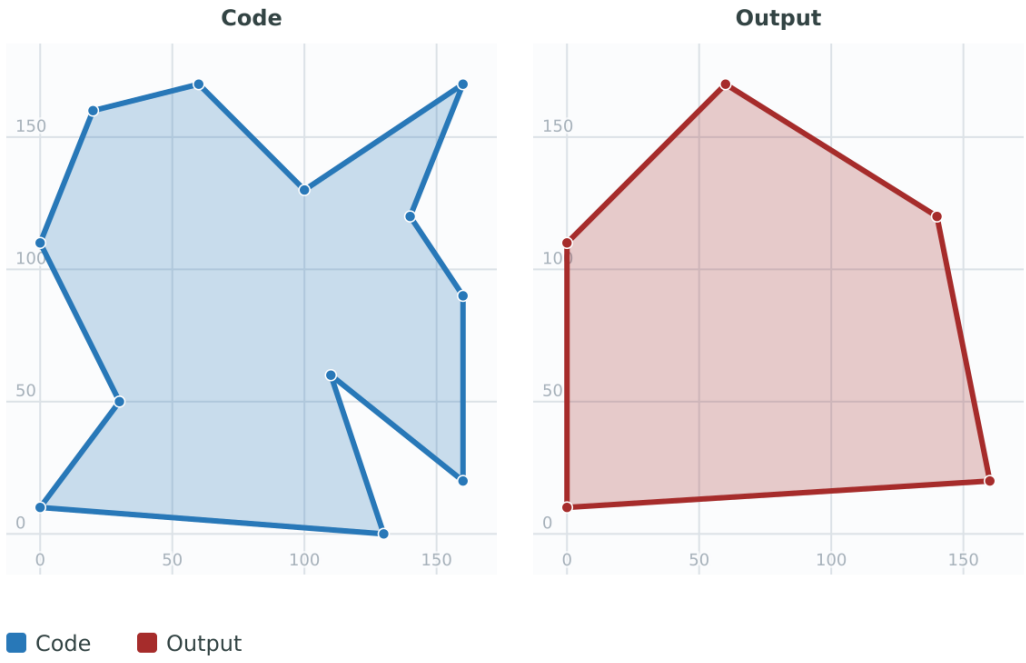
Code

```
SELECT ST_SimplifyVW('POLYGON ((20 20,50 60,20 120,40 170,80 180,120 140,180 180,160 130,180 ↩
    100,180 30,130 70,150 10,20 20))'::geometry,
1600);
```

Output

```
POLYGON((20 20,20 120,80 180,160 130,180 30,20 20))
```

Figure



See Also

ST_SetEffectiveArea, ST_Simplify, ST_SimplifyPreserveTopology, ST_CoverageSimplify, Topology ST_Simplify

7.14.27 ST_SetEffectiveArea

ST_SetEffectiveArea — Sets the effective area for each vertex, using the Visvalingam-Whyatt algorithm.

Synopsis

geometry **ST_SetEffectiveArea**(geometry geom, float threshold = 0, integer set_area = 1);

Description

Sets the effective area for each vertex, using the Visvalingam-Whyatt algorithm. The effective area is stored as the M-value of the vertex. If the optional "threshold" parameter is used, a simplified geometry will be returned, containing only vertices with an effective area greater than or equal to the threshold value.

This function can be used for server-side simplification when a threshold is specified. Another option is to use a threshold value of zero. In this case, the full geometry will be returned with effective areas as M-values, which can be used by the client to simplify very quickly.

Will actually do something only with (multi)lines and (multi)polygons but you can safely call it with any kind of geometry. Since simplification occurs on a object-by-object basis you can also feed a GeometryCollection to this function.



Note
Note that returned geometry might lose its simplicity (see ST_IsSimple)



Note
Note topology may not be preserved and may result in invalid geometries. Use (see ST_SimplifyPreserveTopology) to preserve topology.



Note
The output geometry will lose all previous information in the M-values



Note
This function handles 3D and the third dimension will affect the effective area

Availability: 2.2.0

Examples

Calculating the effective area of a LineString. Because we use a threshold value of zero, all vertices in the input geometry are returned.

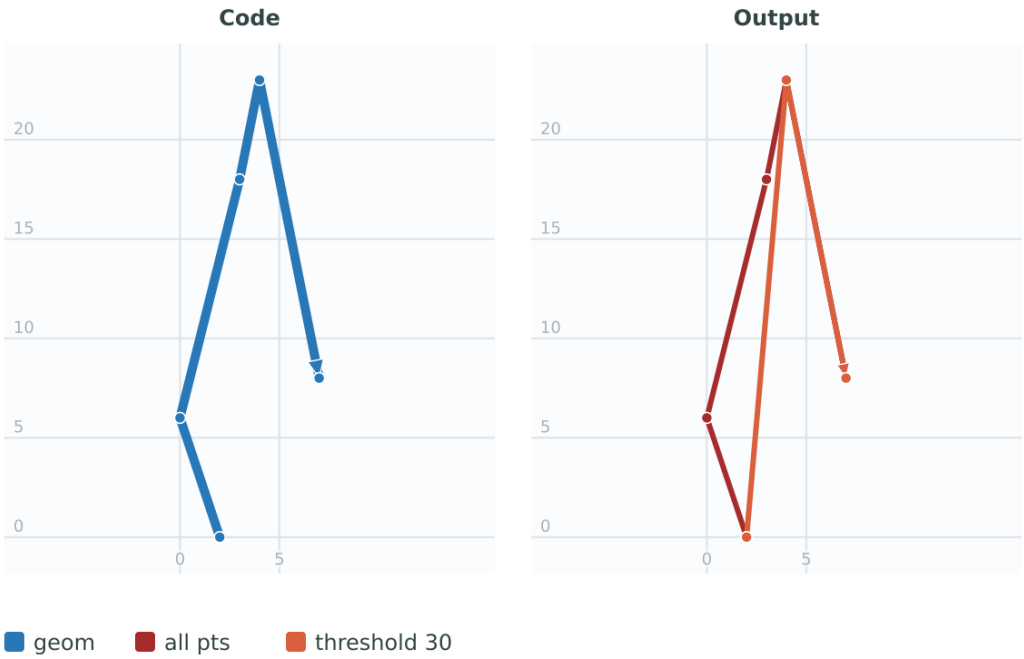
Code

```
SELECT ST_SetEffectiveArea(geom) AS all_pts,
       ST_SetEffectiveArea(geom, 30) AS threshold_30
FROM (SELECT  'LINESTRING(5 2,3 8,6 20,7 25,10 10)'::geometry geom) As foo;
```

Output

```
-[ RECORD 1 ]------------------
all_pts      | LINESTRING M (5 2 3.4e+38,3 8 29,6 20 1.5,7 25 49.5,10 10 3.4e+38)
threshold_30 | LINESTRING M (5 2 3.4e+38,7 25 49.5,10 10 3.4e+38)
```

Figure



See Also

ST_SimplifyVW

7.14.28 ST_TriangulatePolygon

ST_TriangulatePolygon — Computes the constrained Delaunay triangulation of polygons

Synopsis

geometry **ST_TriangulatePolygon**(geometry geom);

Description

Computes the constrained Delaunay triangulation of polygons. Holes and Multipolygons are supported.

The "constrained Delaunay triangulation" of a polygon is a set of triangles formed from the vertices of the polygon, and covering it exactly, with the maximum total interior angle over all possible triangulations. It provides the "best quality" triangulation of the polygon.

Availability: 3.3.0.

Requires GEOS >= 3.11.0.

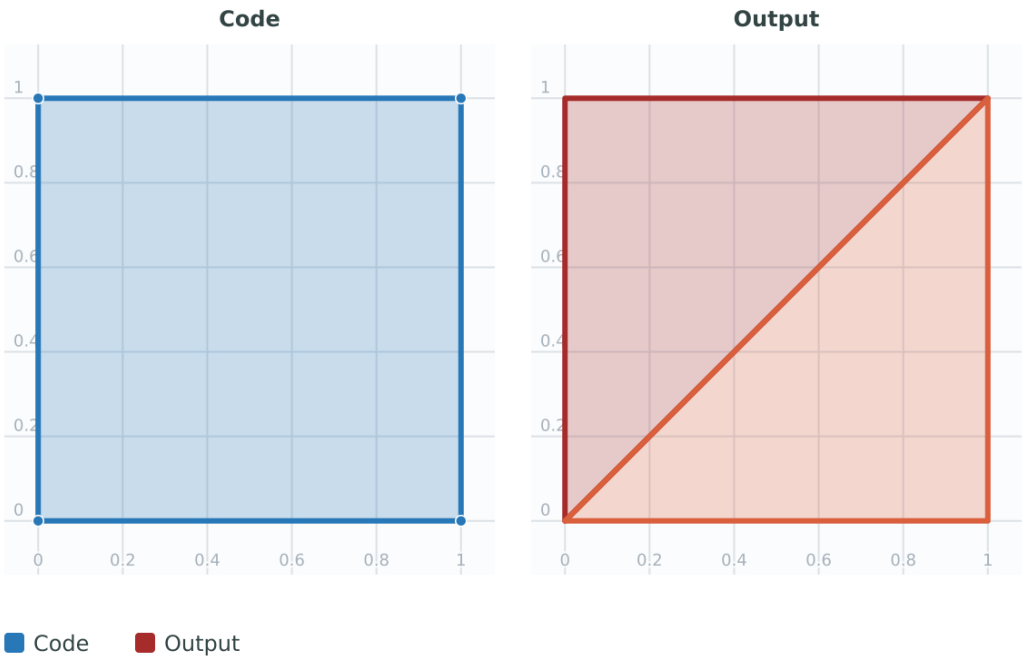
Examples

Triangulation of a square.

Code

```
SELECT ST_TriangulatePolygon('POLYGON((0 0,0 1,1 1,1 0,0 0))');
```

Figure

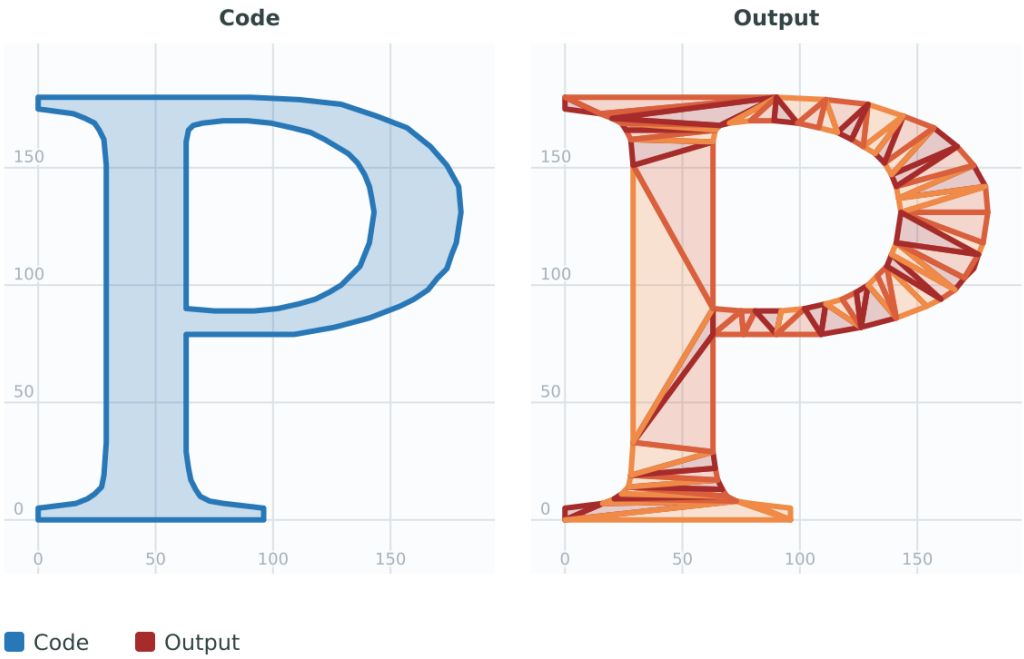


Triangulation of the letter P.

Code

```
SELECT ST_TriangulatePolygon('POLYGON ((26 17,31 19,34 21,37 24,38 29,39 43,39 161,38 172,36 ↩
     176,34 179,30 181,25 183,10 185,10 190,100 190,121 189,139 187,154 182,167 177,177  ↩
    169,184 161,189 152,190 141,188 128,186 123,184 117,180 113,176 108,170 104,164 101,151  ↩
    96,136 92,119 89,100 89,86 89,73 89,73 39,74 32,75 27,77 23,79 20,83 18,89 17,106 15,106 ↩
     10,10 10,10 15,26 17),(152 147,151 152,149 157,146 162,142 166,137 169,132 172,126  ↩
    175,118 177,109 179,99 180,89 180,80 179,76 178,74 176,73 171,73 100,85 99,91 99,102  ↩
    99,112 100,121 102,128 104,134 107,139 110,143 114,147 118,149 123,151 128,153 141,152  ↩
    147))'
);
```

Figure

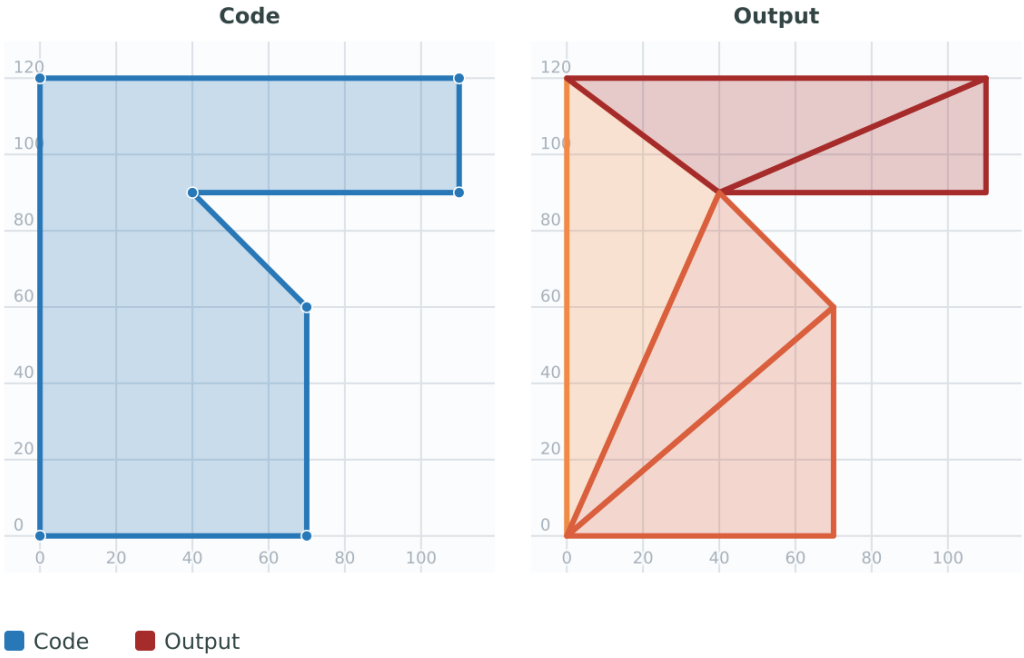


Triangulation of the polygon used in the CG_Tesselate example.

Code

```
SELECT ST_TriangulatePolygon('POLYGON (( 10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 ↩
    190 ))'::geometry
);
```

Figure



See Also

CG_ConstrainedDelaunayTriangles, ST_DelaunayTriangles, CG_Tesselate

7.14.29 ST_VoronoiLines

ST_VoronoiLines — Returns the boundaries of the Voronoi diagram of the vertices of a geometry.

Synopsis

geometry **ST_VoronoiLines**(geometry geom , float8 tolerance = 0.0 , geometry extend_to = NULL);

Description

Computes a two-dimensional Voronoi diagram from the vertices of the supplied geometry and returns the boundaries between cells in the diagram as a MultiLineString. Returns null if input geometry is null. Returns an empty geometry collection if the input geometry contains only one vertex. Returns an empty geometry collection if the `extend_to` envelope has zero area.

Optional parameters:

- `tolerance`: The distance within which vertices will be considered equivalent. Robustness of the algorithm can be improved by supplying a nonzero tolerance distance. (default = 0.0)
- `extend_to`: If present, the diagram is extended to cover the envelope of the supplied geometry, unless smaller than the default envelope (default = NULL, default envelope is the bounding box of the input expanded by about 50%).

Performed by the GEOS module.

Availability: 2.3.0

Examples

Voronoi diagram lines with a tolerance of 30 units.

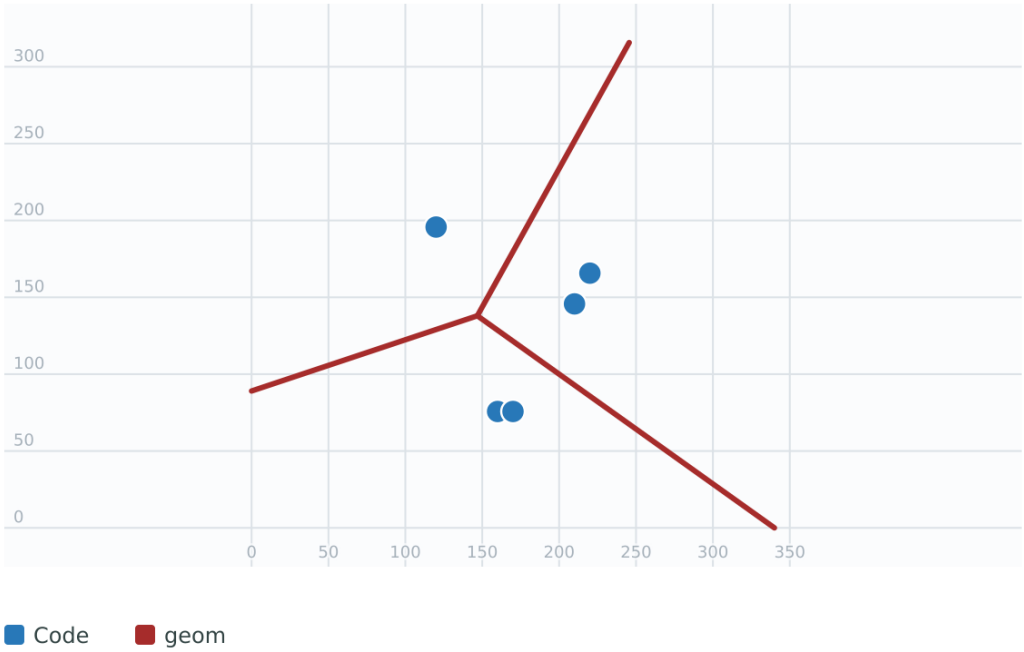
Code

```
SELECT ST_VoronoiLines('MULTIPOINT (50 30,60 30,100 100,10 150,110 120)'::geometry,
            30) AS geom;
```

Output

```
MULTILINESTRING((135.555555555556 270,36.8181818181818 92.2727272727273),(36.8181818181818 92.2727272727273,-110 ↩
    43.3333333333333),(230 -45.7142857142858,36.8181818181818 92.2727272727273))
```

Figure



See Also

ST_DelaunayTriangles, ST_VoronoiPolygons

7.14.30 ST_VoronoiPolygons

ST_VoronoiPolygons — Returns the cells of the Voronoi diagram of the vertices of a geometry.

Synopsis

geometry **ST_VoronoiPolygons**(geometry geom , float8 tolerance = 0.0 , geometry extend_to = NULL);

Description

Computes a two-dimensional Voronoi diagram from the vertices of the supplied geometry. The result is a GEOMETRYCOLLECTION of POLYGONs that covers an envelope larger than the extent of the input vertices. Returns null if input geometry is null. Returns an empty geometry collection if the input geometry contains only one vertex. Returns an empty geometry collection if the `extend_to` envelope has zero area.

Optional parameters:

- `tolerance`: The distance within which vertices will be considered equivalent. Robustness of the algorithm can be improved by supplying a nonzero tolerance distance. (default = 0.0)
- `extend_to`: If present, the diagram is extended to cover the envelope of the supplied geometry, unless smaller than the default envelope (default = NULL, default envelope is the bounding box of the input expanded by about 50%).

Performed by the GEOS module.

Availability: 2.3.0

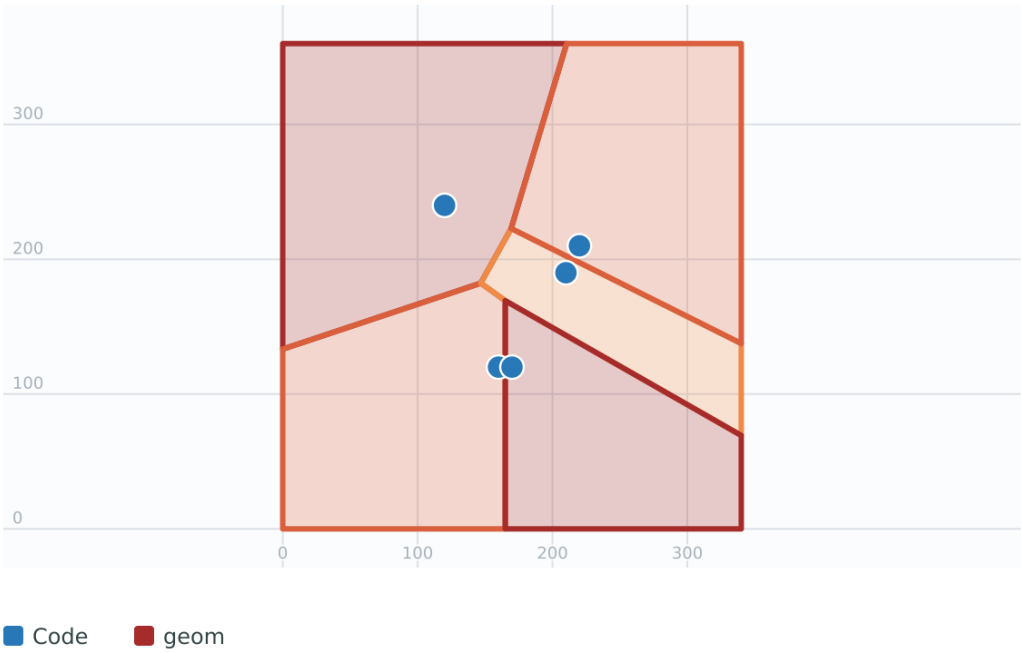
Examples

Points overlaid on top of the Voronoi diagram.

Code

```
SELECT ST_VoronoiPolygons('MULTIPOINT (50 30,60 30,100 100,10 150,110 120)'::geometry
            ) AS geom;
```

Figure

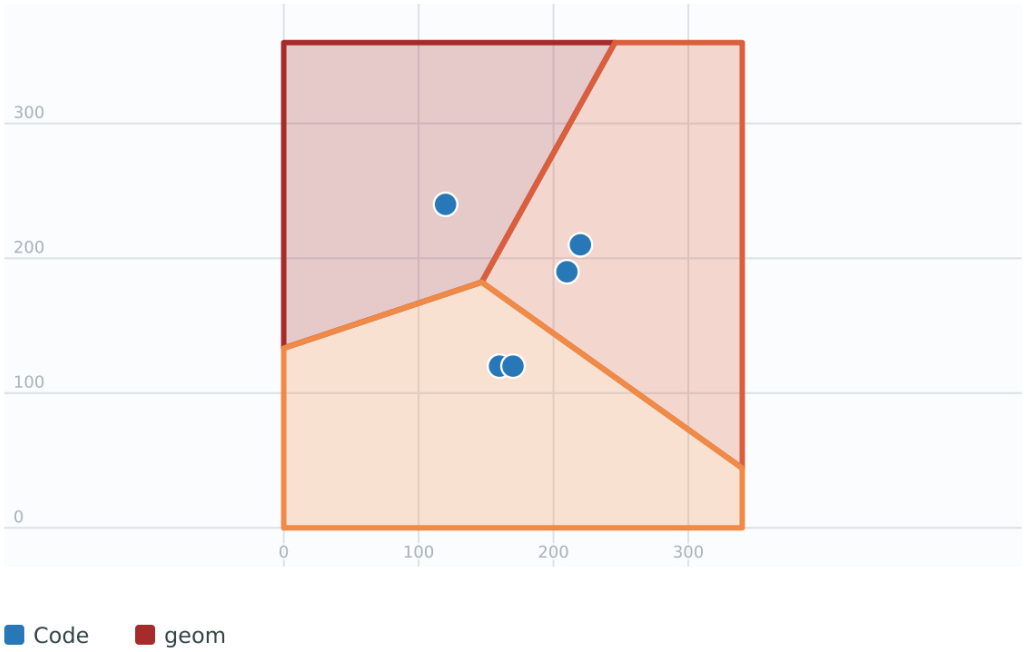


Voronoi diagram with a tolerance of 30 units.

Code

```
SELECT ST_VoronoiPolygons('MULTIPOINT (50 30,60 30,100 100,10 150,110 120)'::geometry,
            30) AS geom;
```

Figure



See Also

ST_DelaunayTriangles, ST_VoronoiLines

7.15 Coverages

7.15.1 ST_CoverageInvalidEdges

ST_CoverageInvalidEdges — Window function that finds locations where polygons fail to form a valid coverage.

Synopsis

geometry **ST_CoverageInvalidEdges**(geometry winset geom, float8 tolerance = 0);

Description

A window function which checks if the polygons in the window partition form a valid polygonal coverage. It returns linear indicators showing the location of invalid edges (if any) in each polygon.

A set of valid polygons is a valid coverage if the following conditions hold:

- **Non-overlapping** - polygons do not overlap (their interiors do not intersect)
- **Edge-Matched** - vertices along shared edges are identical

As a window function a value is returned for every input polygon. For polygons which violate one or more of the validity conditions the return value is a MULTILINESTRING containing the problematic edges. Coverage-valid polygons return the value NULL. Non-polygonal or empty geometries also produce NULL values.

The conditions allow a valid coverage to contain holes (gaps between polygons), as long as the surrounding polygons are edge-matched. However, very narrow gaps are often undesirable. If the *tolerance* parameter is specified with a non-zero distance, edges forming narrower gaps will also be returned as invalid.

The polygons being checked for coverage validity must also be valid geometries. This can be checked with ST_IsValid.

Availability: 3.4.0

Requires GEOS >= 3.12.0

Examples

Invalid edges caused by overlap and non-matching vertices.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((10 190,30 160,40 110,100 70,120 10,10 10,10 190))'::geometry),
  (2, 'POLYGON ((100 190,10 190,30 160,40 110,50 80,74 110.5,100 130,140 120,140 160,100 ↩
    190))'::geometry),
  (3, 'POLYGON ((140 190,190 190,190 80,140 80,140 190))'::geometry),
  (4, 'POLYGON ((180 40,120 10,100 70,140 80,190 80,180 40))'::geometry)
)
SELECT id, ST_CoverageInvalidEdges(geom) OVER ()
  FROM coverage;
```

Figure



This example tests the entire table for coverage validity.

Code

```
SELECT true = ALL (
    SELECT ST_CoverageInvalidEdges(geom) OVER () IS NULL
    FROM coverage
    );
```

See Also

ST_IsValid, ST_CoverageUnion, ST_CoverageClean, ST_CoverageSimplify

7.15.2 ST_CoverageClean

ST_CoverageClean — Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.

Synopsis

geometry **ST_CoverageClean**(geometry winset geom, float8 gapMaximumWidth = 0, float8 snappingDistance = -1, text overlapMergeStrategy = 'MERGE_LONGEST_BORDER');

Description

A window function which adjusts the edges of a set of valid polygonal geometries to produce a clean coverage. Cleaning involves:

- snapping vertices and edges to remove small discrepancies and ensure common edges are identically noded
- merging overlaps into a parent polygon
- merging narrow gaps into adjacent polygons

gapMaximumWidth controls which gaps between polygons are merged. Gaps with width <= this distance are merged into an adjacent polygon.

snappingDistance controls snapping of vertices and edges. The default (-1) automatically determines a snapping distance based on the input extent. Set to 0.0 to turn off snapping.

overlapMergeStrategy specifies how overlaps are merged into a parent polygon:

- MERGE_LONGEST_BORDER - merges into polygon with longest common border
- MERGE_MAX_AREA - merges into polygon with maximum area
- MERGE_MIN_AREA - merges into polygon with minimum area
- MERGE_MIN_INDEX - merges into polygon with smallest input index (defined by order of input polygons)

The result is a clean polygonal coverage that will pass validation by ST_CoverageInvalidEdges and can be input to coverage processing functions.



Note
To aid in determining a maximum gap width, gaps can be computed by cleaning with `gapMaximumWidth => 0` and using ST_CoverageUnion to union the result coverage. Holes in the union correspond to gaps in the original dataset. Gap widths can be measured by extracting the holes as polygons and running ST_MaximumInscribedCircle on them; the gap width is twice the computed radius.

Availability: 3.6.0 - requires GEOS >= 3.14.0

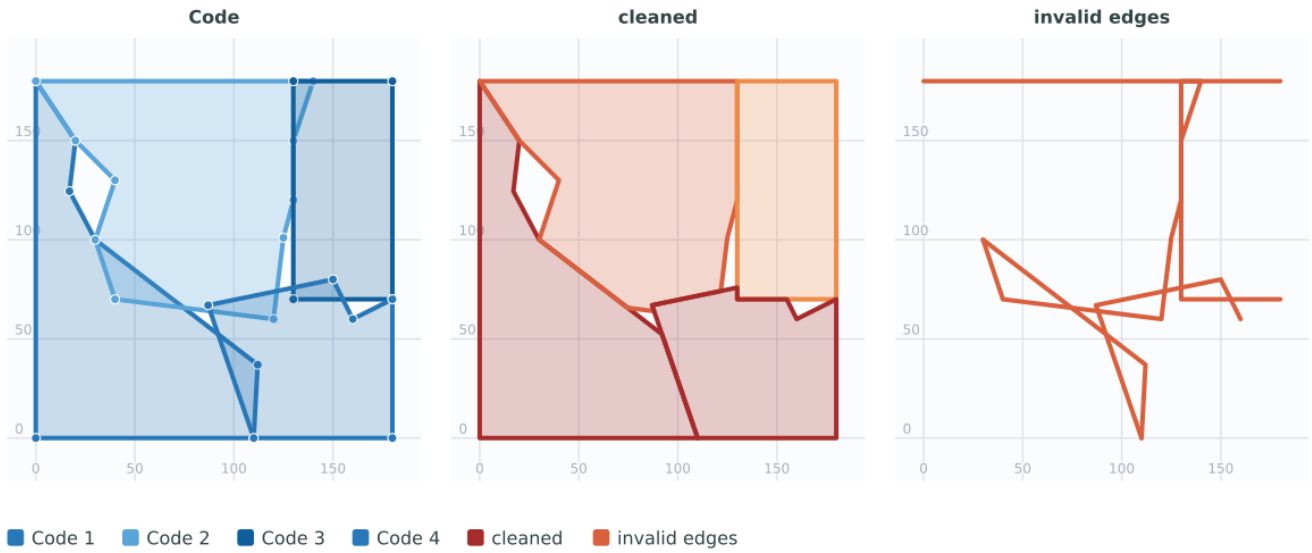
Examples

Clean a coverage containing overlaps and narrow gaps, and compare the cleaned polygons with the original invalid edges.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((10 190,30 160,27 134.5,40 110,122 47,120 10,10 10,10 190))'::geometry),
  (2, 'POLYGON ((150 190,10 190,30 160,50 140,40 110,50 80,130 70,135 111,140 130,140 ↩
    160,150 190))'::geometry),
  (3, 'POLYGON ((140 190,190 190,190 80,140 80,140 190))'::geometry),
  (4, 'POLYGON ((190 10,120 10,97 77,160 90,170 70,190 80,190 10))'::geometry)
), processed AS (
  SELECT id,
         ST_CoverageClean(geom, 1) OVER () AS cleaned,
         ST_CoverageInvalidEdges(geom) OVER () AS invalid_edges
  FROM coverage
)
SELECT ST_Collect(cleaned ORDER BY id) AS cleaned,
       ST_Collect(invalid_edges ORDER BY id) AS invalid_edges
FROM processed;
```

Figure



See Also

ST_CoverageInvalidEdges, ST_CoverageUnion, ST_CoverageSimplify, ST_MaximumInscribedCircle

7.15.3 ST_CoverageSimplify

ST_CoverageSimplify — Window function that simplifies the edges of a polygonal coverage.

Synopsis

geometry **ST_CoverageSimplify**(geometry winset geom, float8 tolerance, boolean simplifyBoundary = true);

Description

A window function which simplifies the edges of polygons in a polygonal coverage. The simplification preserves the coverage topology. This means the simplified output polygons are consistent along shared edges, and still form a valid coverage.

The simplification uses a variant of the Visvalingam–Whyatt algorithm. The *tolerance* parameter has units of distance, and is roughly equal to the square root of triangular areas to be simplified.

To simplify only the "internal" edges of the coverage (those that are shared by two polygons) set the *simplifyBoundary* parameter to false.

Use this function to simplify adjacent polygon layers when shared boundaries must remain edge-matched. For a single geometry where only self-intersection and ring validity matter, see ST_SimplifyPreserveTopology.



Note
If the input is not a valid coverage there may be unexpected artifacts in the output (such as boundary intersections, or separated boundaries which appeared to be shared). Use ST_CoverageInvalidEdges to determine if a coverage is valid.

Availability: 3.4.0

Requires GEOS >= 3.12.0

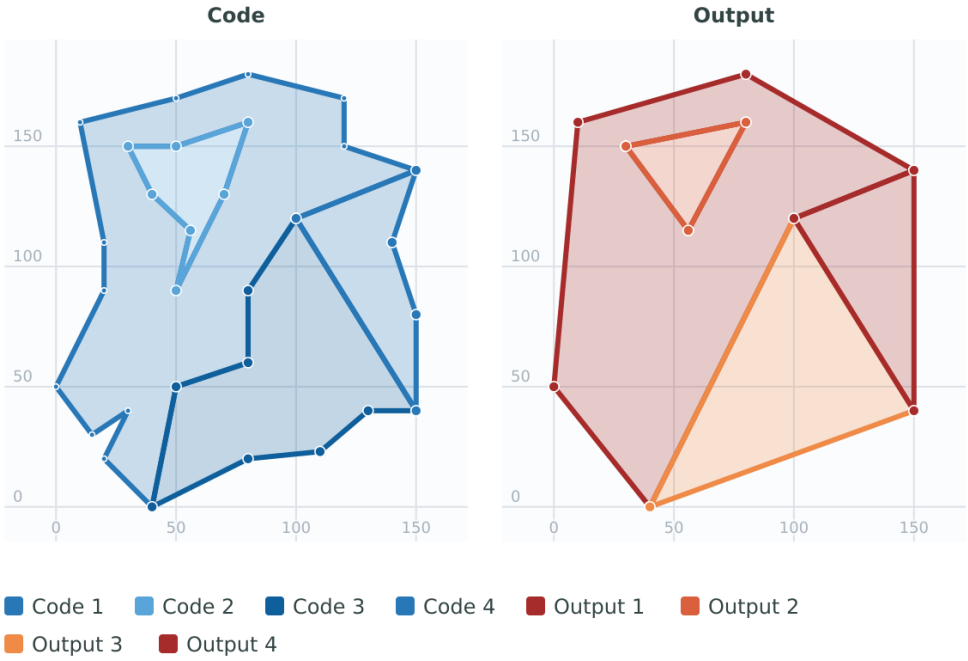
Examples

Simplify a polygonal coverage while preserving shared boundaries.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((160 150,110 130,90 100,90 70,60 60,50 10,30 30,40 50,25 40,10 60,30 100,30 ↩
    120,20 170,60 180,90 190,130 180,130 160,160 150),(40 160,50 140,66 125,60 100,80 140,90 ↩
     170,60 160,40 160))'::geometry),
  (2, 'POLYGON ((40 160,60 160,90 170,80 140,60 100,66 125,50 140,40 160))'::geometry),
  (3, 'POLYGON ((110 130,160 50,140 50,120 33,90 30,50 10,60 60,90 70,90 100,110 130))':: ↩
    geometry),
  (4, 'POLYGON ((160 150,150 120,160 90,160 50,110 130,160 150))'::geometry)
)
SELECT id, ST_CoverageSimplify(geom, 30) OVER ()
  FROM coverage;
```

Figure



See Also

ST_CoverageInvalidEdges, ST_CoverageUnion, ST_CoverageClean

7.15.4 ST_CoverageEdges

ST_CoverageEdges — Computes the unique edges of a polygonal coverage.

Synopsis

geometry **ST_CoverageEdges**(geometry geom, integer edgetype = 0);
geometry **ST_CoverageEdges**(geometry[] geoms, integer edgetype = 0);

Description

Returns a MultiLineString representing the unique edges of a polygonal coverage. A polygonal coverage is a set of non-overlapping polygons where adjacent polygons have matching vertices along shared edges.

The *edgetype* parameter can be used to select which edges are returned:

- **0 (ALL)** - all unique edges (default)
- **1 (EXTERIOR)** - only exterior edges (non-shared)
- **2 (INTERIOR)** - only interior edges (shared)

Availability: 3.7.0

Requires GEOS >= 3.15.0

Examples

This example returns all unique edges of a coverage.

Code

```
WITH coverage(geom) AS (
  VALUES (
    'GEOMETRYCOLLECTION(
      POLYGON((0 0,10 0,10 10,0 10,0 0)),
      POLYGON((10 0,20 0,20 10,10 10,10 0))
    )'::geometry
  )
)
SELECT geom AS input_coverage,
       ST_CoverageEdges(geom) AS all_edges
FROM coverage;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------------------
input_coverage | GEOMETRYCOLLECTION(POLYGON((0 0,10 0,10 10,0 10,0 0)),POLYGON((10 0,20 0,20 10,10 10,10 0)))
all_edges      | MULTILINESTRING((10 0,10 10),(10 10,0 10,0 0,10 0),(10 0,20 0,20 10,10 10))
```

This example returns only the interior shared edges of a coverage.

Code

```
WITH coverage(geom) AS (
  VALUES (
    'GEOMETRYCOLLECTION(
      POLYGON((0 0,10 0,10 10,0 10,0 0)),
      POLYGON((10 0,20 0,20 10,10 10,10 0))
    )'::geometry
  )
)
SELECT geom AS input_coverage,
       ST_CoverageEdges(geom, 2) AS shared_edges
FROM coverage;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------------------
input_coverage | GEOMETRYCOLLECTION(POLYGON((0 0,10 0,10 10,0 10,0 0)),POLYGON((10 0,20 0,20 10,10 10,10 0)))
shared_edges   | MULTILINESTRING((10 0,10 10))
```

See Also

ST_CoverageInvalidEdges, ST_CoverageSimplify, ST_CoverageUnion

7.15.5 ST_CoverageUnion

ST_CoverageUnion — Computes the union of a set of polygons forming a coverage by removing shared edges.

Synopsis

geometry **ST_CoverageUnion**(geometry set geom);

Description

An aggregate function which unions a set of polygons forming a polygonal coverage. The result is a polygonal geometry covering the same area as the coverage. This function produces the same result as ST_Union, but uses the coverage structure to compute the union much faster.



Note
If the input is not a valid coverage there may be unexpected artifacts in the output (such as unmerged or overlapping polygons). Use ST_CoverageInvalidEdges to determine if a coverage is valid.

Availability: 3.4.0 - requires GEOS >= 3.8.0

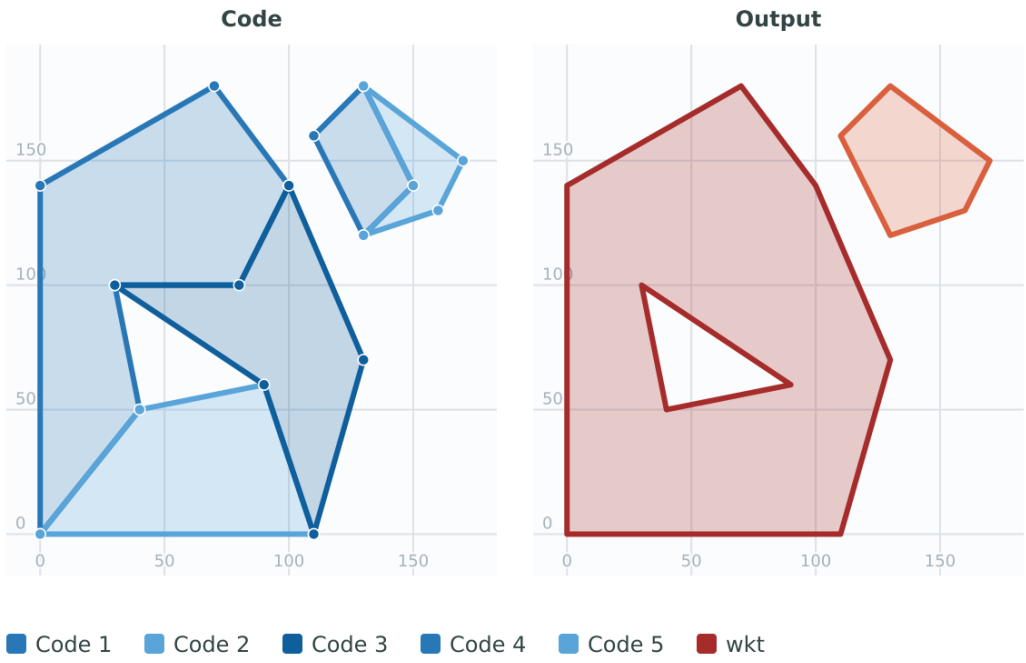
Examples

Union a polygonal coverage.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((10 10,10 150,80 190,110 150,90 110,40 110,50 60,10 10))'::geometry),
  (2, 'POLYGON ((120 10,10 10,50 60,100 70,120 10))'::geometry),
  (3, 'POLYGON ((140 80,120 10,100 70,40 110,90 110,110 150,140 80))'::geometry),
  (4, 'POLYGON ((140 190,120 170,140 130,160 150,140 190))'::geometry),
  (5, 'POLYGON ((180 160,170 140,140 130,160 150,140 190,180 160))'::geometry)
)
SELECT ST_CoverageUnion(geom) AS wkt
  FROM coverage;
```

Figure



See Also

ST_CoverageInvalidEdges, ST_CoverageSimplify, ST_CoverageClean, ST_Union

7.16 Affine Transformations

7.16.1 ST_Affine

ST_Affine — Apply a 3D affine transformation to a geometry.

Synopsis

geometry **ST_Affine**(geometry geomA, float a, float b, float c, float d, float e, float f, float g, float h, float i, float xoff, float yoff, float zoff);
geometry **ST_Affine**(geometry geomA, float a, float b, float d, float e, float xoff, float yoff);

Description

Applies a 3D affine transformation to the geometry to do things like translate, rotate, scale in one step.

Version 1: The call

Code

```
ST_Affine(geom, a, b, c, d, e, f, g, h, i, xoff, yoff, zoff)
```

represents the transformation matrix

Code

```
/ a  b  c  xoff \
| d  e  f  yoff |
| g  h  i  zoff |
\ 0  0  0     1 /
```

and the vertices are transformed as follows:

Code

```
x' = a*x + b*y + c*z + xoff
y' = d*x + e*y + f*z + yoff
z' = g*x + h*y + i*z + zoff
```

All of the translate / scale functions below are expressed via such an affine transformation.

Version 2: Applies a 2d affine transformation to the geometry. The call

Code

```
ST_Affine(geom, a, b, d, e, xoff, yoff)
```

represents the transformation matrix

Code

```
/  a  b  0  xoff  \       /  a  b  xoff  \
|  d  e  0  yoff  | rsp.  |  d  e  yoff  |
|  0  0  1     0  |       \  0  0     1  /
\  0  0  0     1  /
```

and the vertices are transformed as follows:

Code

```
x' = a*x + b*y + xoff
y' = d*x + e*y + yoff
z' = z
```

This method is a subcase of the 3D method above.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Availability: 1.1.2. Name changed from Affine to ST_Affine in 1.2.2



Note

Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

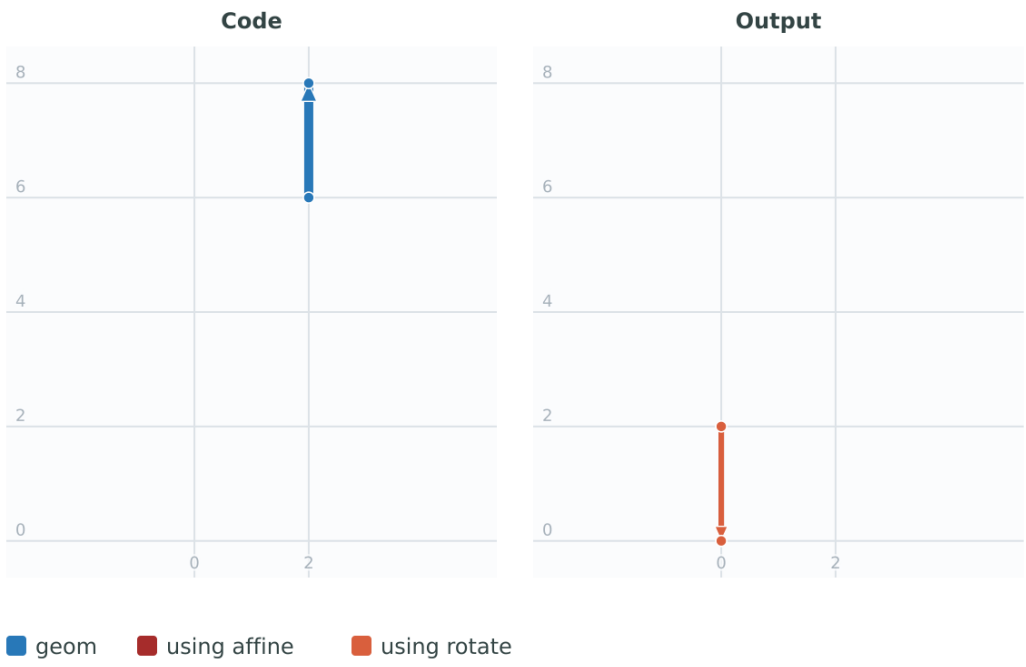
Examples

Rotate a 3D line 180 degrees about the z axis. Note this is long-hand for doing ST_Rotate();

Code

```
SELECT ST_Affine(geom, cos(pi()), -sin(pi()), 0, sin(pi()), cos(pi()), 0, 0, 0, 1, 0, 0, 0)  ↩
    As using_affine,
     ST_Rotate(geom, pi()) As using_rotate
   FROM (SELECT 'LINESTRING(1 2 3,1 4 3)'::geometry As geom) As foo;
```

Figure

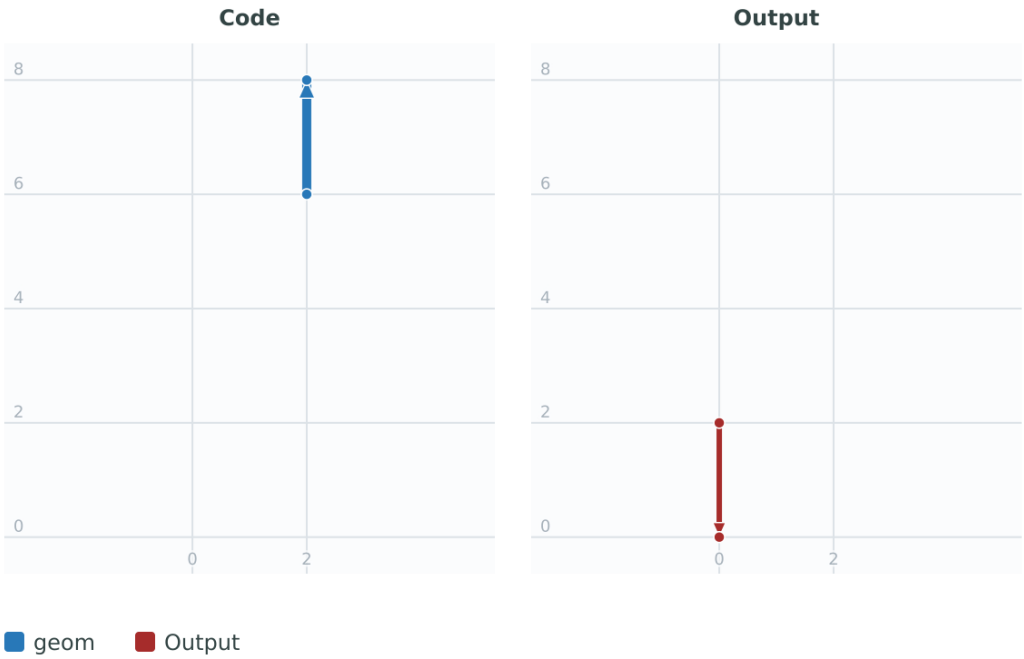


Rotate a 3D line 180 degrees in both the x and z axis.

Code

```
SELECT ST_Affine(geom, cos(pi()), -sin(pi()), 0, sin(pi()), cos(pi()), -sin(pi()), 0, sin(pi ↩
    ()), cos(pi()), 0, 0, 0)
    FROM (SELECT 'LINESTRING(1 2 3,1 4 3)'::geometry As geom) As foo;
```

Figure



See Also

ST_Rotate, ST_Scale, ST_Translate, ST_TransScale

7.16.2 ST_Rotate

ST_Rotate — Rotates a geometry about an origin point.

Synopsis

geometry **ST_Rotate**(geometry geomA, float rotRadians);
geometry **ST_Rotate**(geometry geomA, float rotRadians, float x0, float y0);
geometry **ST_Rotate**(geometry geomA, float rotRadians, geometry pointOrigin);

Description

Rotates geometry rotRadians counter-clockwise about the origin point. The rotation origin can be specified either as a POINT geometry, or as x and y coordinates. If the origin is not specified, the geometry is rotated about POINT(0 0).

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Enhanced: 2.0.0 additional parameters for specifying the origin of rotation were added.

Availability: 1.1.2. Name changed from Rotate to ST_Rotate in 1.2.2

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

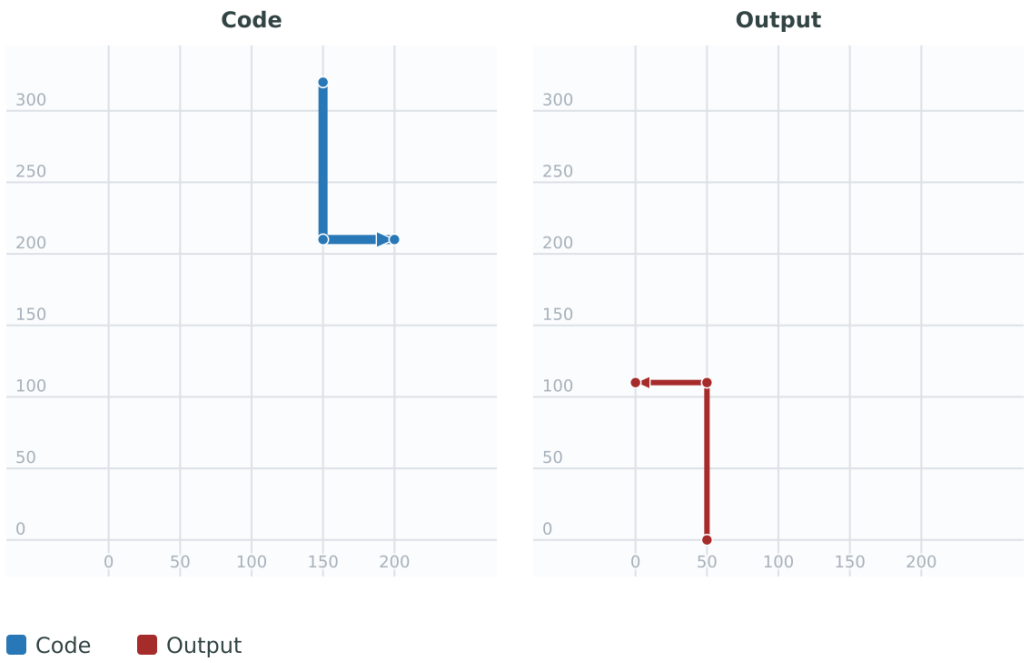
Examples

This example rotates a geometry by 180 degrees.

Code

```
SELECT ST_Rotate('LINESTRING (50 160,50 50,100 50)'::geometry, pi());
```

Figure

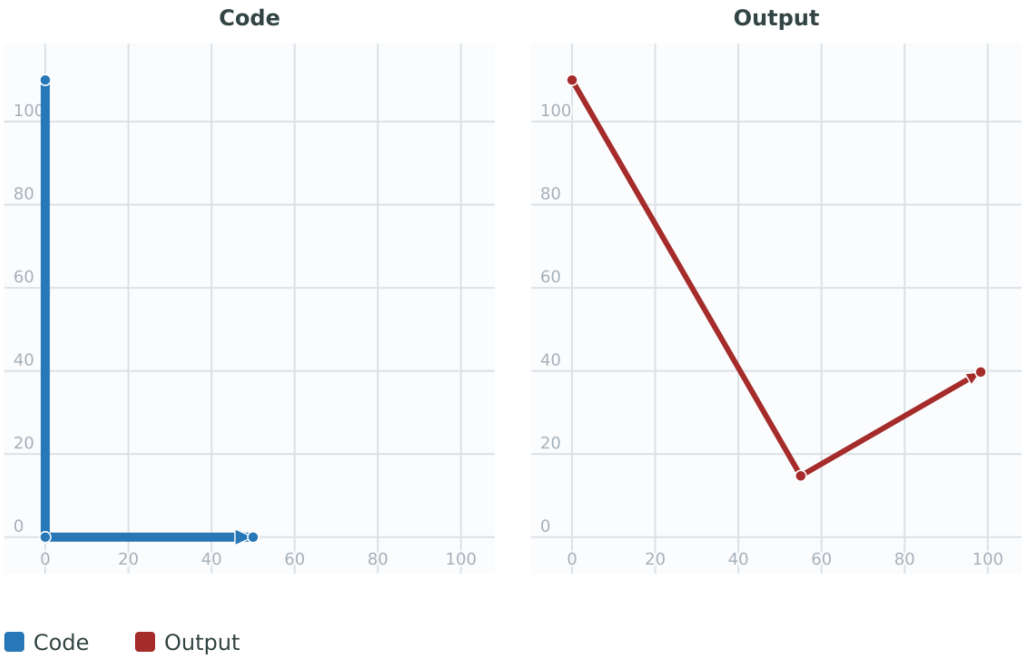


This example rotates a geometry 30 degrees counter-clockwise around x=50, y=160.

Code

```
SELECT ST_Rotate('LINESTRING (50 160,50 50,100 50)'::geometry, pi()/6, 50, 160);
```

Figure

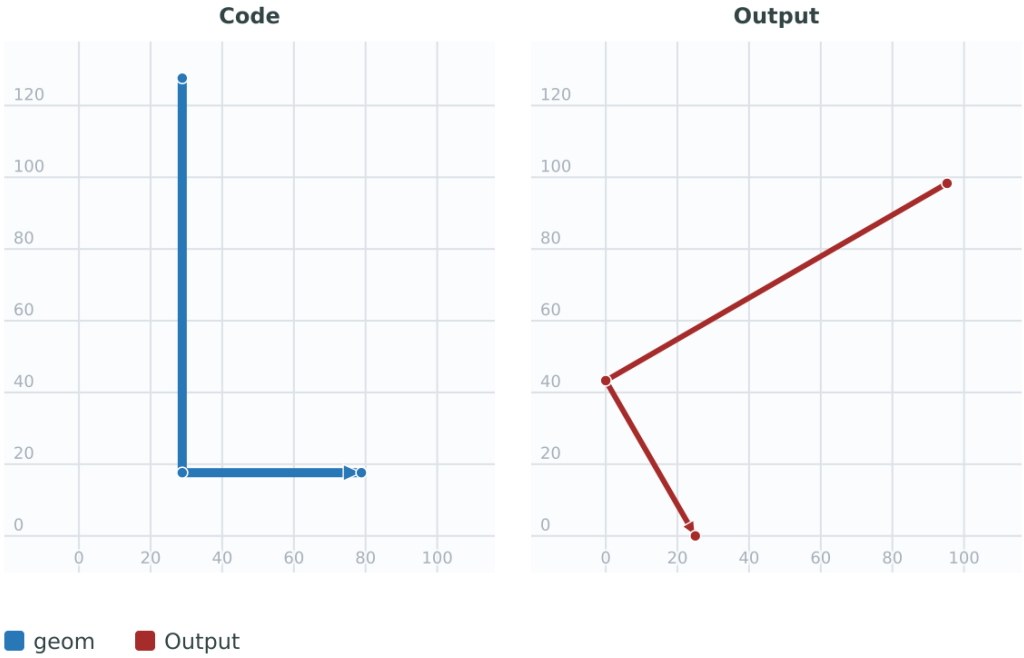


This example rotates a geometry 60 degrees clockwise around its centroid.

Code

```
SELECT ST_Rotate(geom, -pi()/3, ST_Centroid(geom))
FROM (SELECT 'LINESTRING (50 160,50 50,100 50)'::geometry AS geom) AS foo;
```

Figure



See Also

ST_Affine, ST_RotateX, ST_RotateY, ST_RotateZ

7.16.3 ST_RotateX

ST_RotateX — Rotates a geometry about the X axis.

Synopsis

geometry **ST_RotateX**(geometry geomA, float rotRadians);

Description

Rotates a geometry geomA - rotRadians about the X axis.



Note
`ST_RotateX(geomA, rotRadians)` is short-hand for `ST_Affine(geomA, 1, 0, 0, 0, cos(rotRadians), -sin(rotRadians), 0, sin(rotRadians), cos(rotRadians), 0, 0, 0).`

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Availability: 1.1.2. Name changed from RotateX to ST_RotateX in 1.2.2

 This function supports Polyhedral surfaces.

 This function supports 3d and will not drop the z-index.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

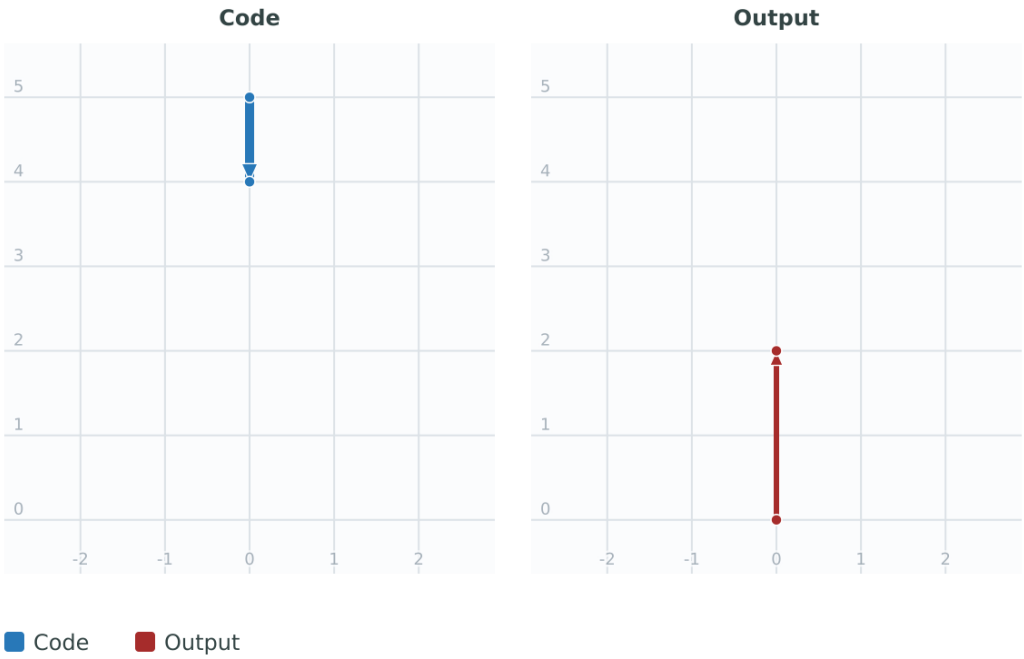
Examples

This example rotates a line 90 degrees around the X axis.

Code

```
SELECT ST_RotateX('LINESTRING(1 2 3,1 1 1)', pi()/2);
```

Figure



See Also

ST_Affine, ST_RotateY, ST_RotateZ

7.16.4 ST_RotateY

ST_RotateY — Rotates a geometry about the Y axis.

Synopsis

geometry **ST_RotateY**(geometry geomA, float rotRadians);

Description

Rotates a geometry geomA - rotRadians about the y axis.



Note

`ST_RotateY(geomA, rotRadians)` is short-hand for `ST_Affine(geomA, cos(rotRadians), 0, sin(rotRadians), 0, 1, 0, -sin(rotRadians), 0, cos(rotRadians), 0, 0, 0)`.

Availability: 1.1.2. Name changed from RotateY to ST_RotateY in 1.2.2

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.


This function supports Polyhedral surfaces.


This function supports 3d and will not drop the z-index.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

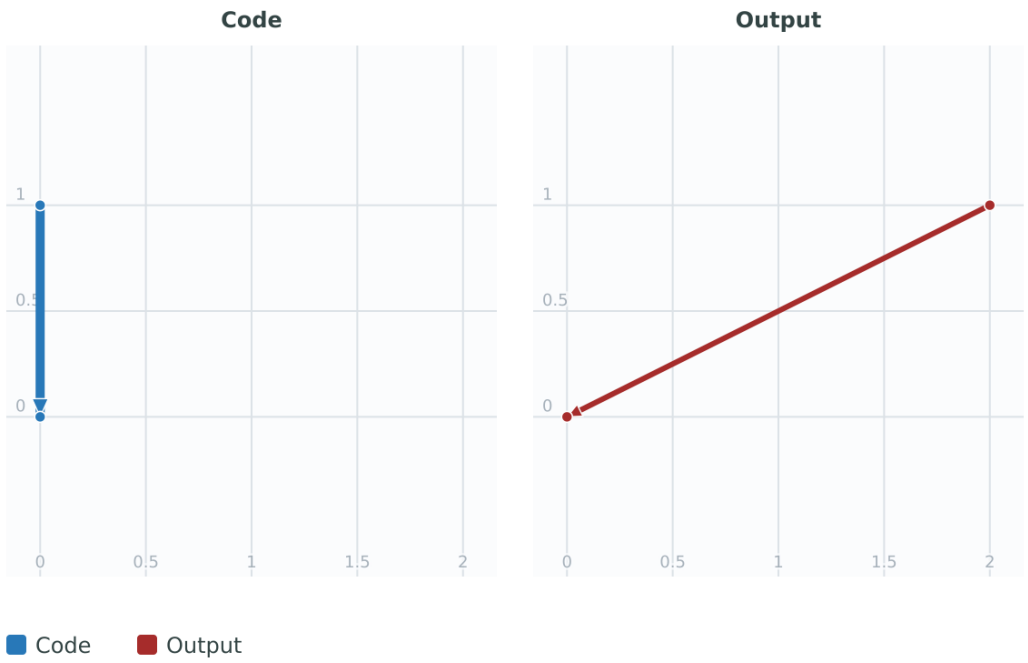
Examples

This example rotates a line 90 degrees around the Y axis.

Code

```
SELECT ST_RotateY('LINESTRING(1 2 3,1 1 1)', pi()/2);
```

Figure



See Also

ST_Affine, ST_RotateX, ST_RotateZ

7.16.5 ST_RotateZ

ST_RotateZ — Rotates a geometry about the Z axis.

Synopsis

geometry **ST_RotateZ**(geometry geomA, float rotRadians);

Description

Rotates a geometry geomA - rotRadians about the Z axis.



Note
This is a synonym for ST_Rotate

Note
`ST_RotateZ(geomA, rotRadians)` is short-hand for `SELECT ST_Affine(geomA, cos(rotRadians), -sin(rotRadians), 0, sin(rotRadians), cos(rotRadians), 0, 0, 0, 1, 0, 0, 0).`

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Availability: 1.1.2. Name changed from RotateZ to ST_RotateZ in 1.2.2

Note
Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

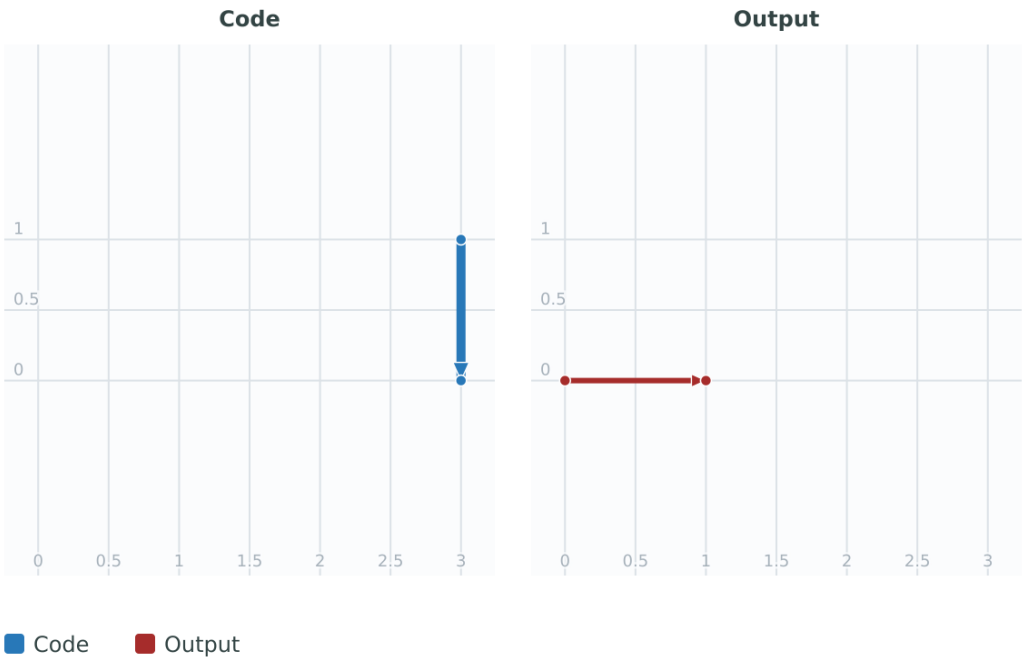
Examples

This example rotates a line 90 degrees around the Z axis.

Code

```
SELECT ST_RotateZ('LINESTRING(1 2 3,1 1 1)', pi()/2);
```

Figure

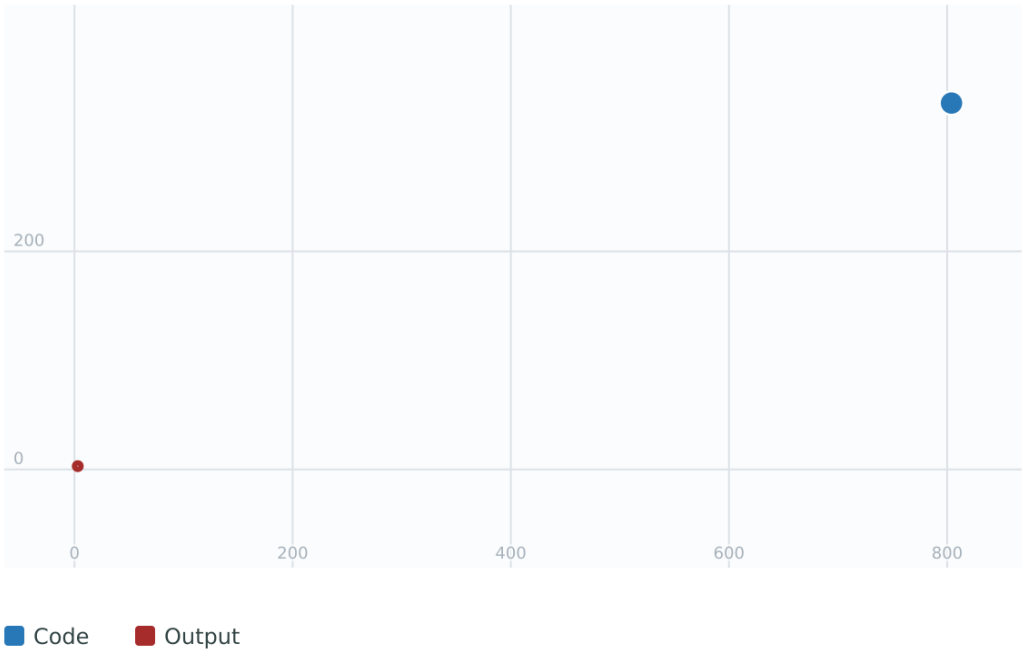


This example rotates a curved circle around the Z axis.

Code

```
SELECT ST_RotateZ(geom, pi()/2)
FROM (SELECT ST_LineToCurve(ST_Buffer('POINT(234 567)'::geometry, 3)) As geom) As foo;
```

Figure



See Also

ST_Affine, ST_RotateX, ST_RotateY

7.16.6 ST_Scale

ST_Scale — Scales a geometry by given factors.

Synopsis

geometry **ST_Scale**(geometry geomA, float XFactor, float YFactor, float ZFactor);
geometry **ST_Scale**(geometry geomA, float XFactor, float YFactor);
geometry **ST_Scale**(geometry geom, geometry factor);
geometry **ST_Scale**(geometry geom, geometry factor, geometry origin);

Description

Scales the geometry to a new size by multiplying the ordinates with the corresponding factor parameters.

The version taking a geometry as the `factor` parameter allows passing a 2d, 3dm, 3dz or 4d point to set scaling factor for all supported dimensions. Missing dimensions in the `factor` point are equivalent to no scaling the corresponding dimension.

The three-geometry variant allows a "false origin" for the scaling to be passed in. This allows "scaling in place", for example using the centroid of the geometry as the false origin. Without a false origin, scaling takes place relative to the actual origin, so all coordinates are just multiplied by the scale factor.



Note

Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

Availability: 1.1.0.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Enhanced: 2.2.0 support for scaling all dimension (`factor` parameter) was introduced.

Enhanced: 2.5.0 support for scaling relative to a local origin (`origin` parameter) was introduced.

 This function supports Polyhedral surfaces.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

 This function supports M coordinates.

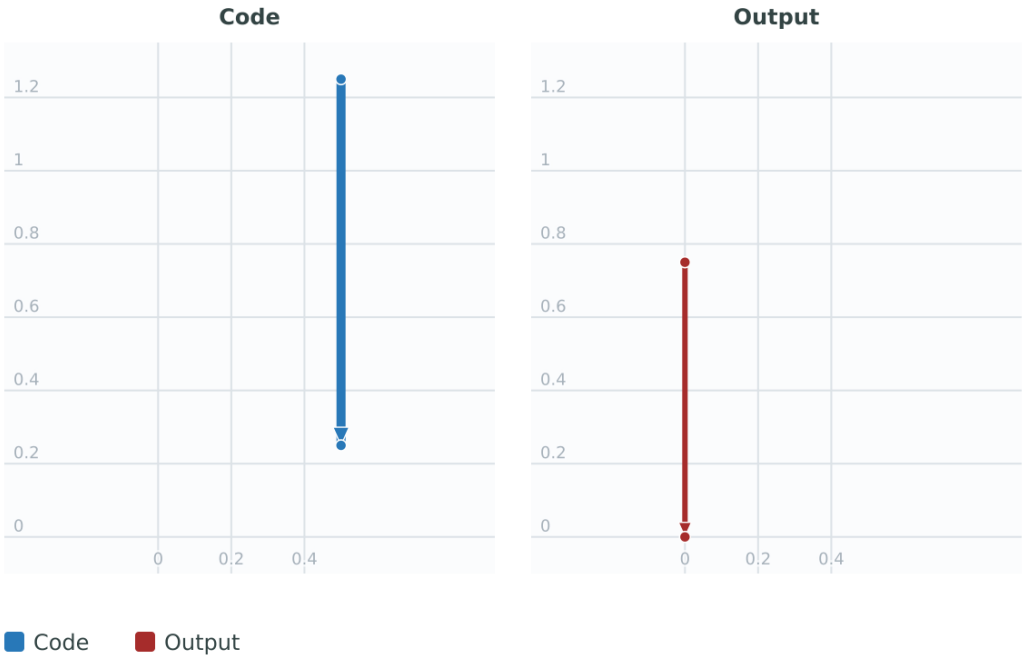
Examples

This example scales X, Y, and Z values.

Code

```
SELECT ST_Scale('LINESTRING(1 2 3,1 1 1)', 0.5, 0.75, 0.8);
```

Figure

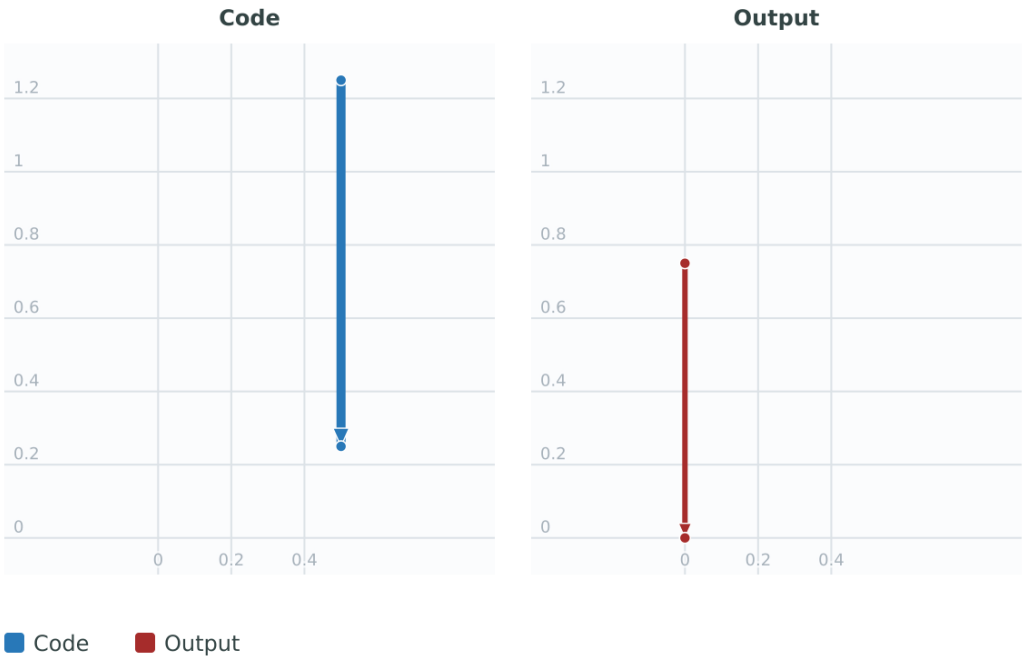


This example scales X and Y values.

Code

```
SELECT ST_Scale('LINESTRING(1 2 3,1 1 1)', 0.5, 0.75);
```

Figure

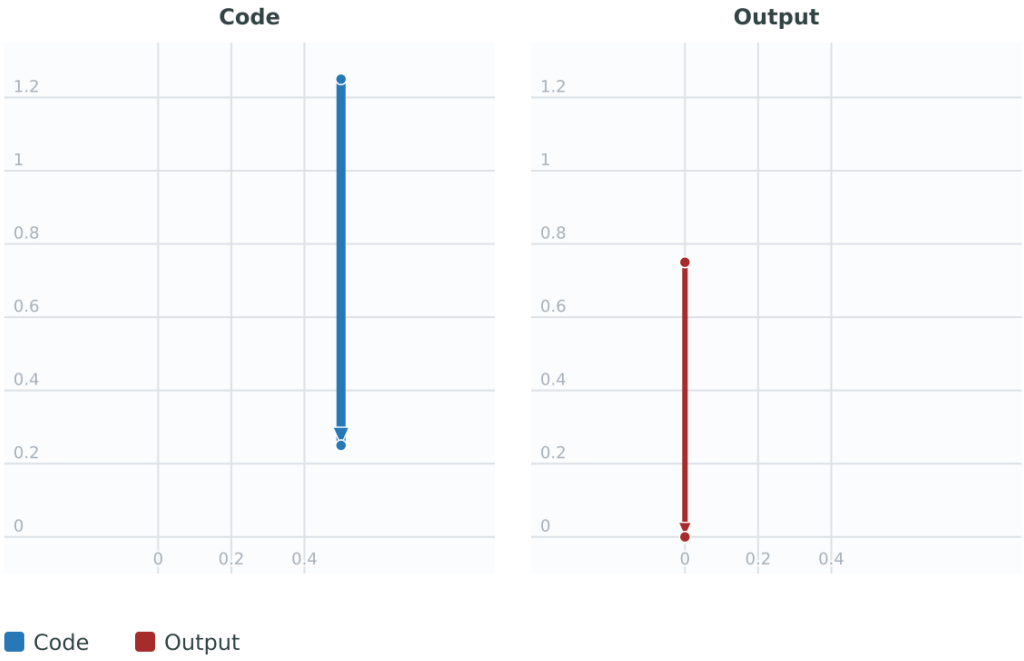


This example scales X, Y, Z, and M values.

Code

```
SELECT ST_Scale('LINESTRING(1 2 3 4,1 1 1 1)',
    ST_MakePoint(0.5, 0.75, 2, -1));
```

Figure

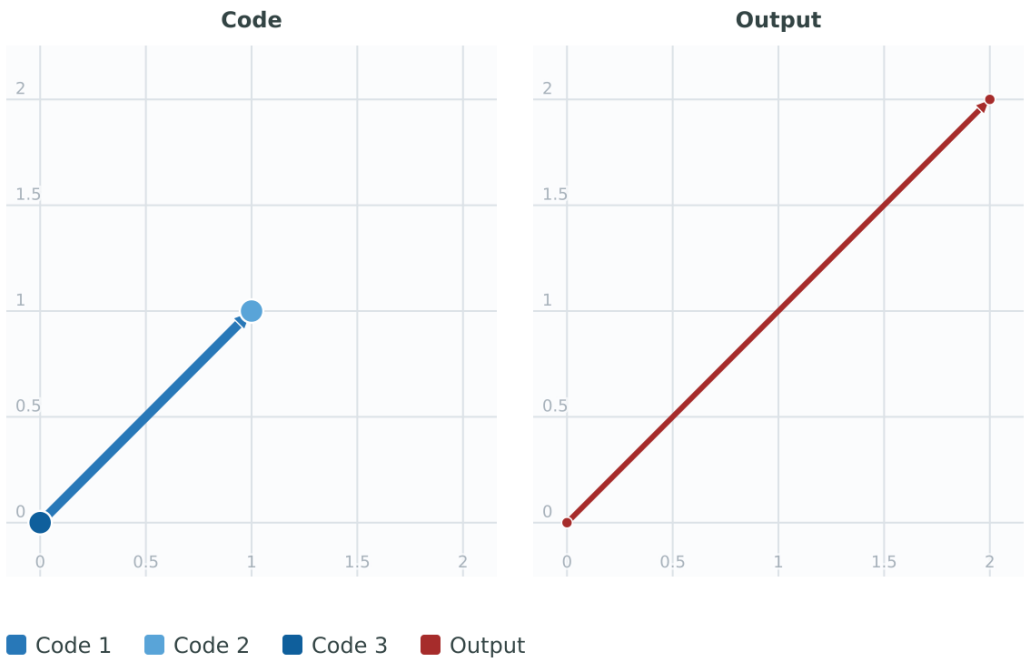


This example scales X and Y values using a false origin.

Code

```
SELECT ST_Scale('LINESTRING(1 1,2 2)', 'POINT(2 2)', 'POINT(1 1)'::geometry);
```

Figure



This example constructs an ellipse centered at (10, 20), with X radius 8, Y radius 4, and rotated 30 degrees counter-clockwise. Scaling and rotation use the center as their origin, so the ellipse keeps both its position and SRID.

Code

```
WITH params AS (
  SELECT
    ST_SetSRID(ST_Point(10, 20), 3857) AS center,
    8.0 AS x_radius,
    4.0 AS y_radius,
    radians(30) AS angle
)
SELECT ST_Rotate(
         ST_Scale(
           ST_Buffer(center, 1, 'quad_segs=32'),
           ST_MakePoint(x_radius, y_radius),
           center
         ),
         angle,
         center
       ) AS ellipse
FROM params;
```

See Also

ST_Affine, ST_Buffer, ST_MakePoint, ST_Rotate, ST_TransScale

7.16.7 ST_Translate

ST_Translate — Translates a geometry by given offsets.

Synopsis

geometry **ST_Translate**(geometry g1, float deltax, float deltay);
geometry **ST_Translate**(geometry g1, float deltax, float deltay, float deltaz);

Description

Returns a new geometry whose coordinates are translated delta x,delta y,delta z units. Units are based on the units defined in spatial reference (SRID) for this geometry. M coordinates are preserved.



Note
Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

Availability: 1.2.2

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

 This function supports M coordinates.

Examples

Move a point 1 degree longitude

Code

```
SELECT ST_Translate(ST_GeomFromText('POINT(-71.01 42.37)', 4326), 1, 0) AS wgs_transgeomtxt;
```

Output

```
POINT(-70.01 42.37)
```

Move a linestring 1 degree longitude and 1/2 degree latitude

Code

```
SELECT ST_Translate(ST_GeomFromText('LINESTRING(-71.01 42.37,-71.11 42.38)', 4326), 1, 0.5)  ↩
    AS wgs_transgeomtxt;
```

Figure



Move a 3d point

Code

```
SELECT ST_Translate(CAST('POINT(0 0 0)' AS geometry), 5, 12, 3);
```

Output

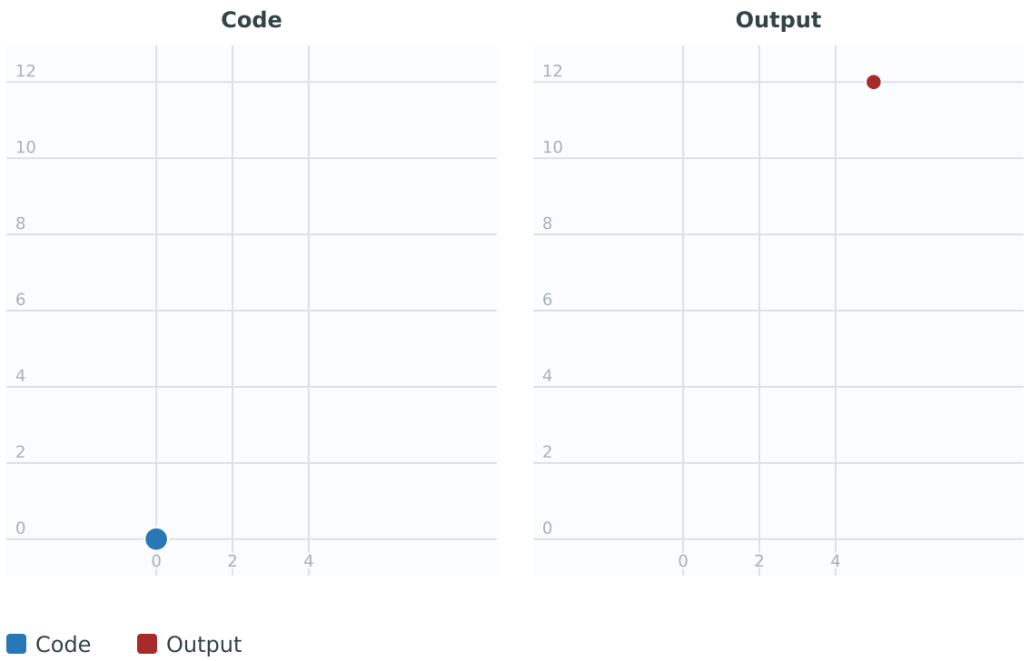
```
POINT(5 12 3)
```

Move points with a measure coordinate

Code

```
SELECT ST_Translate('POINTM(0 0 11)', 5, 12);
```

Figure



Code

```
SELECT ST_Translate('POINT(0 0 7 11)', 5, 12, 3);
```

Output

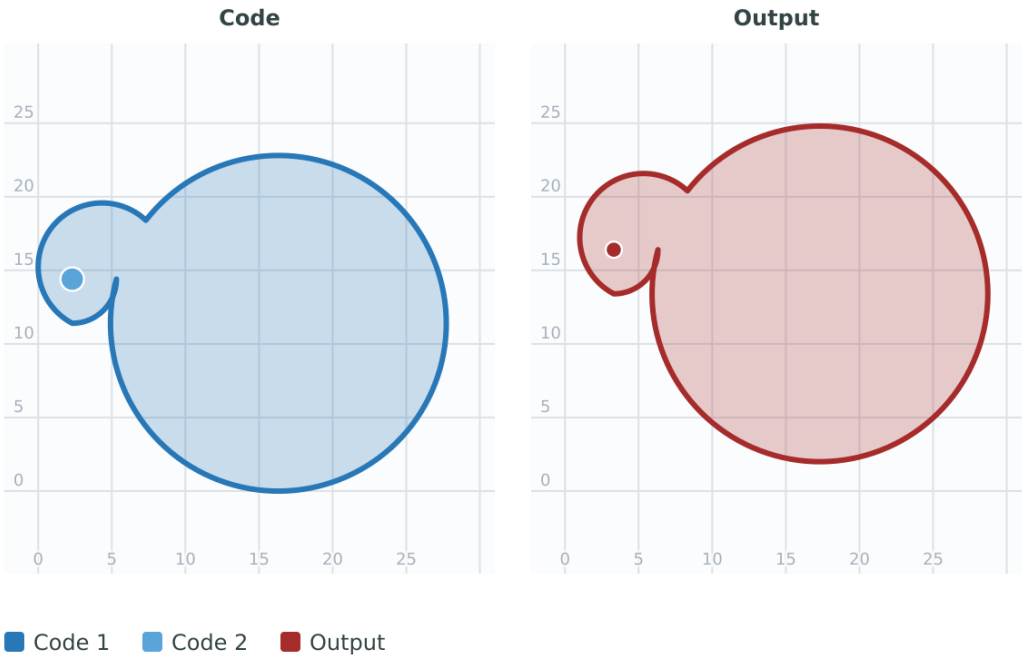
```
POINT(5 12 10 11)
```

Move a curve and a point

Code

```
SELECT ST_Translate(ST_Collect('CURVEPOLYGON(CIRCULARSTRING(4 3,3.12 0.878,1 0,-1.121 ↩
    5.1213,6 7,8 9,4 3))', 'POINT(1 3)'), 1, 2);
```

Figure



See Also

ST_Affine, ST_AsText, ST_GeomFromText

7.16.8 ST_TransScale

ST_TransScale — Translates and scales a geometry by given offsets and factors.

Synopsis

geometry **ST_TransScale**(geometry geomA, float deltaX, float deltaY, float XFactor, float YFactor);

Description

Translates the geometry using the deltaX and deltaY args, then scales it using the XFactor, YFactor args, working in 2D only.



Note
`ST_TransScale(geomA, deltaX, deltaY, XFactor, YFactor)` is short-hand for `ST_Affine(geomA, XFactor, 0, 0, 0, YFactor, 0, 0, 0, 1, deltaX*XFactor, deltaY*YFactor, 0)`.



Note
Prior to 1.3.4, this function crashes if used with geometries that contain CURVES. This is fixed in 1.3.4+

Availability: 1.1.0.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_TransScale('LINESTRING(1 2 3,1 1 1)', 0.5, 1, 1, 2);
```

Figure

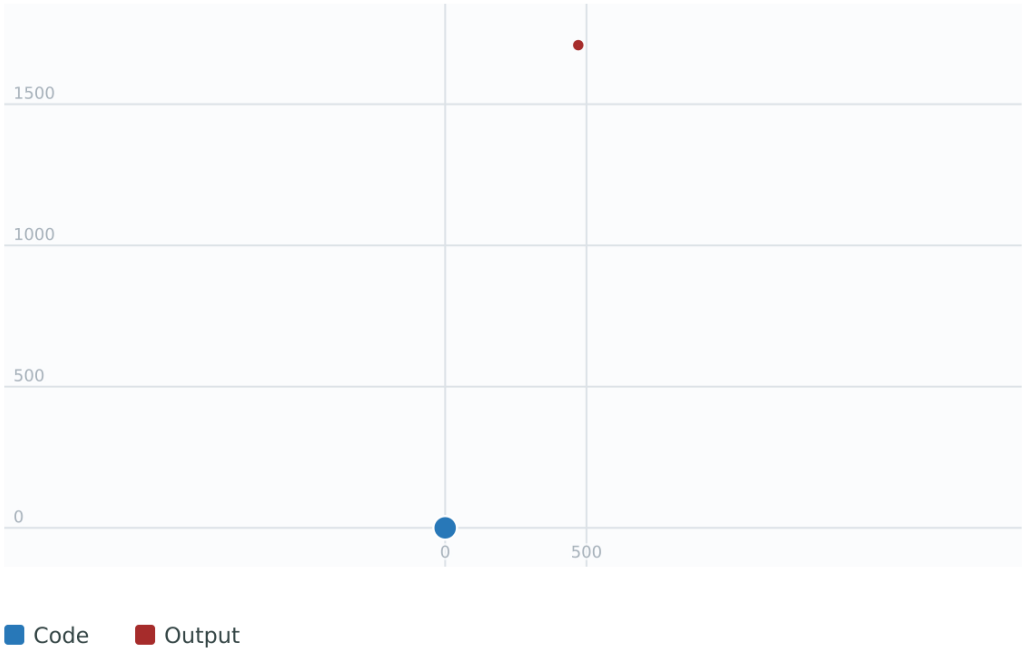


This example buffers a point to approximate a circle, converts it to a curve, translates it by 1,2, and scales it by 3,4.

Code

```
SELECT ST_TransScale(ST_LineToCurve(ST_Buffer('POINT(234 567)', 3)), 1, 2, 3, 4);
```

Figure



See Also

ST_Affine, ST_Translate

7.17 Clustering Functions

7.17.1 ST_ClusterDBSCAN

ST_ClusterDBSCAN — Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.

Synopsis

integer **ST_ClusterDBSCAN**(geometry winset geom, float8 eps, integer minpoints);

Description

A window function that returns a cluster number for each input geometry, using the 2D Density-based spatial clustering of applications with noise (DBSCAN) algorithm. Unlike ST_ClusterKMeans, it does not require the number of clusters to be specified, but instead uses the desired distance (`eps`) and density (`minpoints`) parameters to determine each cluster.

An input geometry is added to a cluster if it is either:

- A "core" geometry, that is within `eps` distance of at least `minpoints` input geometries (including itself); or
- A "border" geometry, that is within `eps` distance of a core geometry.

Note that border geometries may be within `eps` distance of core geometries in more than one cluster. Either assignment would be correct, so the border geometry will be arbitrarily assigned to one of the available clusters. In this situation it is possible for a correct cluster to be generated with fewer than `minpoints` geometries. To ensure deterministic assignment of border geometries (so that repeated calls to ST_ClusterDBSCAN will produce identical results) use an `ORDER BY` clause in the window definition. Ambiguous cluster assignments may differ from other DBSCAN implementations.



Note
Geometries that do not meet the criteria to join any cluster are assigned a cluster number of NULL.

Availability: 2.3.0

 This method supports Circular Strings and Curves.

Examples

Cluster polygons within 50 units of each other, requiring at least two polygons per cluster. The isolated polygon is reported as noise.

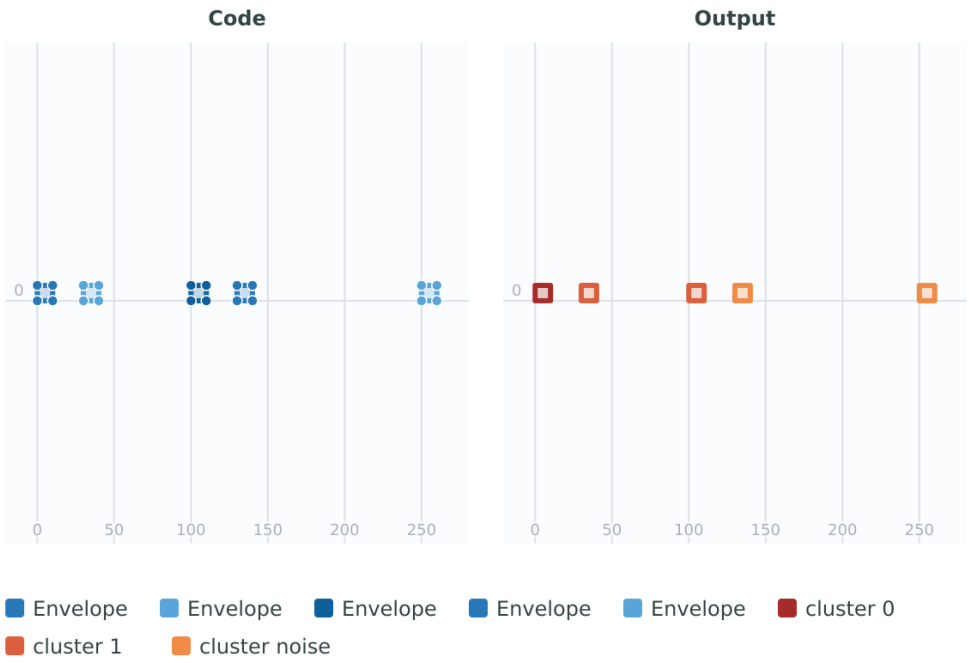
Code

```
WITH input(name, geom) AS (
  VALUES
    ('A1', ST_MakeEnvelope(0, 0, 10, 10)),
    ('A2', ST_MakeEnvelope(30, 0, 40, 10)),
    ('B1', ST_MakeEnvelope(100, 0, 110, 10)),
    ('B2', ST_MakeEnvelope(130, 0, 140, 10)),
    ('noise', ST_MakeEnvelope(250, 0, 260, 10))
), clustered AS (
  SELECT name,
         geom,
         ST_ClusterDBSCAN(geom, eps => 50, minpoints => 2)
           OVER (ORDER BY name) AS cid
  FROM input
)
SELECT COALESCE(cid::text, 'noise') AS cluster,
       string_agg(name, ',' ORDER BY name) AS members,
       ST_Collect(geom ORDER BY name) AS cluster_geom
FROM clustered
GROUP BY cid
ORDER BY cid NULLS LAST;
```

Output

```
0 | A1,A2 | MULTIPOLYGON(((0 0,0 10,10 10,10 0,0 0)),((30 0,30 10,40 10,40 0,30 0)))
1 | B1,B2 | MULTIPOLYGON(((100 0,100 10,110 10,110 0,100 0)),((130 0,130 10,140 10,140 0,130 0)))
noise | noise | MULTIPOLYGON(((250 0,250 10,260 10,260 0,250 0)))
```

Figure



A example showing combining parcels with the same cluster number into geometry collections.

Code

```
SELECT cid, ST_Collect(geom) AS cluster_geom, array_agg(parcel_id) AS ids_in_cluster FROM (
    SELECT parcel_id, ST_ClusterDBSCAN(geom, eps => 0.5, minpoints => 5) over () AS cid,  ↩
    geom
    FROM parcels) sq
GROUP BY cid;
```

See Also

ST_DWithin, ST_ClusterKMeans, ST_ClusterIntersecting, ST_ClusterIntersectingWin, ST_ClusterWithin, ST_ClusterWithinWin

7.17.2 ST_ClusterIntersecting

ST_ClusterIntersecting — Aggregate function that clusters input geometries into connected sets.

Synopsis

geometry[] **ST_ClusterIntersecting**(geometry set g);

Description

An aggregate function that returns an array of GeometryCollections partitioning the input geometries into connected clusters that are disjoint. Each geometry in a cluster intersects at least one other geometry in the cluster, and does not intersect any geometry in other clusters.

Availability: 2.2.0

Examples

Code

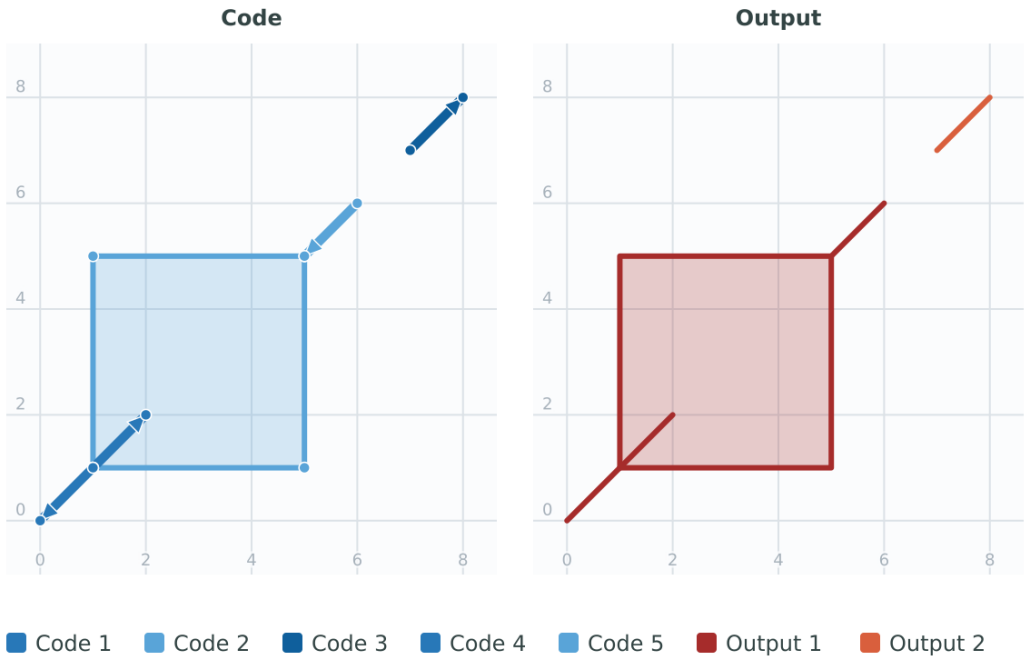
```
WITH testdata AS
  (SELECT unnest(
            ARRAY['LINESTRING (0 0,1 1)'::geometry,
            'LINESTRING (5 5,4 4)'::geometry,
            'LINESTRING (6 6,7 7)'::geometry,
            'LINESTRING (0 0,-1 -1)'::geometry,
            'POLYGON ((0 0,4 0,4 4,0 4,0 0))'::geometry]) AS geom)

SELECT unnest(ST_ClusterIntersecting(geom)) FROM testdata;
```

Output

```
GEOMETRYCOLLECTION(LINESTRING(0 0,1 1),LINESTRING(5 5,4 4),LINESTRING(0 0,-1 -1),POLYGON((0 0,4 0,4 4,0 4,0 0)))
GEOMETRYCOLLECTION(LINESTRING(6 6,7 7))
```

Figure



See Also

ST_ClusterIntersectingWin, ST_ClusterWithin, ST_ClusterWithinWin

7.17.3 ST_ClusterIntersectingWin

ST_ClusterIntersectingWin — Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.

Synopsis

integer **ST_ClusterIntersectingWin**(geometry winset geom);

Description

A window function that builds connected clusters of geometries that intersect. It is possible to traverse all geometries in a cluster without leaving the cluster. The return value is the cluster number that the geometry argument participates in, or null for null inputs.

Availability: 3.4.0

Examples

Return an explicit cluster id for each input geometry. Using the conventional `cid` alias keeps the geometry column ready for color-by-cluster rendering.

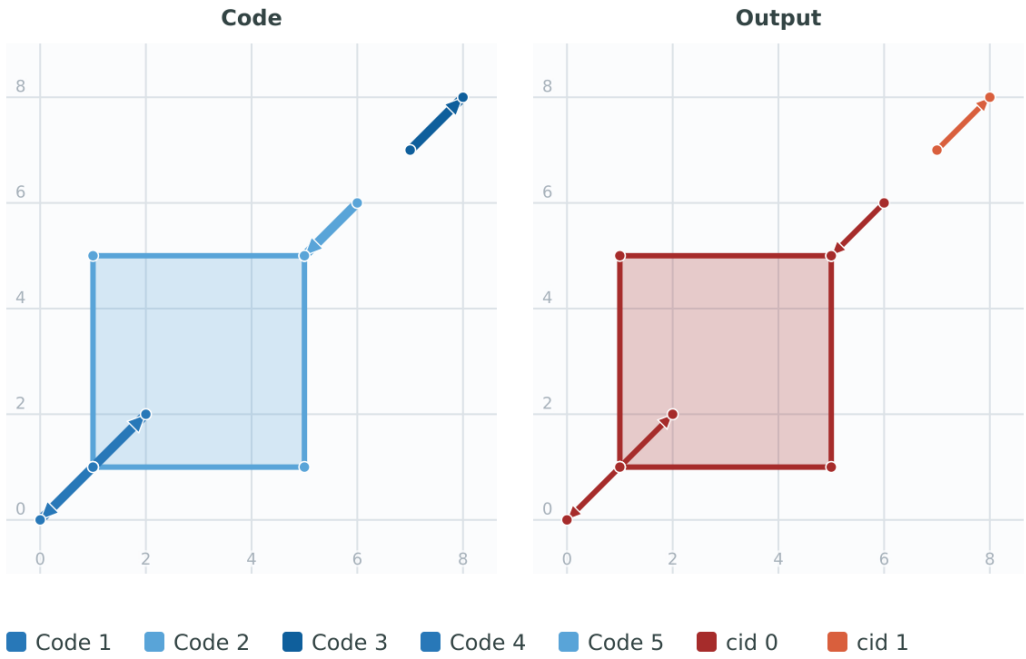
Code

```
WITH testdata(id, geom) AS (
  VALUES (1, 'LINESTRING (0 0,1 1)'::geometry),
         (2, 'LINESTRING (5 5,4 4)'::geometry),
         (3, 'LINESTRING (6 6,7 7)'::geometry),
         (4, 'LINESTRING (0 0,-1 -1)'::geometry),
         (5, 'POLYGON ((0 0,4 0,4 4,0 4,0 0))'::geometry)
), clustered AS (
  SELECT id,
         geom,
         ST_ClusterIntersectingWin(geom) OVER () AS cid
  FROM testdata
)
SELECT id,
       geom AS geom_wkt,
       cid
FROM clustered
ORDER BY id;
```

Output

```
 id |            geom_wkt             | cid
----+---------------------------------+-----
  1 | LINESTRING(0 0,1 1)             |   0
  2 | LINESTRING(5 5,4 4)             |   0
  3 | LINESTRING(6 6,7 7)             |   1
  4 | LINESTRING(0 0,-1 -1)           |   0
  5 | POLYGON((0 0,4 0,4 4,0 4,0 0)) |   0
```

Figure



See Also

ST_ClusterIntersecting, ST_ClusterWithin, ST_ClusterWithinWin

7.17.4 ST_ClusterRelateWin

ST_ClusterRelateWin — Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.

Synopsis

integer **ST_ClusterRelateWin**(geometry winset geom, text relate_matrix);

Description

A window function that builds connected clusters of geometries that intersect. Geometries are added to a cluster if they share a pairwise DE9IM relationship with another member of the cluster. With this function it is possible to build a cluster of all objects that touch at boundaries, but exclude those that merely overlap.

Availability: 3.7.0

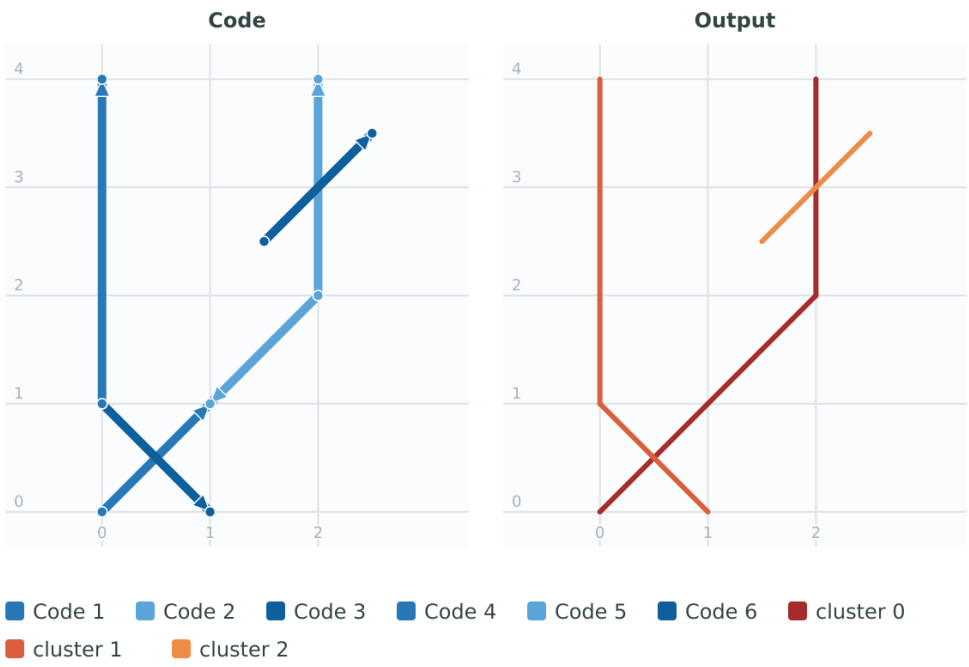
Examples

This collection of line strings would form a single cluster using ST_ClusterIntersectingWin, but using ST_ClusterRelateWin can be clustered into three groups that connect only via their end points.

Code

```
WITH data(id, geom) AS (
  VALUES (1, 'LINESTRING(0 0,1 1)'::geometry),
         (2, 'LINESTRING(2 2,1 1)'::geometry),
         (3, 'LINESTRING(0 1,1 0)'::geometry),
         (4, 'LINESTRING(0 1,0 4)'::geometry),
         (5, 'LINESTRING(2 2,2 4)'::geometry),
         (6, 'LINESTRING(1.5 2.5,2.5 3.5)'::geometry)
), clustered AS (
  SELECT id, geom,
         ST_ClusterRelateWin(geom, '****0****') OVER () AS cluster
  FROM data
)
SELECT cluster,
       string_agg(id::text, ',' ORDER BY id) AS members,
       ST_Collect(geom ORDER BY id) AS cluster_geom
FROM clustered
GROUP BY cluster
ORDER BY cluster;
```

Figure



See Also

ST_Relate, ST_ClusterIntersectingWin, ST_ClusterWithinWin

7.17.5 ST_MinimumSpanningTree

ST_MinimumSpanningTree — Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.

Synopsis

integer **ST_MinimumSpanningTree**(geometry winset geom);

Description

A window function that builds connected graphs of line strings based on the Minimum Spanning Tree (MST) of the input geometries. The return value is the cluster number that the geometry argument participates in, or zero if it is not part of the minimum tree.

The Minimum Spanning Tree connects all geometries in the window partition such that the total length of the connecting lines is minimized. If the graph is not fully connected (e.g. infinite distance between some geometries), it produces a Minimum Spanning Forest, and each connected component is assigned a unique tree ID.

Availability: 3.7.0

Requires GEOS >= 3.15.0

Examples

These six edges describe a square with both diagonals. The minimum spanning tree keeps three edges and excludes the rest.

Code

```
WITH edges(id, geom) AS (
  VALUES
    (1, 'LINESTRING(0 0,1 0)'::geometry),
    (2, 'LINESTRING(0 0,0 1)'::geometry),
    (3, 'LINESTRING(1 1,0 1)'::geometry),
    (4, 'LINESTRING(1 1,1 0)'::geometry),
    (5, 'LINESTRING(0 0,1 1)'::geometry),
    (6, 'LINESTRING(1 0,0 1)'::geometry)
), marked AS (
  SELECT id,
         geom,
         ST_MinimumSpanningTree(geom) OVER (ORDER BY id) AS tree_id
  FROM edges
)
SELECT string_agg(id::text, ',' ORDER BY id)
         FILTER (WHERE tree_id > 0) AS tree_edge_ids,
       ST_Collect(geom ORDER BY id)
         FILTER (WHERE tree_id > 0) AS tree_edges,
       string_agg(id::text, ',' ORDER BY id)
         FILTER (WHERE tree_id = 0) AS excluded_edge_ids,
       ST_Collect(geom ORDER BY id)
         FILTER (WHERE tree_id = 0) AS excluded_edges
FROM marked;
```

Output

```
-[ RECORD 1 ]-----
tree_edge_ids     | 1,2,3
tree_edges        | MULTILINESTRING((0 0,1 0),(0 0,0 1),(1 1,0 1))
excluded_edge_ids | 4,5,6
excluded_edges    | MULTILINESTRING((1 1,1 0),(0 0,1 1),(1 0,0 1))
```

See Also

ST_ClusterDBSCAN, ST_ClusterKMeans, ST_ClusterIntersectingWin

7.17.6 ST_ClusterKMeans

ST_ClusterKMeans — Window function that returns a cluster id for each input geometry using the K-means algorithm.

Synopsis

integer **ST_ClusterKMeans**(geometry winset geom , integer k , float8 max_radius);

Description

Returns K-means cluster number for each input geometry. The distance used for clustering is the distance between the centroids for 2D geometries, and distance between bounding box centers for 3D geometries. For POINT inputs, M coordinate will be treated as weight of input and has to be larger than 0.

`max_radius`, if set, will cause ST_ClusterKMeans to generate more clusters than `k` ensuring that no cluster in output has radius larger than `max_radius`. This is useful in reachability analysis.

Enhanced: 3.2.0 Support for `max_radius`

Enhanced: 3.1.0 Support for 3D geometries and weights

Availability: 2.3.0

Examples

Define the parcel geometries once in a CTE and reuse them across the executable examples.

Parcels color-coded by cluster number (`cid`).

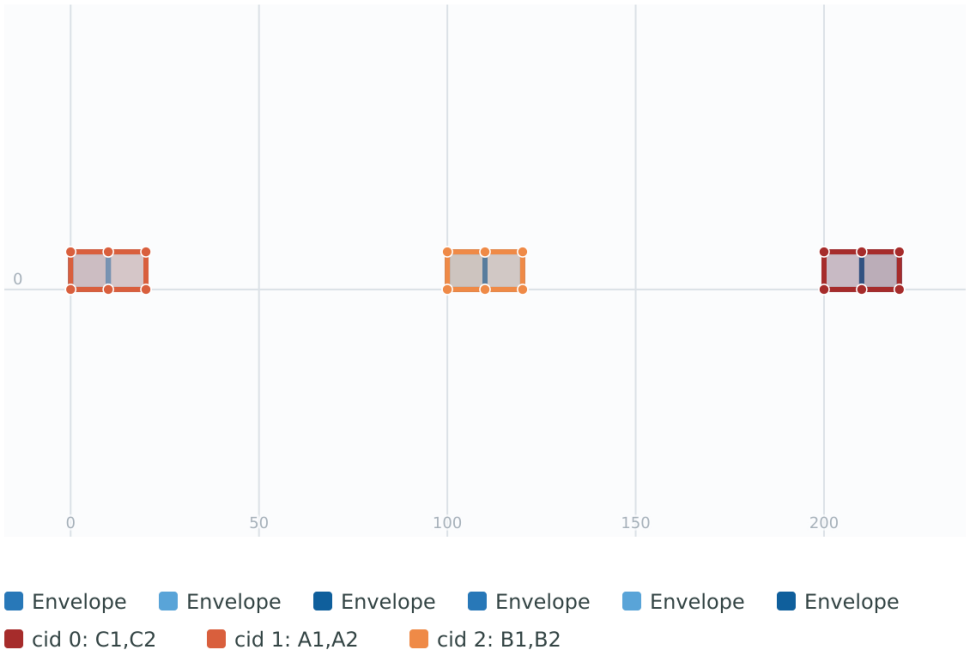
Code

```
WITH parcels(parcel_id, geom) AS (
  VALUES
    ('A1', ST_MakeEnvelope(0, 0, 10, 10)),
    ('A2', ST_MakeEnvelope(10, 0, 20, 10)),
    ('B1', ST_MakeEnvelope(100, 0, 110, 10)),
    ('B2', ST_MakeEnvelope(110, 0, 120, 10)),
    ('C1', ST_MakeEnvelope(200, 0, 210, 10)),
    ('C2', ST_MakeEnvelope(210, 0, 220, 10))
), clustered AS (
  SELECT parcel_id,
         geom,
         ST_ClusterKMeans(geom, 3) OVER (ORDER BY parcel_id) AS cid
  FROM parcels
)
SELECT cid,
       string_agg(parcel_id, ',' ORDER BY parcel_id) AS parcels,
       ST_Union(geom) AS cluster_geom
FROM clustered
GROUP BY cid
ORDER BY cid;
```

Output

```
0 | C1,C2 | POLYGON((200 10,210 10,220 10,220 0,210 0,200 0,200 10))
1 | A1,A2 | POLYGON((0 10,10 10,20 10,20 0,10 0,0 0,0 10))
2 | B1,B2 | POLYGON((100 10,110 10,120 10,120 0,110 0,100 0,100 10))
```

Figure



Partitioning parcel clusters by type:

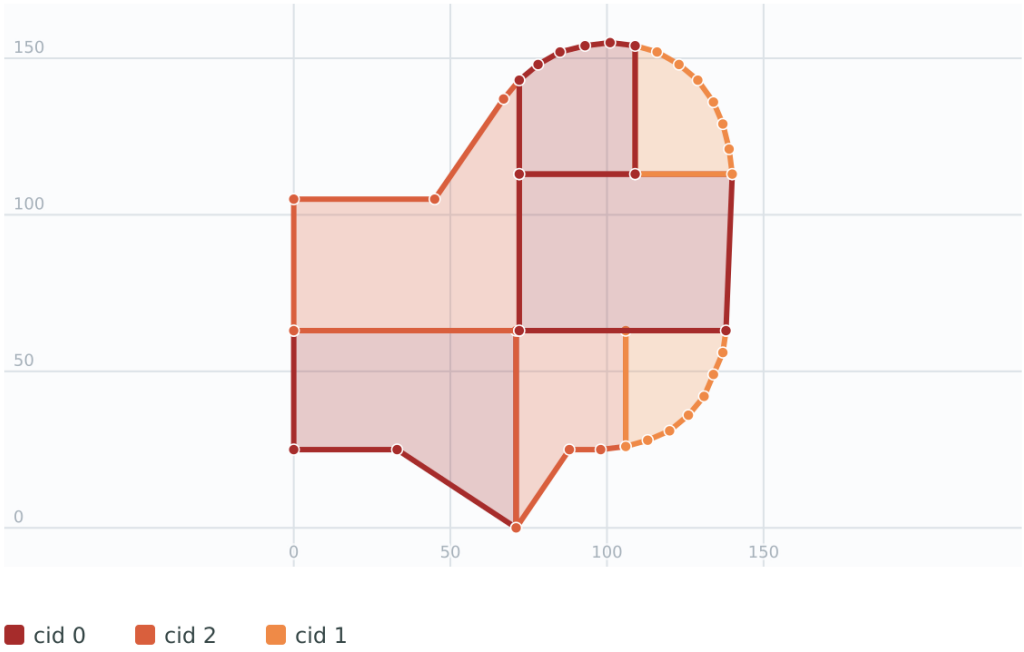
Code

```
WITH parcel_geoms AS (
  SELECT geom
  FROM ST_Subdivide(
    ST_Buffer('SRID=3857;LINESTRING(40 100,98 100,100 150,60 90)'::geometry,
              40, 'endcap=square'),
    12) AS geom
), parcels AS (
  SELECT lpad(row_number() OVER (ORDER BY ST_YMin(geom), ST_XMin(geom))::text, 3, '0') AS ↩
    parcel_id,
         geom,
         ('{residential,commercial}'::text[])[1 + mod(row_number() OVER (ORDER BY ST_YMin( ↩
    geom), ST_XMin(geom)), 2)] AS type
  FROM parcel_geoms
)
SELECT ST_ClusterKMeans(geom, 3) over (PARTITION BY type) AS cid,
       parcel_id,
       type,
       ST_SnapToGrid(geom, 1) AS parcel_geom
FROM parcels
ORDER BY type, parcel_id;
```

Output

```
 cid | parcel_id |    type     | parcel_geom
-----+-----------+-------------+-----------------------------------------------------------
   0 | 001       | commercial  | POLYGON((33 60,0 60,0 98,71 98,71 35,33 60))
   2 | 003       | commercial  | POLYGON((106 98,138 98,137 91,134 84,131 77,126 71,120 66,113 63,106 61,106 98))
   2 | 005       | commercial  | POLYGON((140 148,138 98,72 98,72 148,140 148))
   1 | 007       | commercial  | POLYGON((109 189,116 187,123 183,129 178,134 171,137 164,139 156,140 148,109 148,109 ↩
     189))
   0 | 002       | residential | POLYGON((71 98,106 98,106 61,98 60,88 60,71 35,71 98))
   2 | 004       | residential | POLYGON((0 98,0 140,45 140,67 172,72 178,72 98,0 98))
   1 | 006       | residential | POLYGON((72 178,78 183,85 187,93 189,101 190,109 189,109 148,72 148,72 178))
```

Figure



Example: Clustering a preaggregated planetary-scale data population dataset using 3D clustering and weighting. Identify at least 20 regions based on Kontur Population Data that do not span more than 3000 km from their center:

Code

```
CREATE TABLE kontur_population_3000km_clusters AS
SELECT
    geom,
    ST_ClusterKMeans(
        ST_Force4D(
            ST_Transform(ST_Force3D(geom), 4978),
            mvalue => population
        ),
        20,
        max_radius => 3000000
    ) OVER () AS cid
FROM kontur_population;
```

The clustering produces 46 regions. Clusters are centered at well-populated regions such as New York and Moscow. Greenland forms one cluster, several island clusters span the antimeridian, and cluster edges follow the Earth's curvature.



Kontur population clustered with 3000 km maximum radius.

Cluster weighted city populations on the Earth in geocentric coordinates. The maximum radius is specified in meters, while the returned geometries stay in WGS 84 for display.

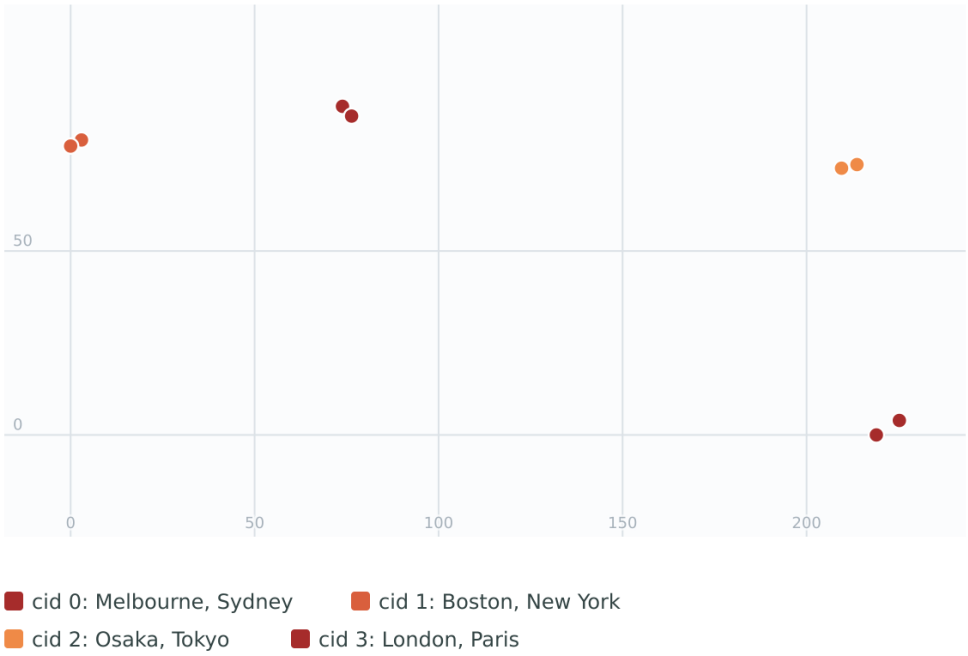
Code

```
WITH population(place, population, geom) AS (
  VALUES
    ('Boston', 5.0, ST_SetSRID(ST_Point(-71.06, 42.36), 4326)),
    ('New York', 20.0, ST_SetSRID(ST_Point(-74.01, 40.71), 4326)),
    ('London', 15.0, ST_SetSRID(ST_Point(-0.13, 51.51), 4326)),
    ('Paris', 11.0, ST_SetSRID(ST_Point(2.35, 48.86), 4326)),
    ('Osaka', 19.0, ST_SetSRID(ST_Point(135.50, 34.69), 4326)),
    ('Tokyo', 37.0, ST_SetSRID(ST_Point(139.69, 35.68), 4326)),
    ('Melbourne', 5.0, ST_SetSRID(ST_Point(144.96, -37.81), 4326)),
    ('Sydney', 5.0, ST_SetSRID(ST_Point(151.21, -33.87), 4326))
), clustered AS (
  SELECT place,
         geom,
         ST_ClusterKMeans(
           ST_Force4D(
             ST_Transform(ST_Force3D(geom), 4978),
             mvalue => population
           ),
           4,
           max_radius => 3000000
         ) OVER (ORDER BY place) AS cid
  FROM population
)
SELECT cid,
       string_agg(place, ', ' ORDER BY place) AS places,
       ST_Collect(geom ORDER BY place) AS cluster_geom
FROM clustered
GROUP BY cid
ORDER BY cid;
```

Output

```
0 | Melbourne, Sydney | MULTIPOINT((144.96 -37.81),(151.21 -33.87))
1 | Boston, New York | MULTIPOINT((-71.06 42.36),(-74.01 40.71))
2 | Osaka, Tokyo | MULTIPOINT((135.5 34.69),(139.69 35.68))
3 | London, Paris | MULTIPOINT((-0.13 51.51),(2.35 48.86))
```

Figure



See Also

ST_ClusterDBSCAN, ST_ClusterIntersectingWin, ST_ClusterWithinWin, ST_ClusterIntersecting, ST_ClusterWithin, ST_Subdivide, ST_Force3D, ST_Force4D,

7.17.7 ST_ClusterWithin

ST_ClusterWithin — Aggregate function that clusters geometries by separation distance.

Synopsis

geometry[] **ST_ClusterWithin**(geometry set g, float8 distance);

Description

An aggregate function that returns an array of GeometryCollections, where each collection is a cluster containing some input geometries. Clustering partitions the input geometries into sets in which each geometry is within the specified *`distance`* of at least one other geometry in the same cluster. Distances are Cartesian distances in the units of the SRID.

ST_ClusterWithin is equivalent to running ST_ClusterDBSCAN with `minpoints => 0`.

Availability: 2.2.0

 This method supports Circular Strings and Curves.

Examples

Code

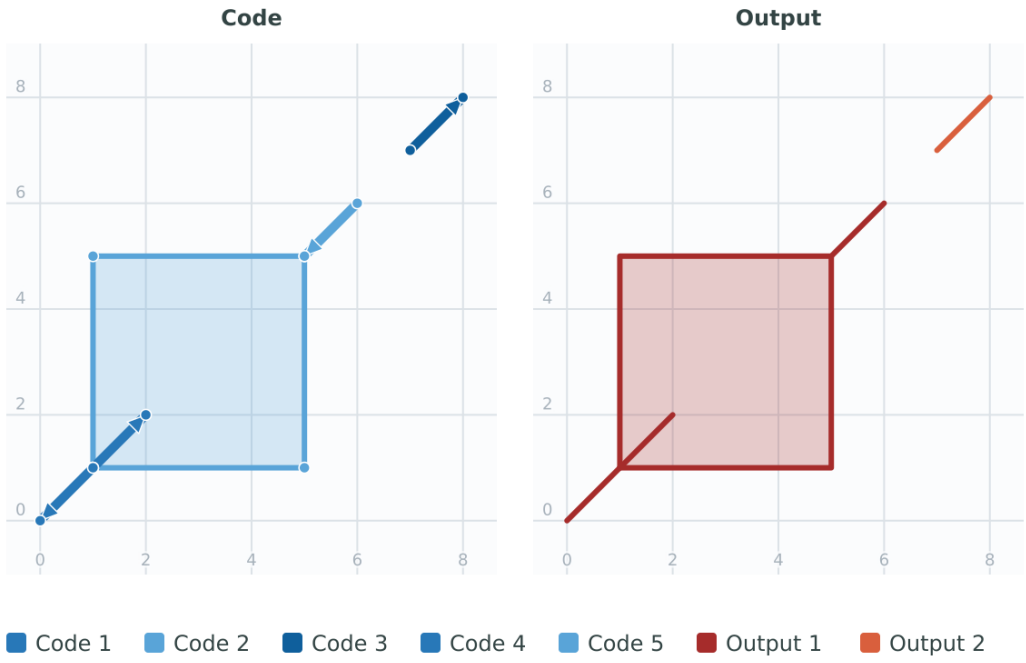
```
WITH testdata AS
  (SELECT unnest(ARRAY['LINESTRING (0 0,1 1)'::geometry,
                'LINESTRING (5 5,4 4)'::geometry,
                'LINESTRING (6 6,7 7)'::geometry,
                'LINESTRING (0 0,-1 -1)'::geometry,
                'POLYGON ((0 0,4 0,4 4,0 4,0 0))'::geometry]) AS geom)

SELECT unnest(ST_ClusterWithin(geom, 1.4)) FROM testdata;
```

Output

```
GEOMETRYCOLLECTION(LINESTRING(0 0,1 1),LINESTRING(5 5,4 4),LINESTRING(0 0,-1 -1),POLYGON((0 0,4 0,4 4,0 4,0 0)))
GEOMETRYCOLLECTION(LINESTRING(6 6,7 7))
```

Figure



See Also

ST_ClusterWithinWin, ST_ClusterDBSCAN, ST_ClusterIntersecting, ST_ClusterIntersectingWin

7.17.8 ST_ClusterWithinWin

ST_ClusterWithinWin — Window function that returns a cluster id for each input geometry, clustering using separation distance.

Synopsis

integer **ST_ClusterWithinWin**(geometry winset geom, float8 distance);

Description

A window function that returns a cluster number for each input geometry. Clustering partitions the geometries into sets in which each geometry is within the specified `distance` of at least one other geometry in the same cluster. Distances are Cartesian distances in the units of the SRID.

ST_ClusterWithinWin is equivalent to running ST_ClusterDBSCAN with `minpoints => 0`.

Availability: 3.4.0

 This method supports Circular Strings and Curves.

Examples

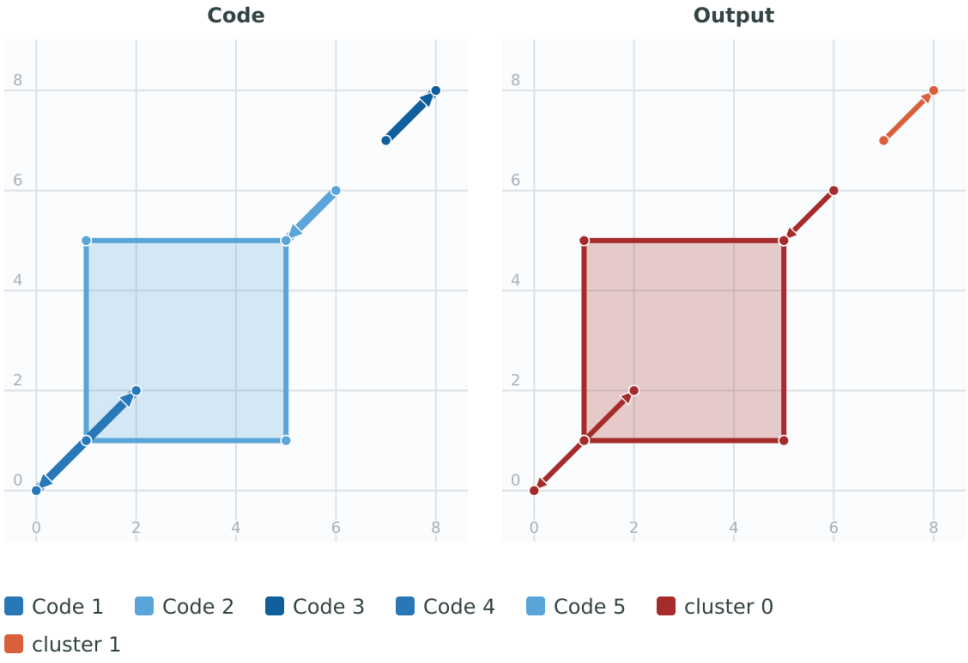
Code

```
WITH testdata AS (
  SELECT id, geom::geometry FROM (
  VALUES  (1, 'LINESTRING (0 0,1 1)'),
          (2, 'LINESTRING (5 5,4 4)'),
          (3, 'LINESTRING (6 6,7 7)'),
          (4, 'LINESTRING (0 0,-1 -1)'),
          (5, 'POLYGON ((0 0,4 0,4 4,0 4,0 0))')) AS t(id, geom)
)
SELECT id,
  geom,
  ST_ClusterWithinWin(geom, 1.4) OVER () AS cluster
FROM testdata;
```

Output

```
 id |           st_astext             | cluster
----+---------------------------------+---------
  1 | LINESTRING(0 0,1 1)             |       0
  2 | LINESTRING(5 5,4 4)             |       0
  3 | LINESTRING(6 6,7 7)             |       1
  4 | LINESTRING(0 0,-1 -1)           |       0
  5 | POLYGON((0 0,4 0,4 4,0 4,0 0)) |       0
```

Figure



See Also

ST_ClusterWithin, ST_ClusterDBSCAN, ST_ClusterIntersecting, ST_ClusterIntersectingWin,

7.18 Bounding Box Functions

7.18.1 Box2D

Box2D — Returns a BOX2D representing the 2D extent of a geometry.

Synopsis

box2d **Box2D**(geometry geom);

Description

Returns a box2d representing the 2D extent of the geometry.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

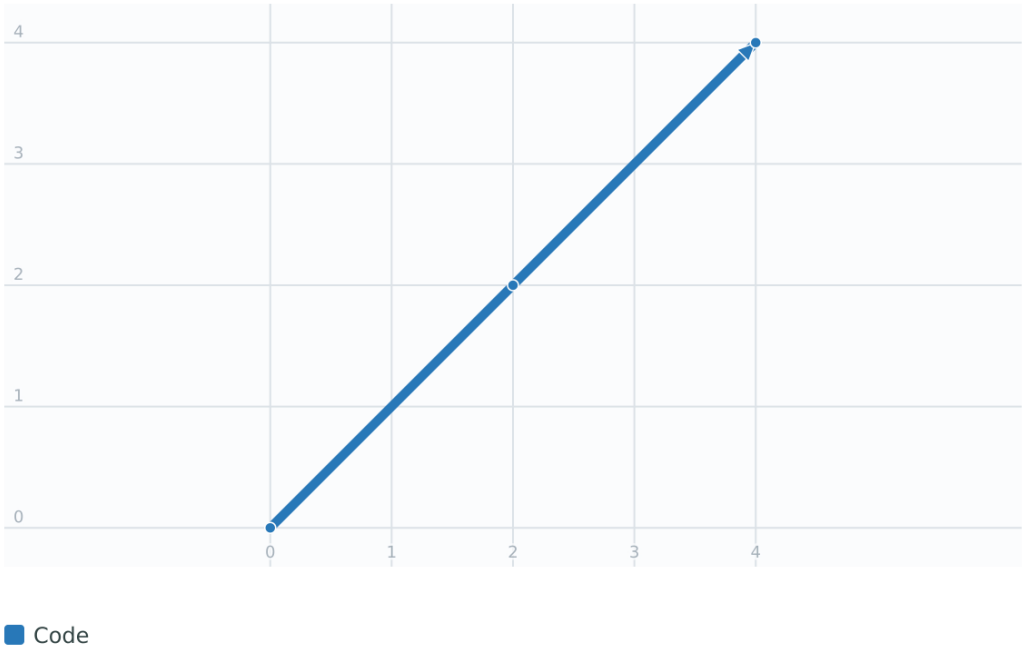
Code

```
SELECT Box2D('LINESTRING(1 2,3 4,5 6)'::geometry);
```

Output

```
BOX(1 2,5 6)
```

Figure



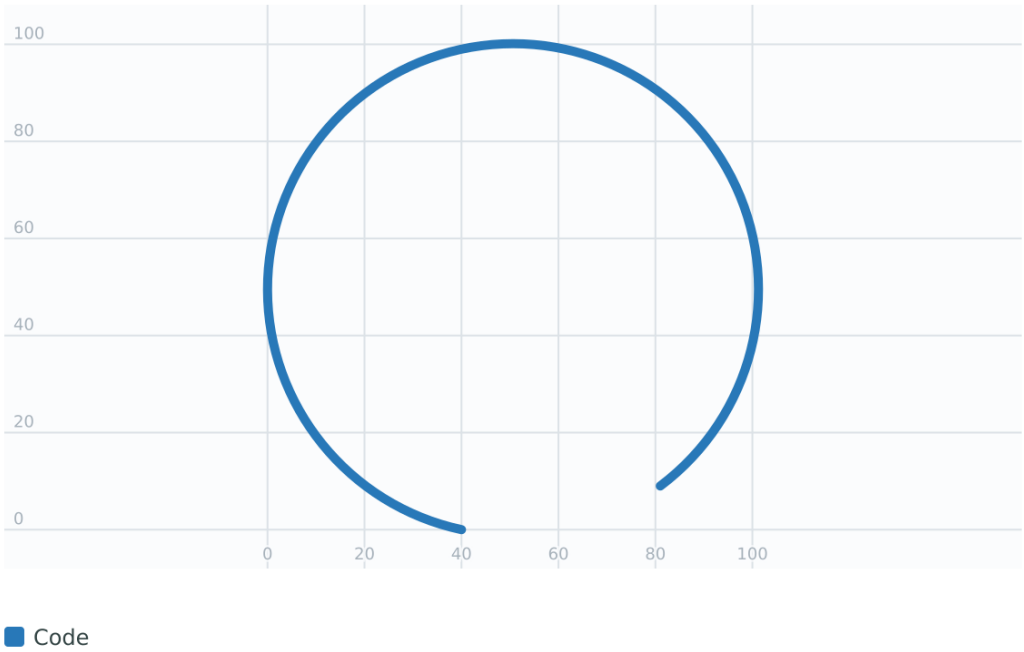
Code

```
SELECT Box2D('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)'::geometry);
```

Output

```
BOX(220186.99512189245 150406,220288.24878054656 150506.12682932706)
```

Figure



See Also

Box3D, ST_GeomFromText

7.18.2 Box3D

Box3D — Returns a BOX3D representing the 3D extent of a geometry.

Synopsis

box3d **Box3D**(geometry geom);

Description

Returns a box3d representing the 3D extent of the geometry.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

 This function supports 3d and will not drop the z-index.

Examples

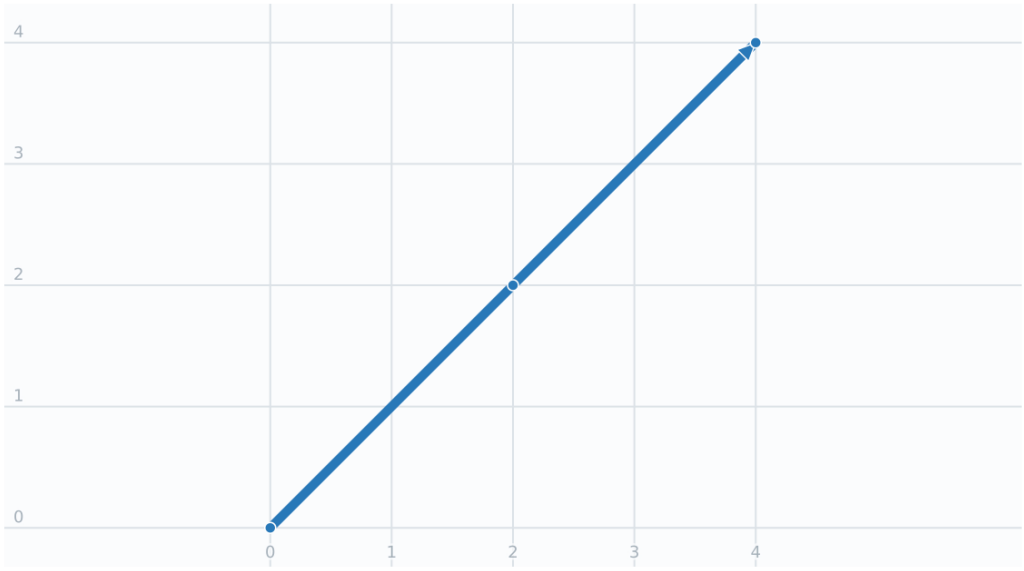
Code

```
SELECT Box3D('LINESTRING(1 2 3,3 4 5,5 6 5)'::geometry);
```

Output

```
BOX3D(1 2 3,5 6 5)
```

Figure



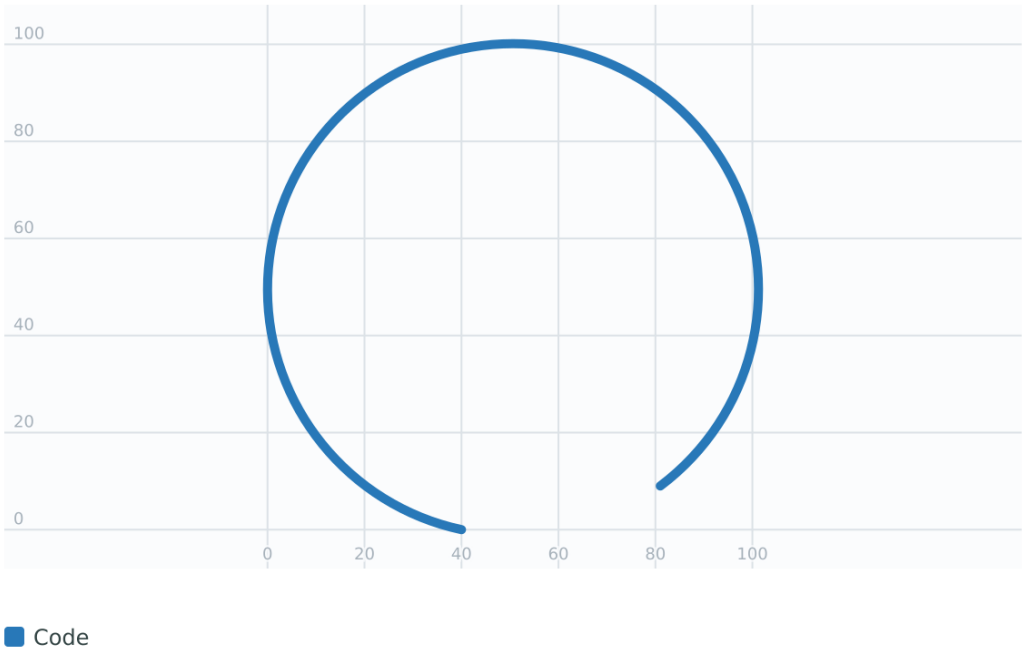
Code

```
SELECT Box3D('CIRCULARSTRING(220268 150415 1,220227 150505 1,220227 150406 1)'::geometry);
```

Output

```
BOX3D(220186.99512189245 150406 1,220288.24878054656 150506.12682932706 1)
```

Figure



See Also

Box2D, ST_GeomFromEWKT

7.18.3 ST_EstimatedExtent

ST_EstimatedExtent — Returns the estimated extent of a spatial table.

Synopsis

box2d **ST_EstimatedExtent**(text schema_name, text table_name, text geocolumn_name, boolean parent_only);
box2d **ST_EstimatedExtent**(text schema_name, text table_name, text geocolumn_name);
box2d **ST_EstimatedExtent**(text table_name, text geocolumn_name);

Description

Returns the estimated extent of a spatial table as a box2d. The current schema is used if not specified. The estimated extent is taken from the geometry column's statistics. This is usually much faster than computing the exact extent of the table using ST_Extent or ST_3DExtent.

The default behavior is to also use statistics collected from child tables (tables with INHERITS) if available. If `parent_only` is set to TRUE, only statistics for the given table are used and child tables are ignored.

For PostgreSQL >= 8.0.0 statistics are gathered by VACUUM ANALYZE and the result extent will be about 95% of the actual one. For PostgreSQL < 8.0.0 statistics are gathered by running `update_geometry_stats()` and the result extent is exact.



Note
In the absence of statistics (empty table or no ANALYZE called) this function returns NULL. Prior to version 1.5.4 an exception was thrown instead.



Note
Escaping names for tables and/or namespaces that include special characters and quotes may require special handling. A user notes: "For schemas and tables, use identifier escaping rules to produce a double-quoted string, and afterwards remove the first and last double-quote character. For geometry column pass as is."

Availability: 1.0.0

Changed: 2.1.0. Up to 2.0.x this was called ST_Estimated_Extent.



This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_EstimatedExtent('ny', 'edges', 'geom');
```

Output

```
               st_estimatedextent
-------------------------------------------------
 BOX(-8877653 4912316,-8010225.5 5589284)
(1 row)
```

Code

```
SELECT ST_EstimatedExtent('feature_poly', 'geom');
```

Output

```
                        st_estimatedextent
------------------------------------------------------------------
 BOX(-124.659652709961 24.6830825805664,-67.7798080444336 49.0012092590332)
(1 row)
```

See Also

ST_Extent, ST_3DExtent

7.18.4 ST_Expand

ST_Expand — Returns a bounding box expanded from another bounding box or a geometry.

Synopsis

geometry **ST_Expand**(geometry geom, float units_to_expand);
geometry **ST_Expand**(geometry geom, float dx, float dy, float dz=0, float dm=0);
box2d **ST_Expand**(box2d box, float units_to_expand);
box2d **ST_Expand**(box2d box, float dx, float dy);
box3d **ST_Expand**(box3d box, float units_to_expand);
box3d **ST_Expand**(box3d box, float dx, float dy, float dz=0);

Description

Returns a bounding box expanded from the bounding box of the input, either by specifying a single distance with which the box should be expanded on both axes, or by specifying an expansion distance for each axis. Uses double-precision. Can be used for distance queries, or to add a bounding box filter to a query to take advantage of a spatial index.

In addition to the version of ST_Expand accepting and returning a geometry, variants are provided that accept and return box2d and box3d data types.

Distances are in the units of the spatial reference system of the input.

ST_Expand is similar to ST_Buffer, except while buffering expands a geometry in all directions, ST_Expand expands the bounding box along each axis.



Note
Pre version 1.3, ST_Expand was used in conjunction with ST_Distance to do indexable distance queries. For example, `geom && ST_Expand('POINT(10 20)', 10) AND ST_Distance(geom, 'POINT(10 20)') < 10`. This has been replaced by the simpler and more efficient ST_DWithin function.

Availability: 1.5.0 behavior changed to output double precision instead of float4 coordinates.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Enhanced: 2.3.0 support was added to expand a box by different amounts in different dimensions.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Note
Examples below use US National Atlas Equal Area (SRID=2163) which is a meter projection

---

10-meter expanded box around bbox of a linestring.

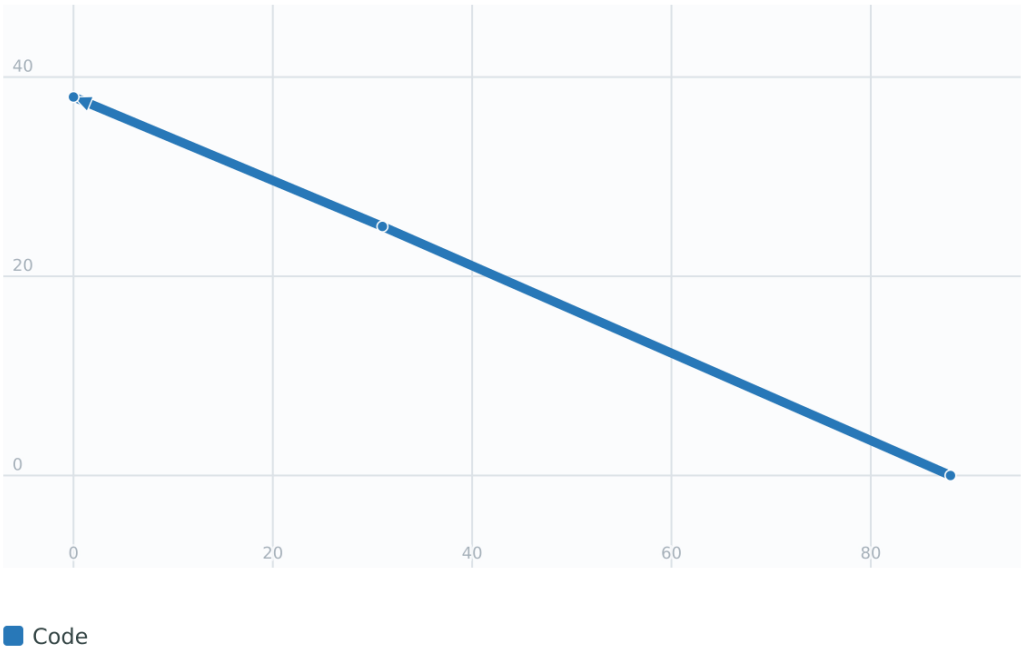
Code

```
SELECT CAST(ST_Expand(ST_GeomFromText('LINESTRING(2312980 110676,2312923 110701,2312892 ↩
    110714)', 2163), 10) As box2d);
```

Output

```
BOX(2312882 110666,2312990 110724)
```

Figure



This example expands a 3D box by 10 meters.

Code

```
SELECT ST_Expand(CAST('BOX3D(778783 2951741 1,794875 2970042.61545891 10)' AS box3d), 10)
```

Output

```
BOX3D(778773 2951731 -9,794885 2970052.61545891 20)
```

This example renders the text representation of a 10-meter expanded box around a point geometry.

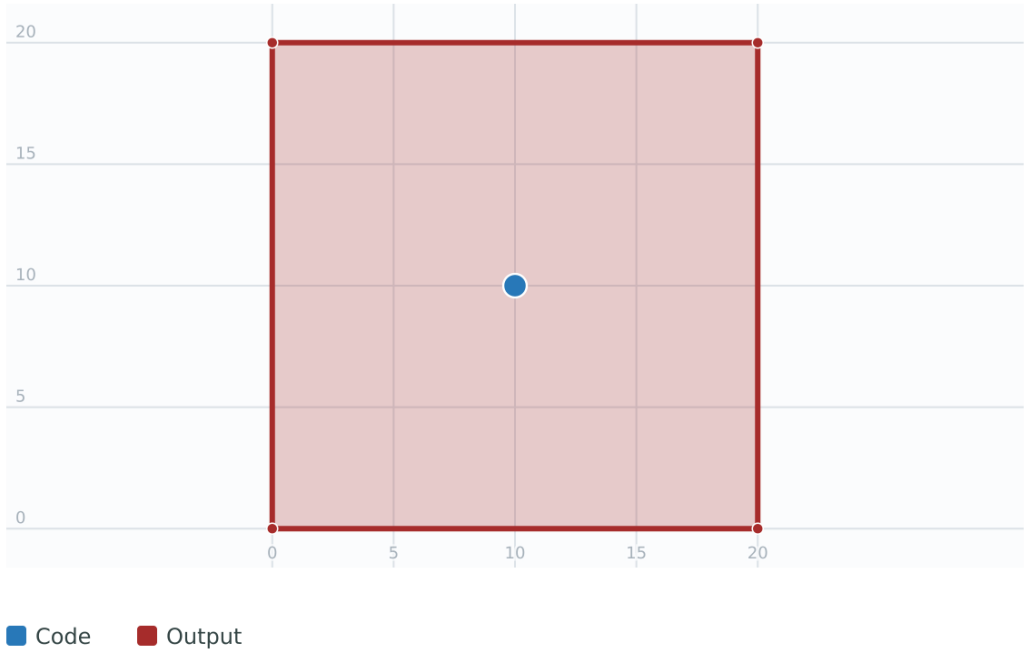
Code

```
SELECT ST_Expand('SRID=2163;POINT(2312980 110676)'::geometry, 10);
```

Output

```
SRID=2163;POLYGON((2312970 110666,2312970 110686,2312990 110686,2312990 110666,2312970 110666))
```

Figure



See Also

ST_Buffer, ST_DWithin, ST_SRID

7.18.5 ST_Extent

ST_Extent — Aggregate function that returns the bounding box of geometries.

Synopsis

box2d **ST_Extent**(geometry set geomfield);

Description

An aggregate function that returns a box2d bounding box that bounds a set of geometries.

The bounding box coordinates are in the spatial reference system of the input geometries.

ST_Extent is similar in concept to Oracle Spatial/Locator's SDO_AGGR_MBR.



Note
ST_Extent returns boxes with only X and Y ordinates even with 3D geometries. To return XYZ ordinates use ST_3DExtent.



Note
The returned `box3d` value does not include a SRID. Use ST_SetSRID to convert it into a geometry with SRID metadata. The SRID is the same as the input geometries.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples



Note
Examples below use Massachusetts State Plane ft (SRID=2249)

Code

```
SELECT ST_Extent(geom) AS bextent FROM sometable;
```

Output

```
BOX(739651.875 2908247.25,794875.8125 2970042.75)
```

Return extent of each geometry category.

Code

```
SELECT ST_Extent(geom) as bextent
FROM sometable
GROUP BY category ORDER BY category;
```

Output

```
                    bextent                     |         name
------------------------------------------------+----------------
 BOX(778783.5625 2951741.25,794875.8125 2970042.75) | A
 BOX(751315.8125 2919164.75,765202.6875 2935417.25) | B
 BOX(739651.875 2917394.75,756688.375 2935866)      | C
```

This example converts the extent back into a geometry and renders its extended text representation.

Code

```
SELECT ST_SetSRID(ST_Extent(geom), 2249) AS bextent FROM sometable;
```

Output

```
 SRID=2249;POLYGON((739651.875 2908247.25,739651.875 2970042.75,794875.8125 2970042.75,
794875.8125 2908247.25,739651.875 2908247.25))
```

See Also

ST_EstimatedExtent, ST_3DExtent, ST_SetSRID

7.18.6 ST_3DExtent

ST_3DExtent — Aggregate function that returns the 3D bounding box of geometries.

Synopsis

box3d **ST_3DExtent**(geometry set geomfield);

Description

An aggregate function that returns a box3d (includes Z ordinate) bounding box that bounds a set of geometries.

The bounding box coordinates are in the spatial reference system of the input geometries.



Note

The returned `box3d` value does not include a SRID. Use ST_SetSRID to convert it into a geometry with SRID metadata. The SRID is the same as the input geometries.

Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced.

Changed: 2.0.0 In prior versions this used to be called ST_Extent3D

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Code

```
SELECT ST_3DExtent(foo.geom) As b3extent
FROM (SELECT ST_MakePoint(x, y, z) As geom
    FROM generate_series(1, 3) As x
        CROSS JOIN generate_series(1, 2) As y
        CROSS JOIN generate_series(0, 2) As Z) As foo;
```

Output

```
BOX3D(1 1 0,3 2 2)
```

This example returns the extent of several elevated CircularStrings.

Code

```
SELECT ST_3DExtent(foo.geom) As b3extent
FROM (SELECT ST_Translate(ST_Force3DZ(ST_LineToCurve(ST_Buffer(ST_Point(x, y), 1))), 0, 0, z ↩
    ) As geom
    FROM generate_series(1, 3) As x
        CROSS JOIN generate_series(1, 2) As y
        CROSS JOIN generate_series(0, 2) As Z) As foo;
```

Output

```
BOX3D(0 0 0,4 3 2)
```

See Also

ST_Extent, ST_Force3DZ, ST_SetSRID

7.18.7 ST_MakeBox2D

ST_MakeBox2D — Creates a BOX2D defined by two 2D point geometries.

Synopsis

box2d **ST_MakeBox2D**(geometry pointLowLeft, geometry pointUpRight);

Description

Creates a box2d defined by two Point geometries. This is useful for doing range queries.

Examples

Return all features that fall within or partly reside in a US National Atlas coordinate bounding box. It is assumed here that the geometries are stored with SRID=2163 (US National Atlas equal area).

Code

```
SELECT feature_id, feature_name, geom
FROM features
WHERE geom && ST_SetSRID(
  ST_MakeBox2D(
    ST_Point(-989502.1875, 528439.5625),
    ST_Point(-987121.375, 529933.1875)
  ),
  2163
);
```

See Also

ST_Point, ST_SetSRID, ST_SRID

7.18.8 ST_3DMakeBox

ST_3DMakeBox — Creates a BOX3D defined by two 3D point geometries.

Synopsis

box3d **ST_3DMakeBox**(geometry point3DLowLeftBottom, geometry point3DUpRightTop);

Description

Creates a box3d defined by two 3D Point geometries.

 This function supports 3D and will not drop the z-index.

Changed: 2.0.0 In prior versions this used to be called ST_MakeBox3D

Examples

Code

```
SELECT ST_3DMakeBox(ST_MakePoint(-989502.1875, 528439.5625, 10),
    ST_MakePoint(-987121.375 , 529933.1875, 10))
```

Output

```
BOX3D(-989502.1875 528439.5625 10,-987121.375 529933.1875 10)
```

See Also

ST_MakePoint, ST_SetSRID, ST_SRID

7.18.9 ST_XMax

ST_XMax — Returns the X maxima of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_XMax**(box3d aGeomorBox2DorBox3D);

Description

Returns the X maxima of a 2D or 3D bounding box or a geometry.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However, it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_XMax('BOX3D(1 2 3,4 5 6)');
```

Output

```
4
```

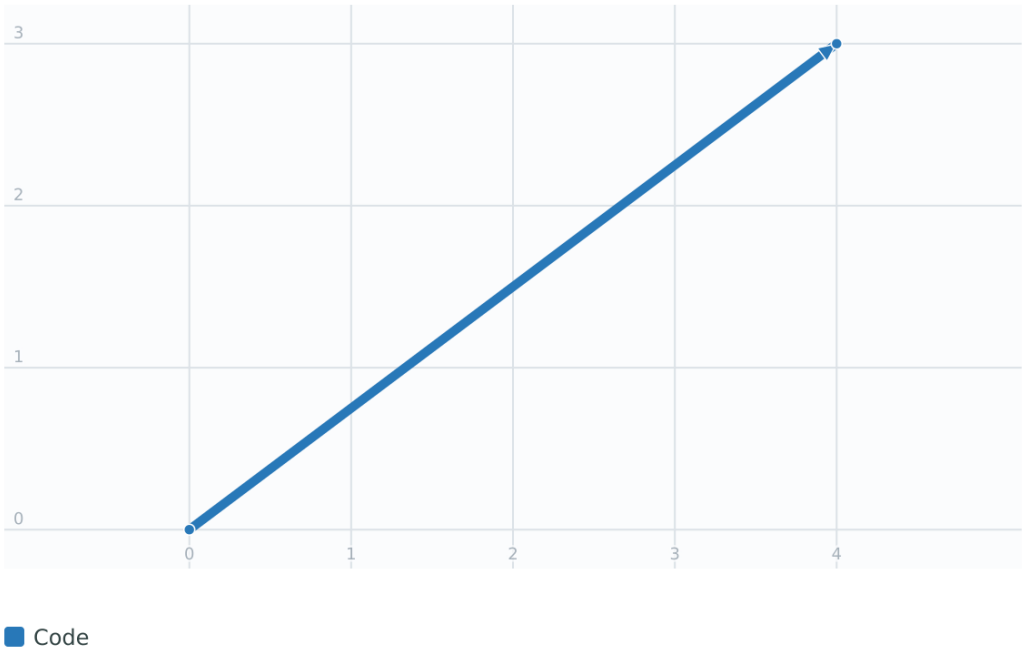
Code

```
SELECT ST_XMax('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
5
```

Figure



Code

```
SELECT ST_XMax(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
3
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_XMax('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

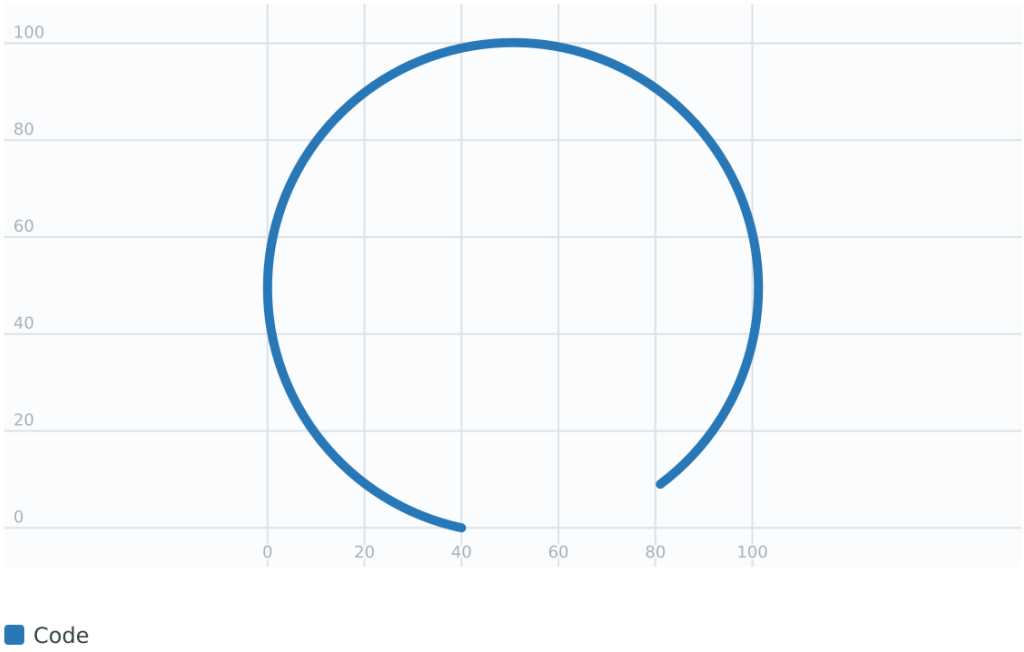
Code

```
SELECT ST_XMax('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry);
```

Output

```
220288.248780547
```

Figure



See Also

ST_XMin, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.10 ST_XMin

ST_XMin — Returns the X minima of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_XMin**(box3d aGeomorBox2DorBox3D);

Description

Returns the X minima of a 2D or 3D bounding box or a geometry.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_XMin('BOX3D(1 2 3,4 5 6)');
```

Output

```
1
```

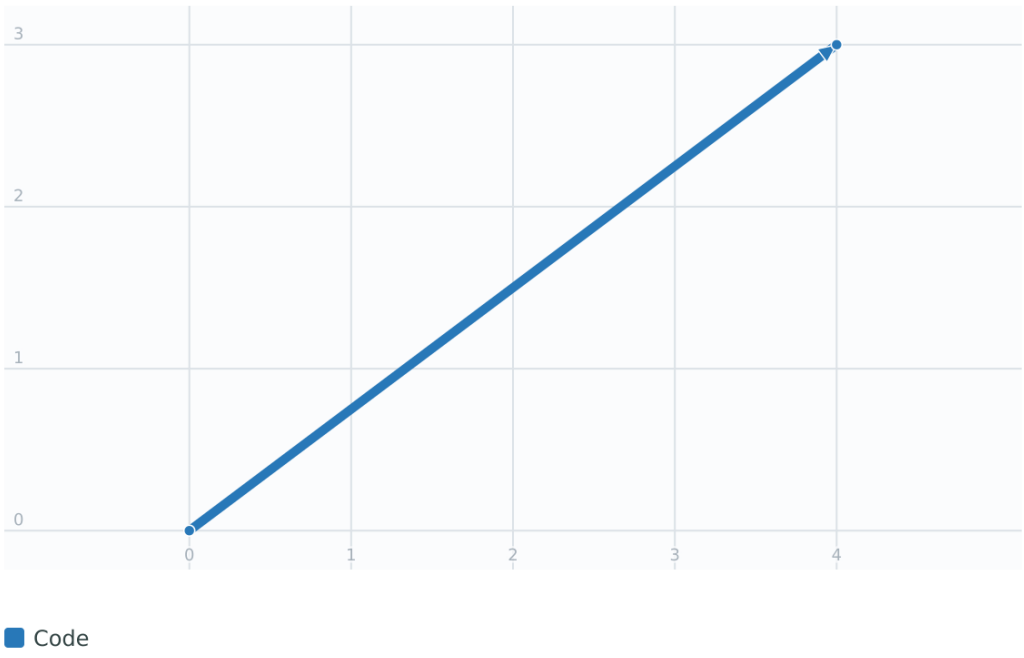
Code

```
SELECT ST_XMin('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
1
```

Figure



Code

```
SELECT ST_XMin(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
-3
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_XMin('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

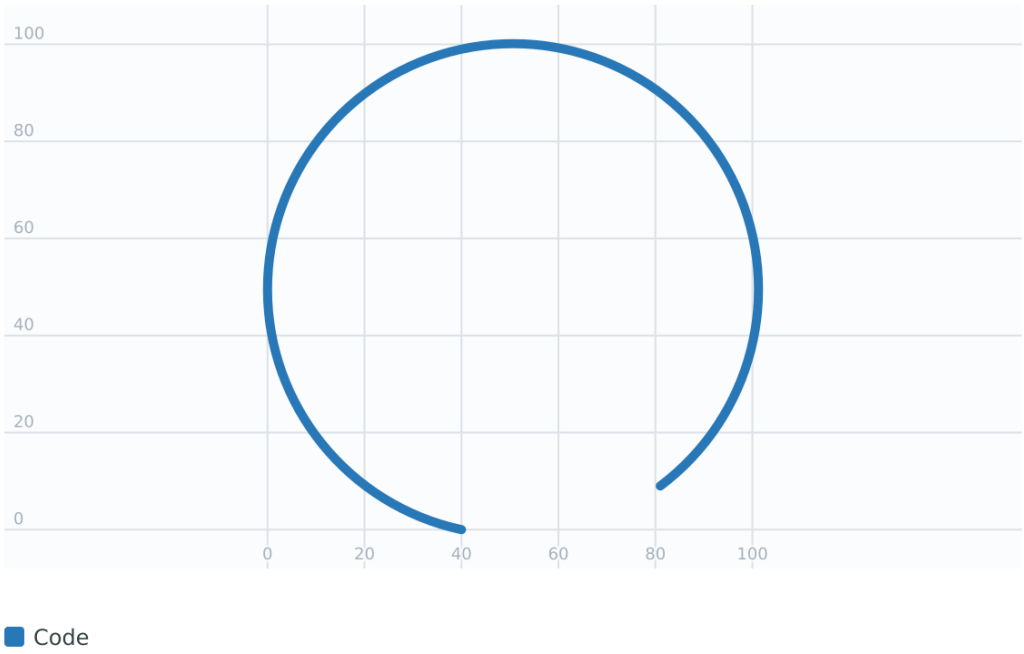
Code

```
SELECT ST_XMin('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry);
```

Output

```
220186.995121892
```

Figure



See Also

ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.11 ST_YMax

ST_YMax — Returns the Y maxima of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_YMax**(box3d aGeomorBox2DorBox3D);

Description

Returns the Y maxima of a 2D or 3D bounding box or a geometry.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_YMax('BOX3D(1 2 3,4 5 6)');
```

Output

```
5
```

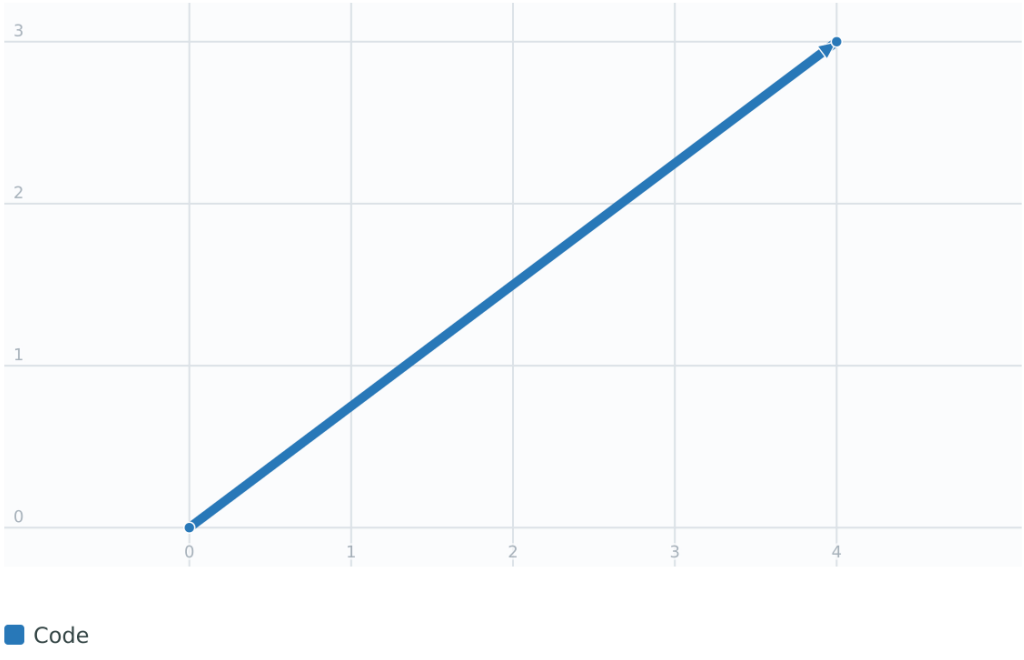
Code

```
SELECT ST_YMax('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
6
```

Figure



Code

```
SELECT ST_YMax(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
4
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_YMax('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

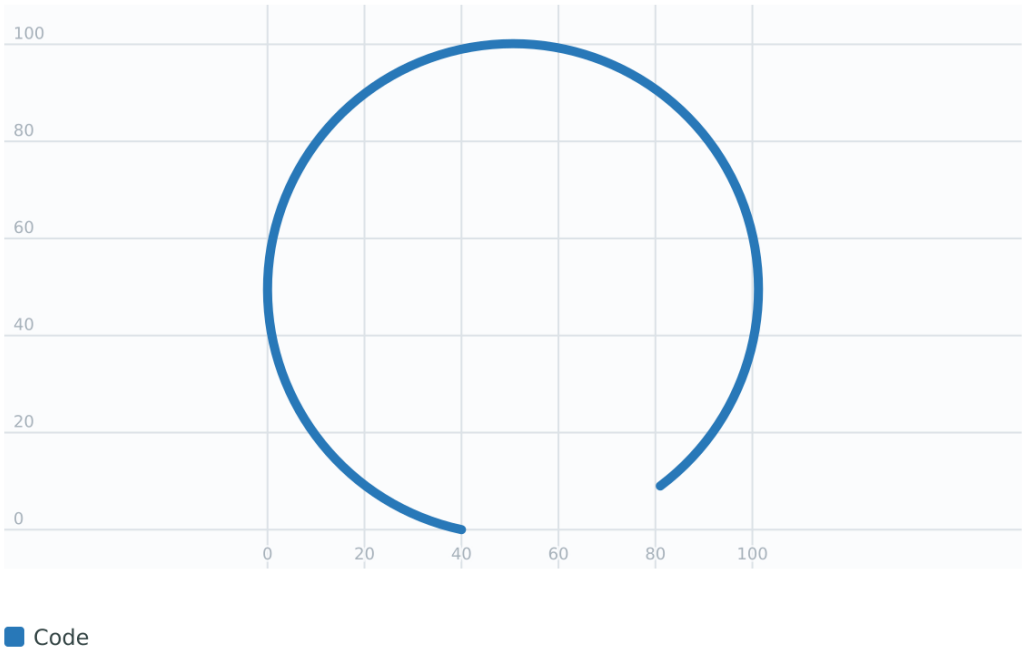
Code

```
SELECT ST_YMax('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry);
```

Output

```
150506.126829327
```

Figure



See Also

ST_XMin, ST_XMax, ST_YMin, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.12 ST_YMin

ST_YMin — Returns the Y minima of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_YMin**(box3d aGeomorBox2DorBox3D);

Description

Returns the Y minima of a 2D or 3D bounding box or a geometry.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_YMin('BOX3D(1 2 3,4 5 6)');
```

Output

```
2
```

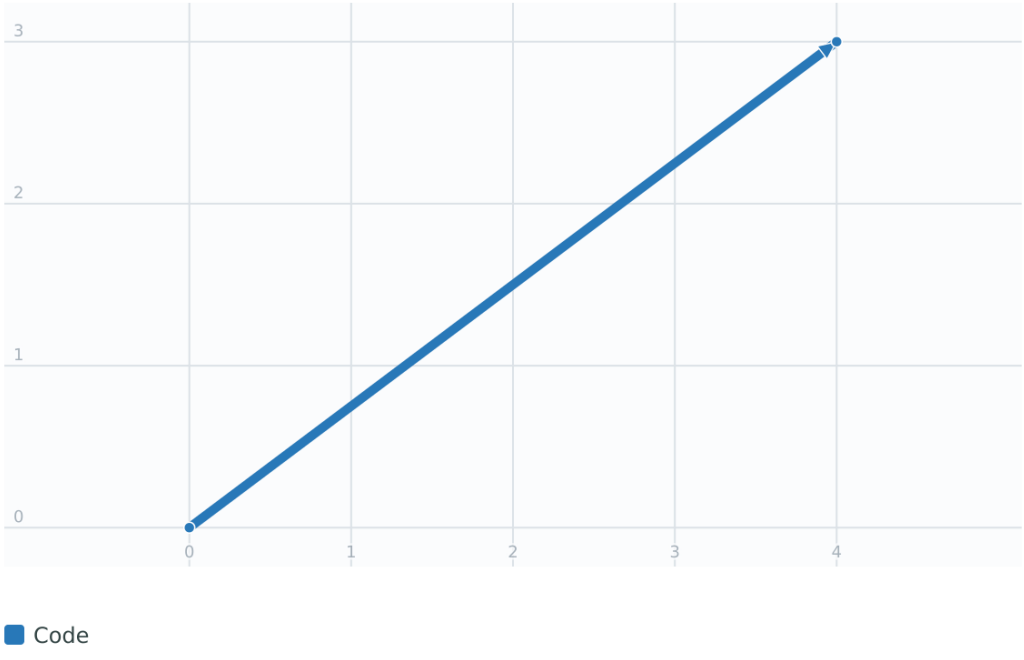
Code

```
SELECT ST_YMin('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
3
```

Figure



Code

```
SELECT ST_YMin(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
2
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_YMin('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

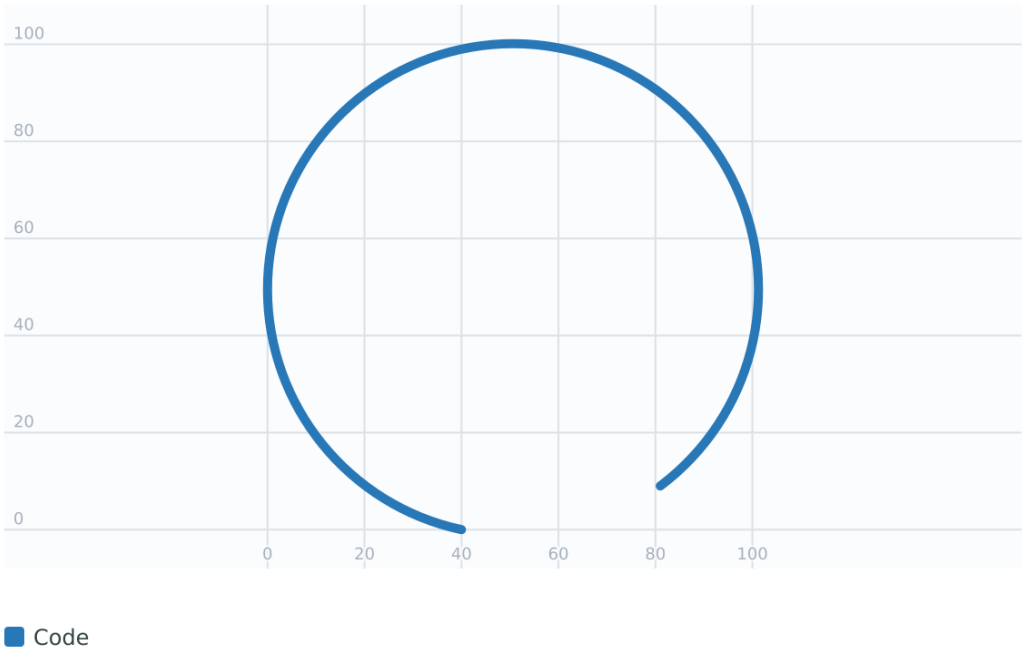
Code

```
SELECT ST_YMin('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry);
```

Output

```
150406
```

Figure



See Also

ST_GeomFromEWKT, ST_XMin, ST_XMax, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.13 ST_ZMax

ST_ZMax — Returns the Z maxima of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_ZMax**(box3d aGeomorBox2DorBox3D);

Description

Returns the Z maxima of a 2D or 3D bounding box or a geometry.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_ZMax('BOX3D(1 2 3,4 5 6)');
```

Output

```
6
```

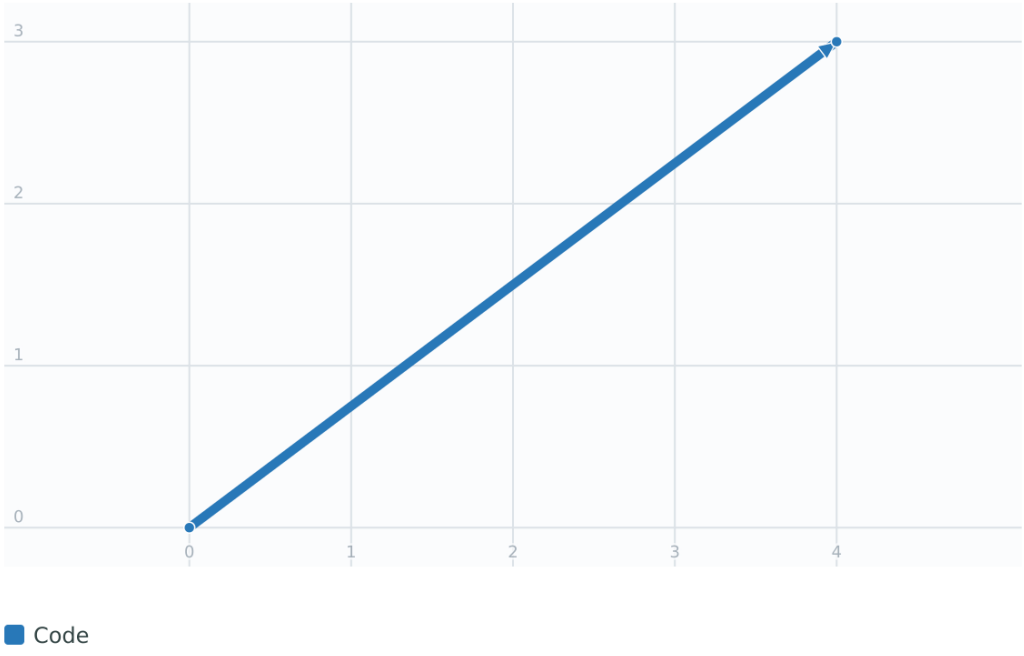
Code

```
SELECT ST_ZMax('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
7
```

Figure



Code

```
SELECT ST_ZMax('BOX3D(-3 2 1,3 4 1)');
```

Output

```
1
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_ZMax('LINESTRING(1 3 4,5 6 7)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

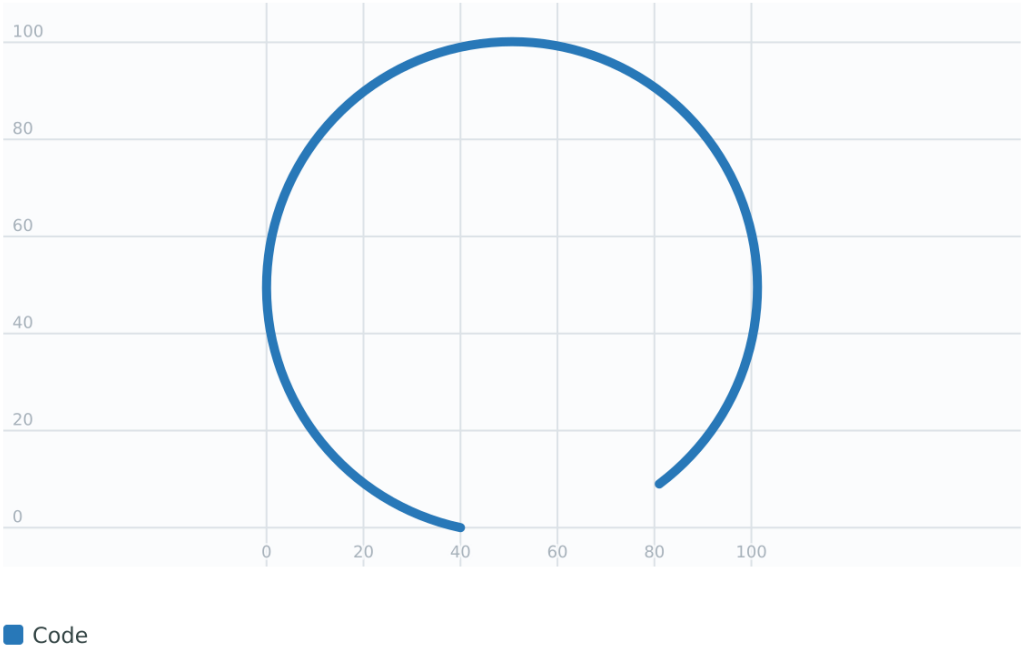
Code

```
SELECT ST_ZMax('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry);
```

Output

```
3
```

Figure



See Also

ST_GeomFromEWKT, ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_MMin, ST_MMax

7.18.14 ST_ZMin

ST_ZMin — Returns the Z minima of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_ZMin**(box3d aGeomorBox2DorBox3D);

Description

Returns the Z minima of a 2D or 3D bounding box or a geometry.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_ZMin('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

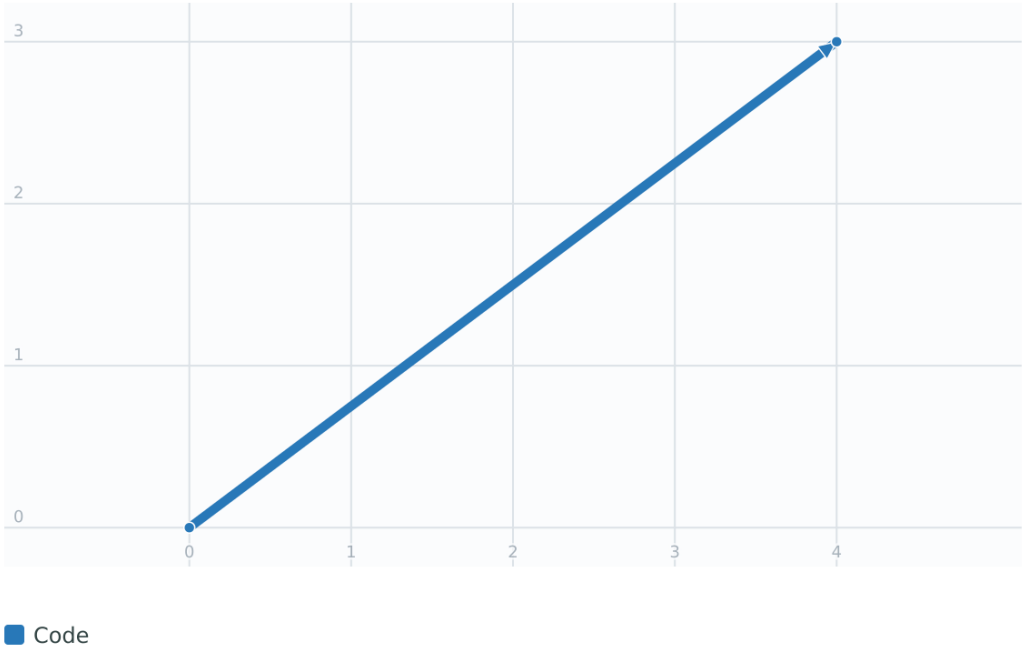
Code

```
SELECT ST_ZMin('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
4
```

Figure



Code

```
SELECT ST_ZMin('BOX3D(-3 2 1,3 4 1)');
```

Output

```
1
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_ZMin('LINESTRING(1 3 4,5 6 7)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

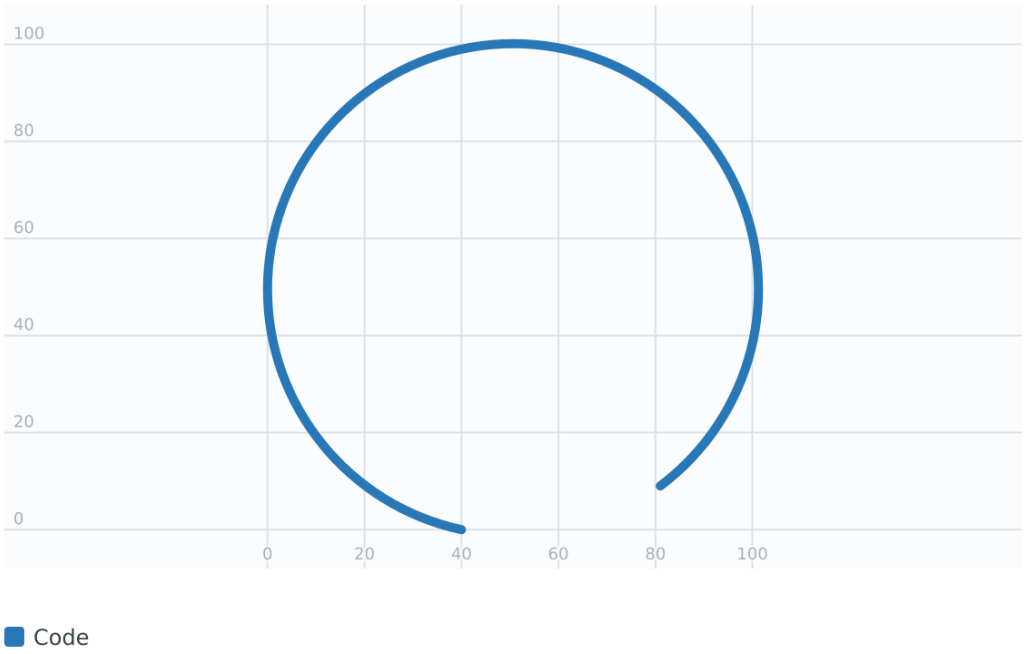
Code

```
SELECT ST_ZMin('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry);
```

Output

```
1
```

Figure



See Also

ST_GeomFromEWKT, ST_GeomFromText, ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMax, ST_MMin, ST_MMax

7.18.15 ST_XSize

ST_XSize — Returns the X size of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_XSize**(box3d aGeomorBox2DorBox3D);

Description

Returns the X size of a 2D or 3D bounding box or a geometry, computed as `ST_XMax(aGeomorBox2DorBox3D) - ST_XMin(aGeomorBox2DorBox3D)`.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Availability: 3.7.0

Examples

Code

```
SELECT ST_XSize('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

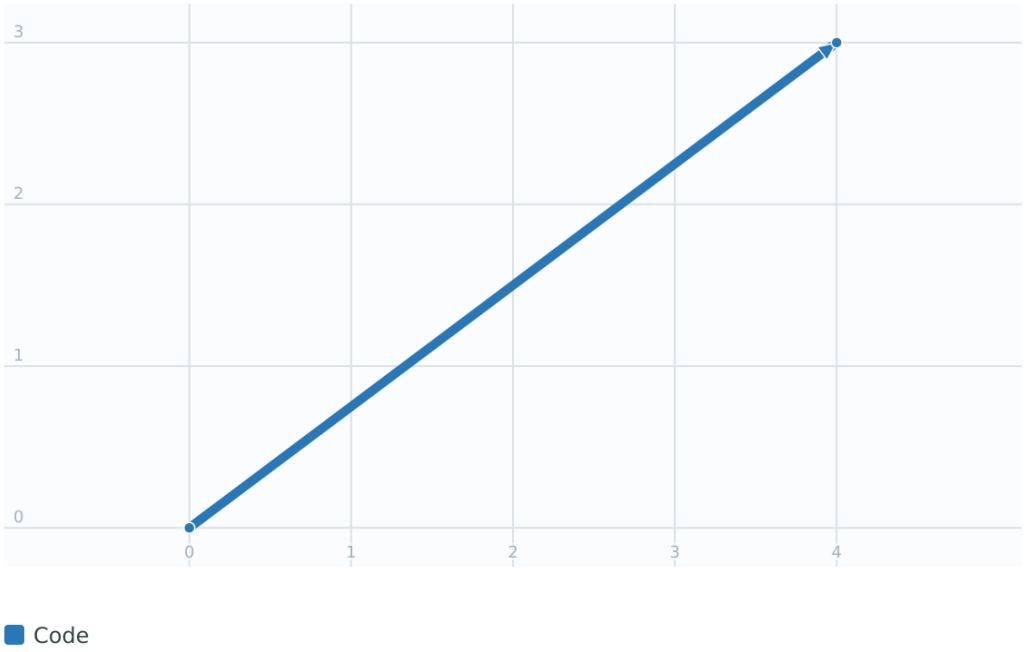
Code

```
SELECT ST_XSize('LINESTRING(1 3,5 6)'::geometry);
```

Output

```
4
```

Figure



See Also

ST_XMin, ST_XMax, ST_YSize, ST_ZSize, ST_MSize

7.18.16 ST_YSize

ST_YSize — Returns the Y size of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_YSize**(box3d aGeomorBox2DorBox3D);

Description

Returns the Y size of a 2D or 3D bounding box or a geometry, computed as `ST_YMax(aGeomorBox2DorBox3D) - ST_YMin(aGeomorBox2DorBox3D)`.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.

Availability: 3.7.0

Examples

Code

```
SELECT ST_YSize('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

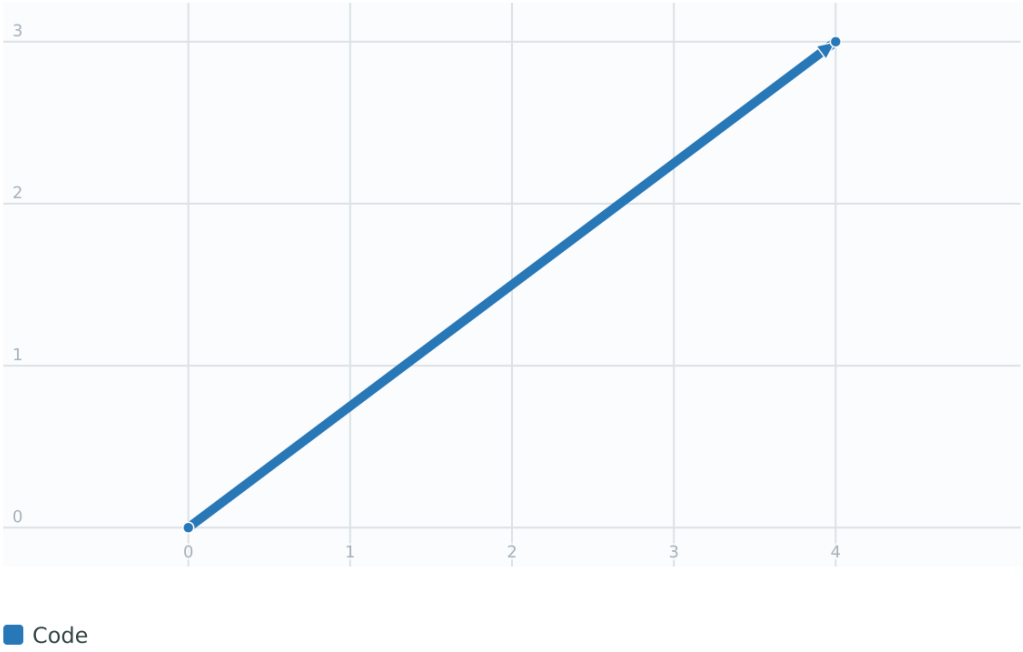
Code

```
SELECT ST_YSize('LINESTRING(1 3,5 6)'::geometry);
```

Output

```
3
```

Figure



See Also

ST_YMin, ST_YMax, ST_XSize, ST_ZSize, ST_MSize

7.18.17 ST_ZSize

ST_ZSize — Returns the Z size of a 2D or 3D bounding box or a geometry.

Synopsis

float **ST_ZSize**(box3d aGeomorBox2DorBox3D);

Description

Returns the Z size of a 2D or 3D bounding box or a geometry, computed as `ST_ZMax(aGeomorBox2DorBox3D) - ST_ZMin(aGeomorBox2DorBox3D)`.



Note
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Availability: 3.7.0

Examples

Code

```
SELECT ST_ZSize('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

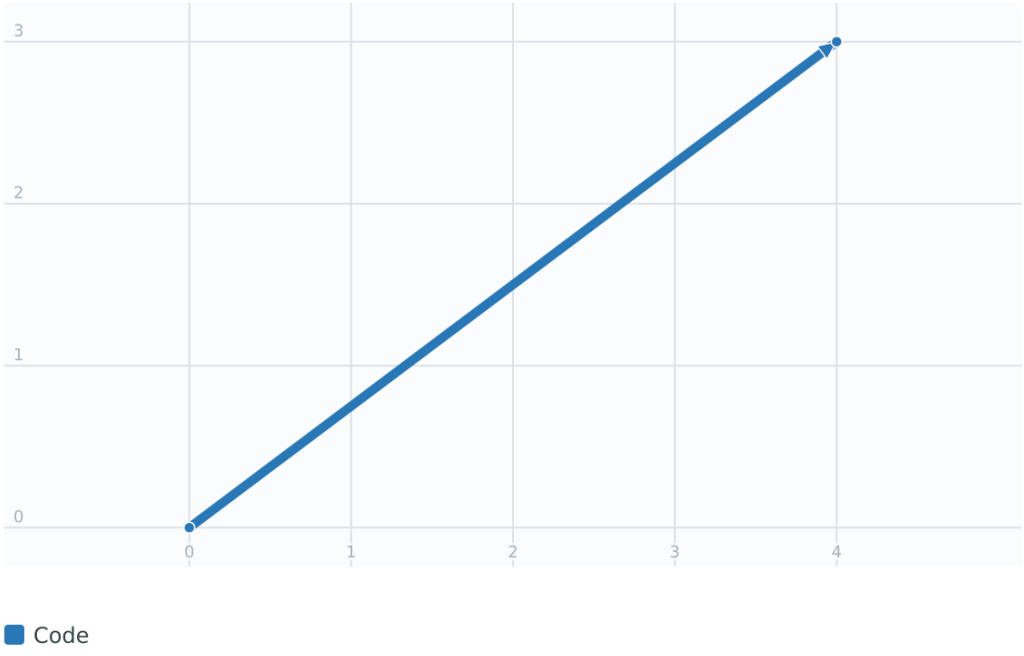
Code

```
SELECT ST_ZSize('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
3
```

Figure



See Also

ST_ZMin, ST_ZMax, ST_XSize, ST_YSize, ST_MSize

7.18.18 ST_MMin

ST_MMin — Returns the M minima of a geometry.

Synopsis

float **ST_MMin**(geometry geom);

Description

Returns the M minima of a geometry, or null if the geometry lacks M values.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_MMin('POINT M (1 2 3)');
```

Output

```
3
```

See Also

ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMax, ST_MSize

7.18.19 ST_MMax

ST_MMax — Returns the M maxima of a geometry.

Synopsis

float **ST_MMax**(geometry geom);

Description

Returns the M maxima of a geometry, or null if the geometry lacks M values.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT ST_MMax('POINT M (1 2 3)');
```

Output

```
3
```

See Also

ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MSize

7.18.20 ST_MSize

ST_MSize — Returns the M size of a geometry.

Synopsis

float **ST_MSize**(geometry geom);

Description

Returns the M size of a geometry, computed as `ST_MMax(geom) - ST_MMin(geom)`, or null if the geometry lacks M values.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Availability: 3.7.0

Examples

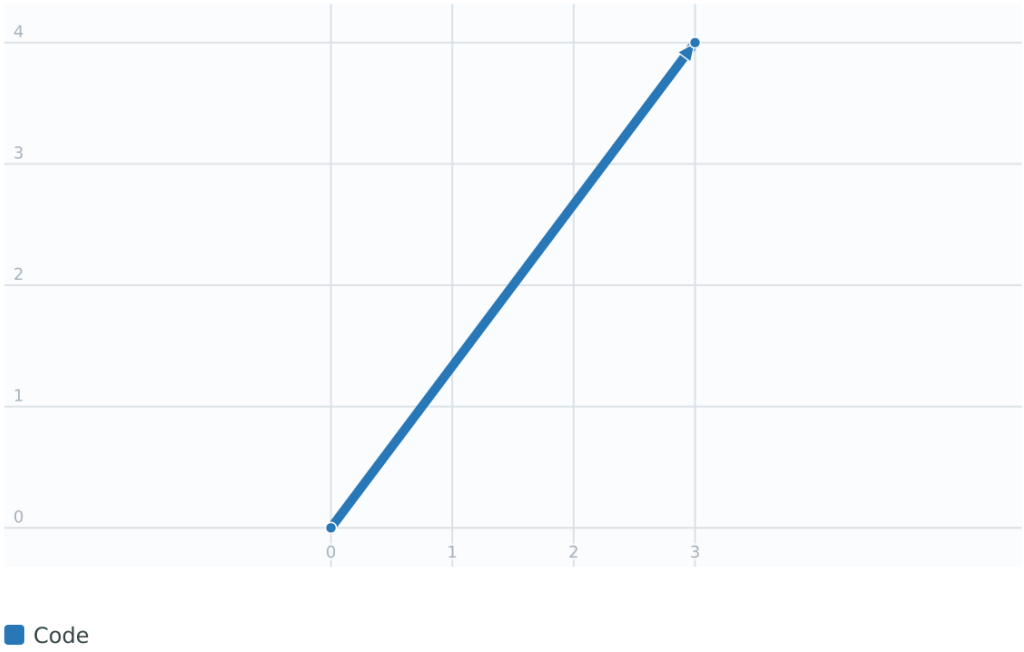
Code

```
SELECT ST_MSize('LINESTRING M (1 2 3,4 6 8)');
```

Output

```
5
```

Figure



See Also

ST_MMin, ST_MMax, ST_XSize, ST_YSize, ST_ZSize

7.19 Linear Referencing

7.19.1 ST_LineInterpolatePoint

ST_LineInterpolatePoint — Returns a point interpolated along a line at a fractional location.

Synopsis

geometry **ST_LineInterpolatePoint**(geometry a_linestring, float8 a_fraction);
geography **ST_LineInterpolatePoint**(geography a_linestring, float8 a_fraction, boolean use_spheroid = true);

Description

Returns a point interpolated along a line at a fractional location. First argument must be a LINESTRING. Second argument is a float between 0 and 1 representing the fraction of line length where the point is to be located. The Z and M values are interpolated if present.

See ST_LineLocatePoint for computing the line location nearest to a Point.



Note
This function computes points in 2D and then interpolates values for Z and M, while ST_3DLineInterpolatePoint computes points in 3D and only interpolates the M value.

Note
Since release 1.1.1 this function also interpolates M and Z values (when present), while prior releases set them to 0.0.

Availability: 0.8.2, Z and M supported added in 1.1.1

Changed: 2.1.0. Up to 2.0.x this was called ST_Line_Interpolate_Point.

This function supports 3d and will not drop the z-index.

Examples

This example returns the point 20 percent along a line.

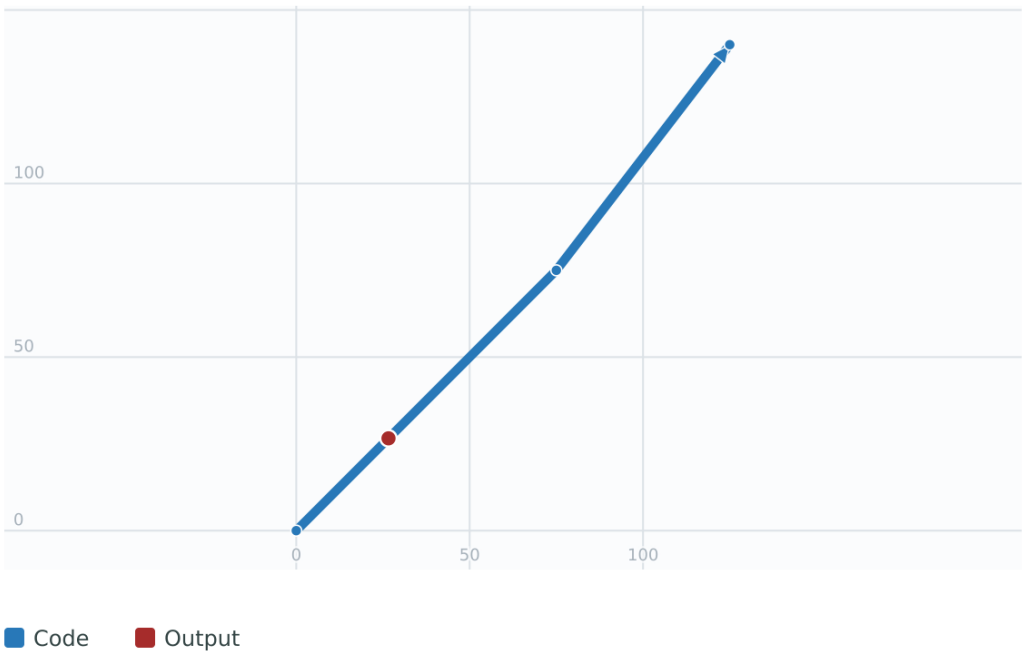
Code

```
SELECT ST_LineInterpolatePoint(
'LINESTRING(25 50,100 125,150 190)',
0.2);
```

Output

```
POINT(51.5974135047432 76.5974135047432)
```

Figure



The mid-point of a 3D line:

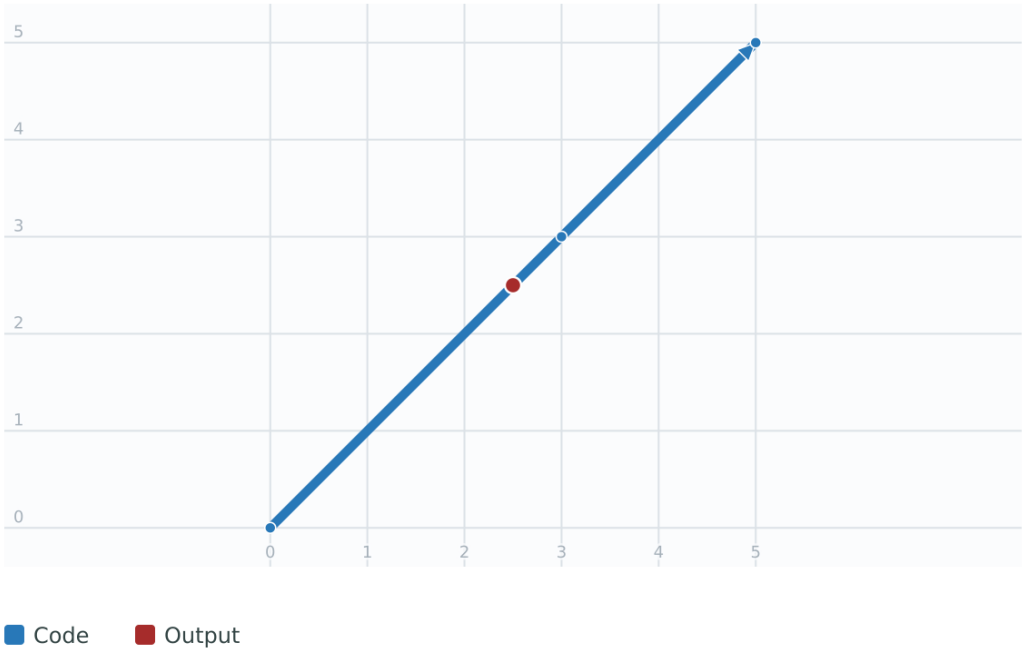
Code

```
SELECT ST_LineInterpolatePoint(
'LINESTRING(1 2 3,4 5 6,6 7 8)',
0.5);
```

Output

```
POINT(3.5 4.5 5.5)
```

Figure



The closest point on a line to a point:

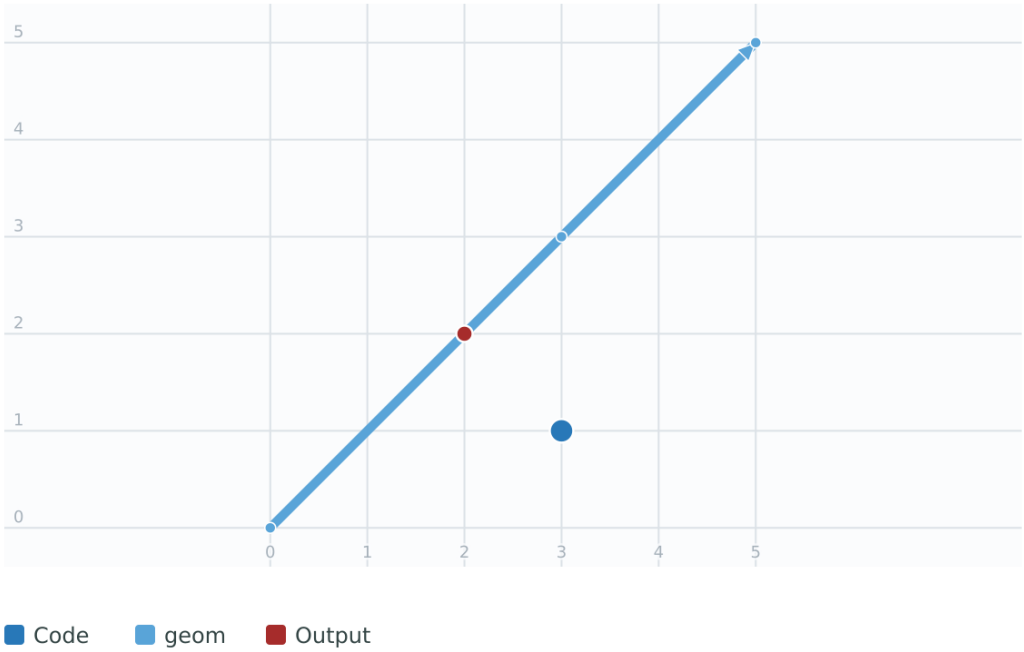
Code

```
SELECT ST_LineInterpolatePoint(
    line.geom,
    ST_LineLocatePoint(line.geom, 'POINT(4 3)'))
FROM (SELECT 'LINESTRING(1 2,4 5,6 7)'::geometry As geom) AS line;
```

Output

```
POINT(3 4)
```

Figure

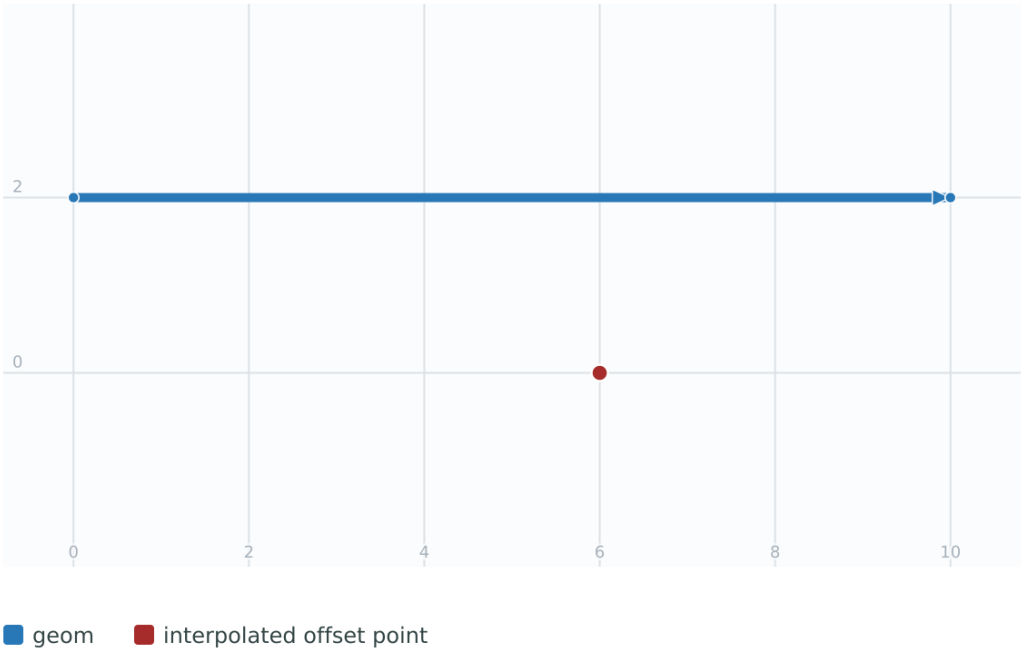


An interpolated point can be offset to the left or right of a line by using ST_OffsetCurve before interpolation. This is useful for locating an address point beside a street centerline.

Code

```
WITH street AS (
    SELECT 'LINESTRING(0 0, 10 0)'::geometry AS geom,
           100 AS from_num,
           200 AS to_num
)
SELECT
    ST_LineInterpolatePoint(
        ST_OffsetCurve(geom, -2),
        (160 - from_num)::float8 / (to_num - from_num)
    ) AS interpolated_offset_point
FROM street;
```

Figure



See Also

ST_LineInterpolatePoints, ST_3DLineInterpolatePoint, ST_LineLocatePoint, ST_OffsetCurve

7.19.2 ST_3DLineInterpolatePoint

ST_3DLineInterpolatePoint — Returns a point interpolated along a 3D line at a fractional location.

Synopsis

geometry **ST_3DLineInterpolatePoint**(geometry a_linestring, float8 a_fraction);

Description

Returns a point interpolated along a 3D line at a fractional location. First argument must be a LINESTRING. Second argument is a float between 0 and 1 representing the point location as a fraction of line length. The M value is interpolated if present.



Note
ST_LineInterpolatePoint computes points in 2D and then interpolates the values for Z and M, while this function computes points in 3D and only interpolates the M value.

Availability: 3.0.0


This function supports 3d and will not drop the z-index.

Examples

Return point 20% along 3D line

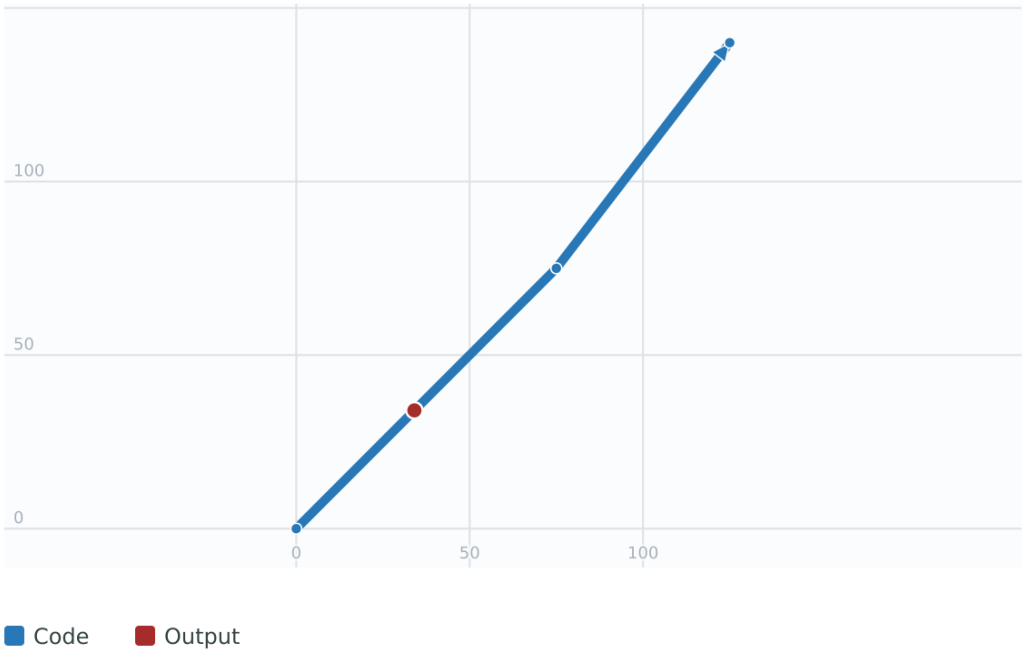
Code

```
SELECT ST_3DLineInterpolatePoint(
'LINESTRING(25 50 70,100 125 90,150 190 200)',
0.20);
```

Output

```
POINT Z (59.06758929108215 84.06758929108216 79.0846904776219)
```

Figure



See Also

ST_LineInterpolatePoint, ST_LineInterpolatePoints, ST_LineLocatePoint

7.19.3 ST_LineInterpolatePoints

ST_LineInterpolatePoints — Returns points interpolated along a line at a fractional interval.

Synopsis

geometry **ST_LineInterpolatePoints**(geometry a_linestring, float8 a_fraction, boolean repeat);
geography **ST_LineInterpolatePoints**(geography a_linestring, float8 a_fraction, boolean use_spheroid = true, boolean repeat = true);

Description

Returns one or more points interpolated along a line at a fractional interval. The first argument must be a LINESTRING. The second argument is a float8 between 0 and 1 representing the spacing between the points as a fraction of line length. If the third argument is false, at most one point will be constructed (which is equivalent to ST_LineInterpolatePoint.)

If the result has zero or one points, it is returned as a POINT. If it has two or more points, it is returned as a MULTIPOINT.

Availability: 2.5.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Examples

This example returns points every 20 percent along a two-dimensional line.

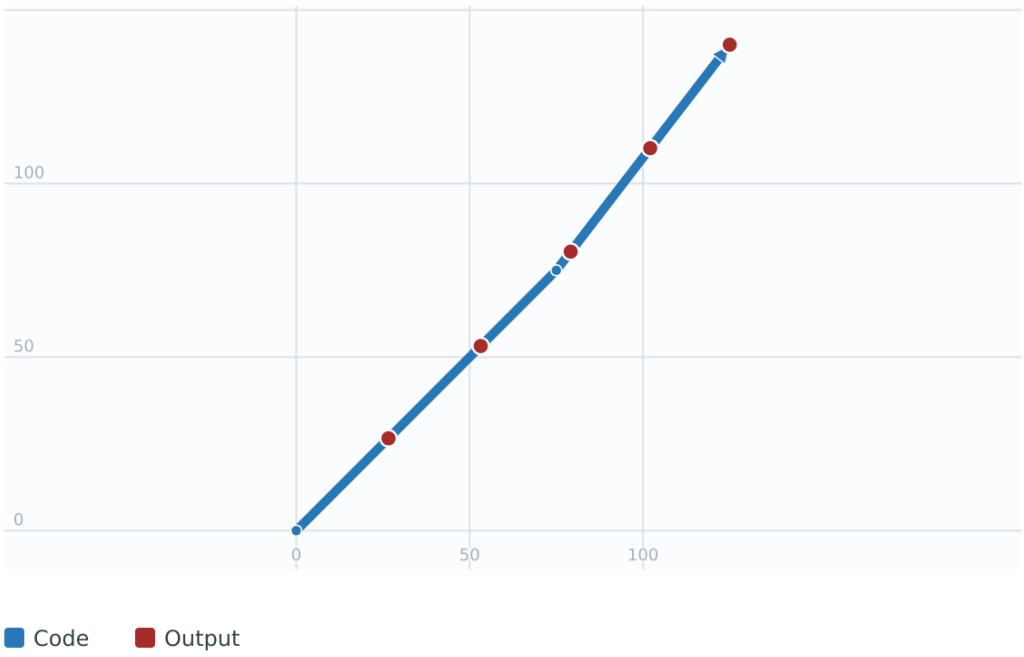
Code

```
SELECT ST_LineInterpolatePoints(
'LINESTRING(25 50,100 125,150 190)',
0.20);
```

Output

```
MULTIPOINT((51.5974135047432 76.5974135047432),(78.1948270094864 103.194827009486),(104.132163186446 130.37181214238) ↩
    ,(127.066081593223 160.18590607119),(150 190))
```

Figure



See Also

ST_LineInterpolatePoint, ST_LineLocatePoint

7.19.4 ST_LineLocatePoint

ST_LineLocatePoint — Returns the fractional location of the closest point on a line to a point.

Synopsis

float8 **ST_LineLocatePoint**(geometry a_linestring, geometry a_point);
float8 **ST_LineLocatePoint**(geography a_linestring, geography a_point, boolean use_spheroid = true);

Description

Returns a float between 0 and 1 representing the location of the closest point on a LineString to the given Point, as a fraction of 2d line length.

You can use the returned location to extract a Point (ST_LineInterpolatePoint) or a substring (ST_LineSubstring).

This is useful for approximating numbers of addresses

Availability: 1.1.0

Changed: 2.1.0. Up to 2.0.x this was called ST_Line_Locate_Point.

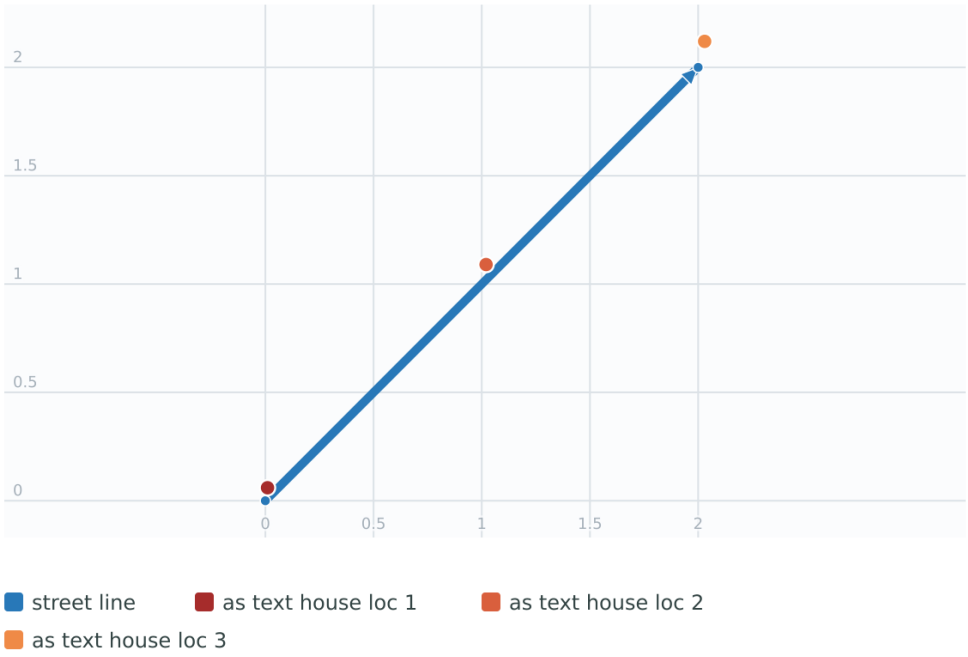
Examples

This example approximates the street number of a house by locating the closest point on a street line. The subquery generates sample house and street data, and `ST_DWithin` excludes houses that are too far from the street.

Code

```
SELECT house_loc As as_text_house_loc,
    start_house_num +
        CAST( (end_house_num - start_house_num)
            * ST_LineLocatePoint(street_line, house_loc) As integer) As house_num
FROM
(SELECT 'LINESTRING(1 2,3 4)'::geometry As street_line,
    ST_Point(x*1.01, y*1.03) As house_loc, 10 As start_house_num,
        20 As end_house_num
FROM generate_series(1, 3) x CROSS JOIN generate_series(2, 4) As y)
As foo
WHERE ST_DWithin(street_line, house_loc, 0.2);
```

Figure



This example finds the closest point on a line to a point or other geometry.

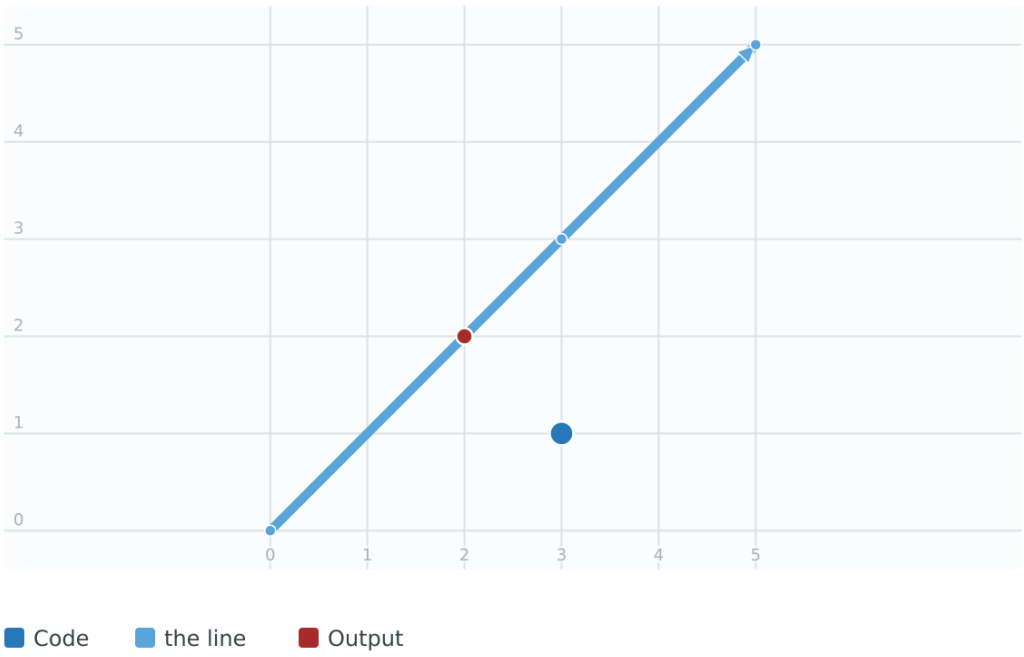
Code

```
SELECT ST_LineInterpolatePoint(
    foo.the_line,
    ST_LineLocatePoint(foo.the_line, 'POINT(4 3)'::geometry))
FROM (SELECT 'LINESTRING(1 2,4 5,6 7)'::geometry As the_line) As foo;
```

Output

```
POINT(3 4)
```

Figure

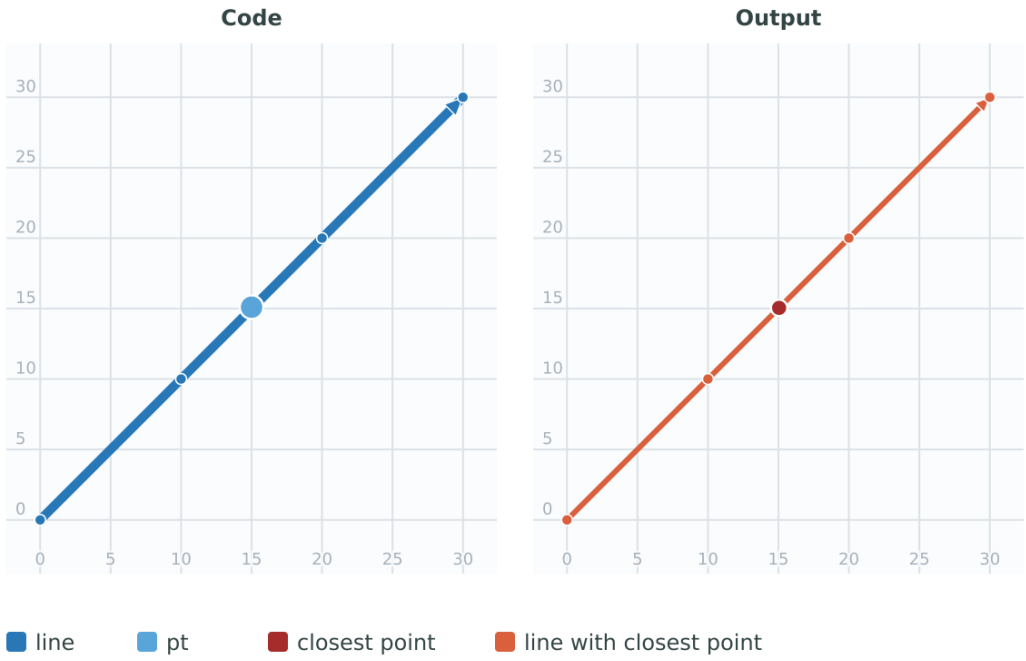


This example locates the closest segment and point on a LineString.

Code

```
WITH data AS (
  SELECT
    'LINESTRING (0 0, 10 10, 20 20, 30 30)'::geometry AS line,
    'POINT(15 15.1)'::geometry AS pt
),
segments AS (
  SELECT
    s.i,
    d.line,
    d.pt,
    ST_MakeLine(ST_PointN(d.line, s.i), ST_PointN(d.line, s.i + 1)) AS segment
  FROM data AS d
  CROSS JOIN LATERAL generate_series(1, ST_NumPoints(d.line) - 1) AS s(i)
)
SELECT
  i AS segment_number,
  round(ST_Distance(segment, pt)::numeric, 3) AS distance,
  ST_ClosestPoint(segment, pt) AS closest_point,
  ST_AddPoint(line, ST_ClosestPoint(segment, pt), i) AS line_with_closest_point
FROM segments
ORDER BY ST_Distance(segment, pt)
LIMIT 1;
```

Figure



See Also

ST_AddPoint, ST_ClosestPoint, ST_DWithin, ST_Length2D, ST_LineInterpolatePoint, ST_LineSubstring

7.19.5 ST_LineSubstring

ST_LineSubstring — Returns the part of a line between two fractional locations.

Synopsis

geometry **ST_LineSubstring**(geometry a_linestring, float8 startfraction, float8 endfraction);
geography **ST_LineSubstring**(geography a_linestring, float8 startfraction, float8 endfraction);

Description

Computes the line which is the section of the input line starting and ending at the given fractional locations. The first argument must be a LINESTRING. The second and third arguments are values in the range [0, 1] representing the start and end locations. For geometry inputs, the fractions are measured in 2D line length. The Z and M values are interpolated for added endpoints if present.

If `startfraction` and `endfraction` have the same value this is equivalent to ST_LineInterpolatePoint.



Note
This only works with LINESTRINGs. To use on contiguous MULTILINESTRINGs first join them with ST_LineMerge.



Note
Since release 1.1.1 this function interpolates M and Z values. Prior releases set Z and M to unspecified values.

Enhanced: 3.4.0 - Support for geography was introduced.

Changed: 2.1.0. Up to 2.0.x this was called ST_Line_Substring.

Availability: 1.1.0, Z and M supported added in 1.1.1

 This function supports 3d and will not drop the z-index.

Examples

Extract the middle third of a LineString, from 0.333 to 0.666.

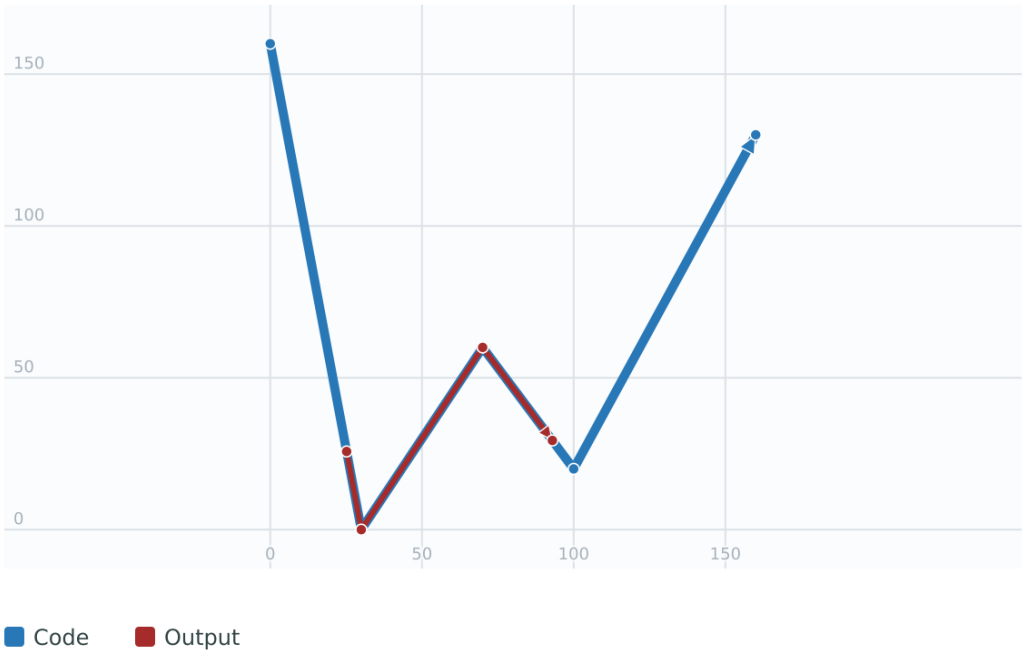
Code

```
SELECT ST_LineSubstring(
'LINESTRING (20 180,50 20,90 80,120 40,180 150)',
0.333,
0.666);
```

Output

```
LINESTRING (45.17311810399485 45.74337011202746, 50 20, 90 80, 112.97593050157862 49.36542599789519)
```

Figure



If start and end locations are the same, the result is a POINT.

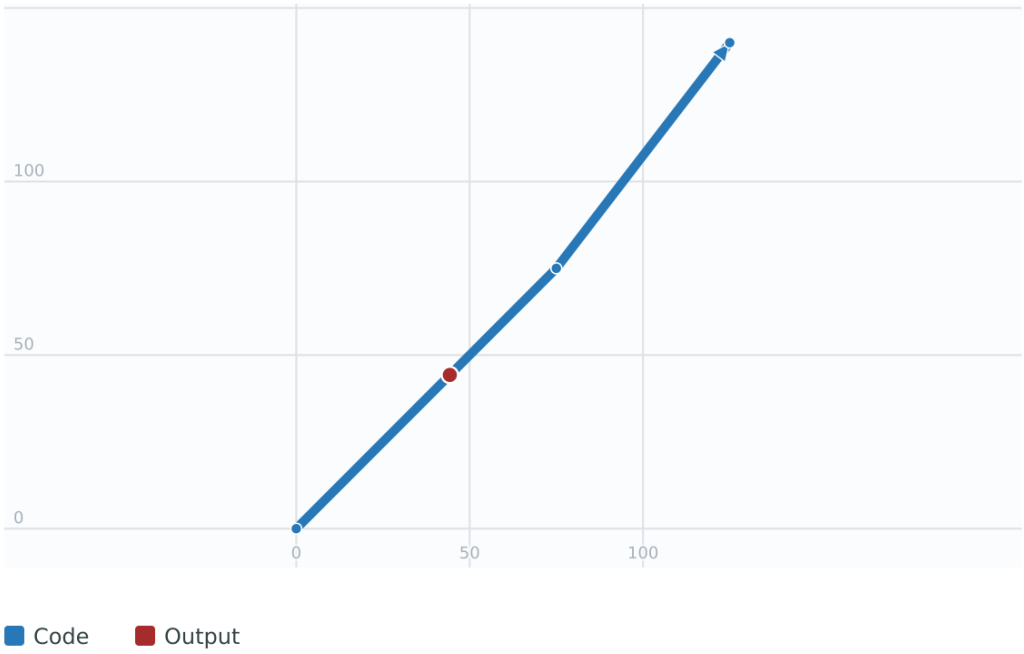
Code

```
SELECT ST_LineSubstring(
'LINESTRING(25 50,100 125,150 190)',
0.333,
0.333);
```

Output

```
POINT(69.28469348539744 94.28469348539744)
```

Figure



A query to cut a LineString into sections of length 100 or shorter. It uses `generate_series()` with a CROSS JOIN LATERAL to produce the equivalent of a FOR loop.

The final generated step is filtered out when the line length is an exact multiple of the requested section length.

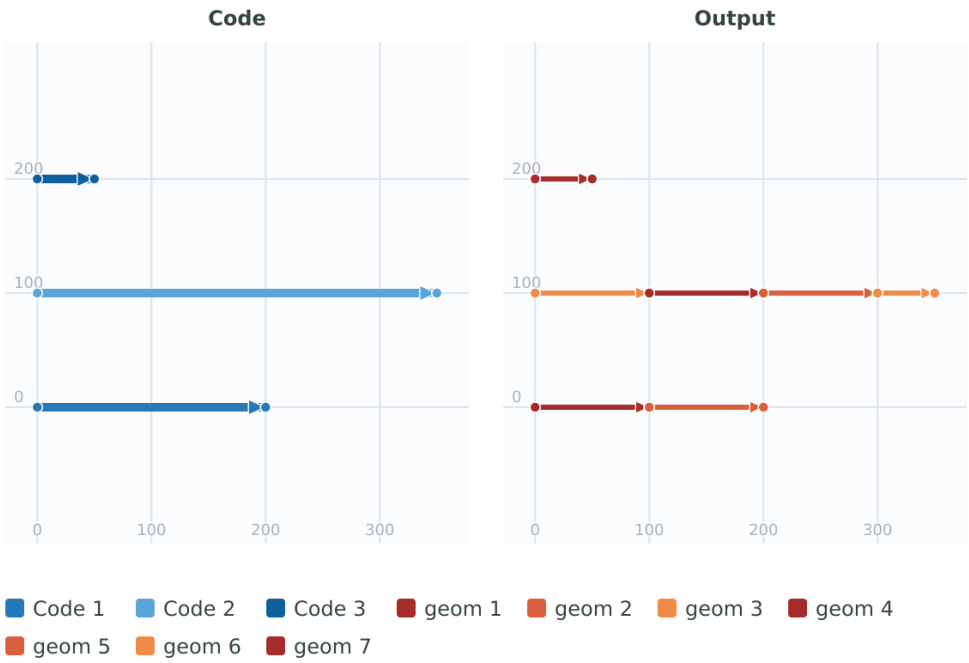
Code

```
WITH data(id, geom) AS (VALUES
        ( 'A', 'LINESTRING( 0 0, 200 0)'::geometry ),
        ( 'B', 'LINESTRING( 0 100, 350 100)'::geometry ),
        ( 'C', 'LINESTRING( 0 200, 50 200)'::geometry )
    )
SELECT id, i,
       ST_LineSubstring(geom, startfrac, LEAST( endfrac, 1 )) AS geom
FROM (
    SELECT id, geom, ST_Length(geom) len, 100 sublen FROM data
    ) AS d
CROSS JOIN LATERAL (
    SELECT i, (sublen * i) / len AS startfrac,
              (sublen * (i+1)) / len AS endfrac
    FROM generate_series(0, floor( len / sublen )::integer ) AS t(i)
    WHERE (sublen * i) / len <> 1.0
    ) AS d2;
```

Output

```
 id | i |            geom
----+---+----------------------------
 A  | 0 | LINESTRING(0 0,100 0)
 A  | 1 | LINESTRING(100 0,200 0)
 B  | 0 | LINESTRING(0 100,100 100)
 B  | 1 | LINESTRING(100 100,200 100)
 B  | 2 | LINESTRING(200 100,300 100)
 B  | 3 | LINESTRING(300 100,350 100)
 C  | 0 | LINESTRING(0 200,50 200)
```

Figure



Split a LineString by points by locating the points along the line, adding the 0 and 1 endpoints, and extracting substrings between adjacent locations.

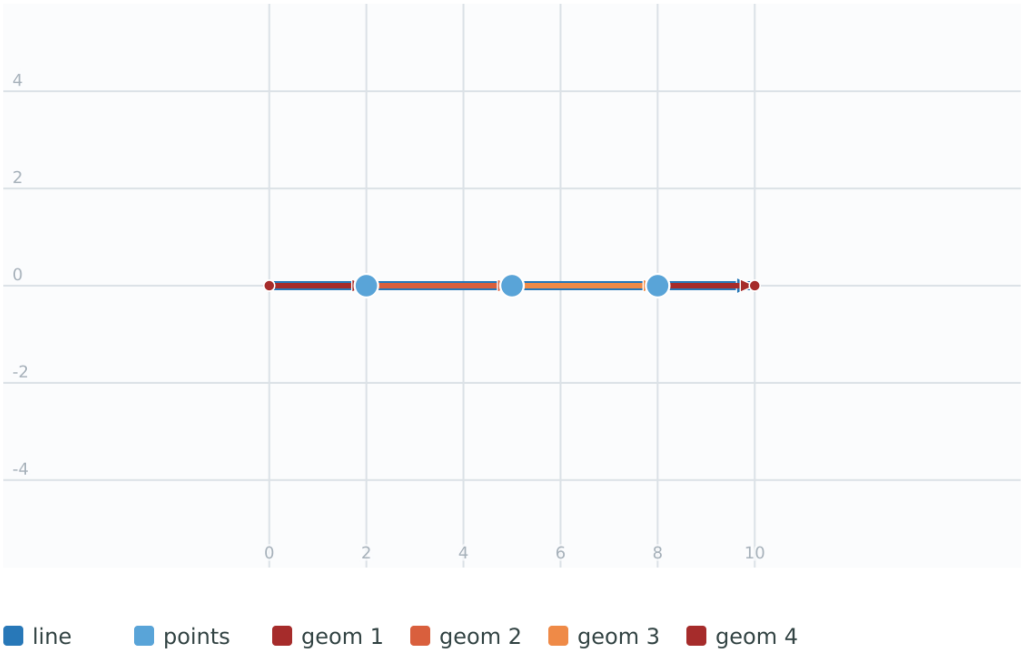
Code

```
WITH data AS (
  SELECT 'LINESTRING(0 0, 10 0)'::geometry AS line,
         'MULTIPOINT((2 0),(5 0),(8 0))'::geometry AS points
),
fractions AS (
  SELECT 0.0 AS fraction
  UNION ALL
  SELECT 1.0
  UNION ALL
  SELECT ST_LineLocatePoint(line, (dp).geom)
  FROM data
  CROSS JOIN LATERAL ST_Dump(points) AS dp
),
ordered AS (
  SELECT DISTINCT fraction
  FROM fractions
),
segments AS (
  SELECT fraction AS start_fraction,
         lead(fraction) OVER (ORDER BY fraction) AS end_fraction
  FROM ordered
)
SELECT row_number() OVER (ORDER BY start_fraction) AS segment_id,
       ST_LineSubstring(line, start_fraction, end_fraction) AS geom
FROM data
CROSS JOIN segments
WHERE end_fraction IS NOT NULL
  AND start_fraction <> end_fraction
ORDER BY start_fraction;
```

Output

```
 segment_id |         geom
------------+----------------------
          1 | LINESTRING(0 0,2 0)
          2 | LINESTRING(2 0,5 0)
          3 | LINESTRING(5 0,8 0)
          4 | LINESTRING(8 0,10 0)
```

Figure



Geography implementation measures along a spheroid, geometry along a line

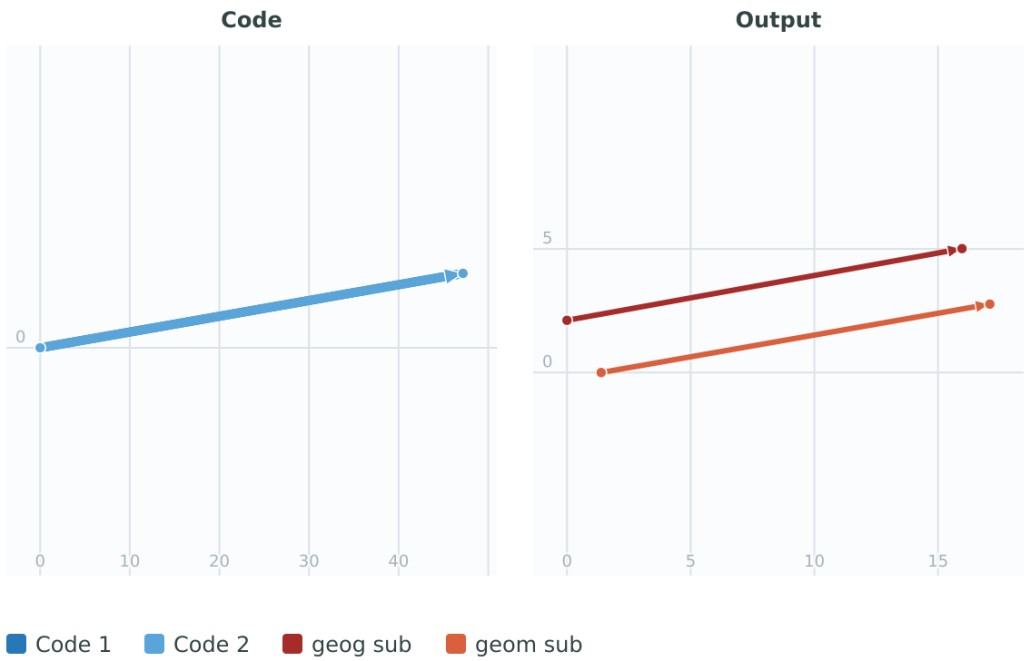
Code

```
SELECT ST_LineSubstring(
'LINESTRING(-118.2436 34.0522,-71.0570 42.3611)'::geography,
0.333,
0.666) AS geog_sub,
ST_LineSubstring(
    'LINESTRING(-118.2436 34.0522,-71.0570 42.3611)'::geometry,
    0.333,
    0.666) AS geom_sub;
```

Output

```
-[ RECORD 1 ]----------
geog_sub | LINESTRING(-103.911641 38.931128,-87.941787 41.831072)
geom_sub | LINESTRING(-102.530462 36.819064,-86.817324 39.585927)
```

Figure



For geometry inputs, the fractional locations are based on the 2D length of the line, even when the input has Z values. The resulting Z values are interpolated along the selected 2D location.

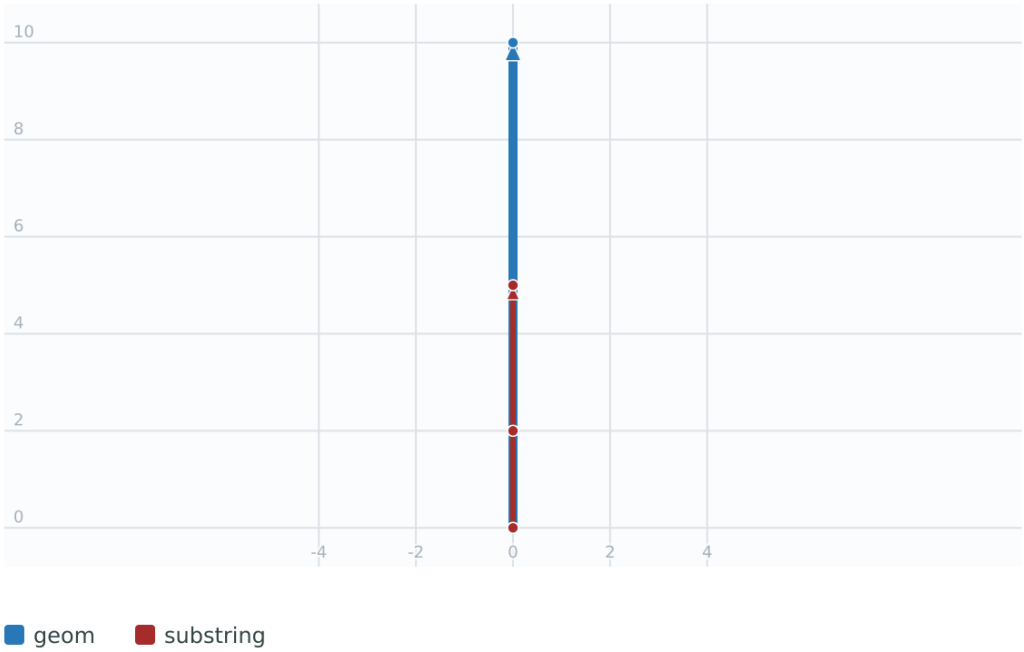
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (0 0 0,0 2 5,0 10 10)'::geometry AS geom
)
SELECT ST_Length(geom) AS length_2d,
       ST_3DLength(geom) AS length_3d,
       ST_LineSubstring(geom, 0, 0.5) AS substring
FROM data;
```

Output

```
 length_2d |    length_3d     |              substring
-----------+------------------+-------------------------------------
        10 | 14.819145939191106 | LINESTRING Z (0 0 0,0 2 5,0 5 6.875)
```

Figure



See Also

ST_Length, ST_LineExtend, ST_LineInterpolatePoint, ST_LineLocatePoint, ST_LineMerge, ST_Split

7.19.6 ST_LocateAlong

ST_LocateAlong — Returns the point(s) on a geometry that match a measure value.

Synopsis

geometry **ST_LocateAlong**(geometry geom_with_measure, float8 measure, float8 offset = 0);

Description

Returns the location(s) along a measured geometry that have the given measure values. The result is a Point or MultiPoint. Polygonal inputs are not supported.

If `offset` is provided, the result is offset to the left or right of the input line by the specified distance. A positive offset will be to the left, and a negative one to the right.



Note
Use this function only for linear geometries with an M component

The semantic is specified by the *ISO/IEC 13249-3 SQL/MM Spatial* standard.

Availability: 1.1.0 by old name ST_Locate_Along_Measure.

Changed: 2.0.0 in prior versions this used to be called ST_Locate_Along_Measure.

This function supports M coordinates.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.13

Examples

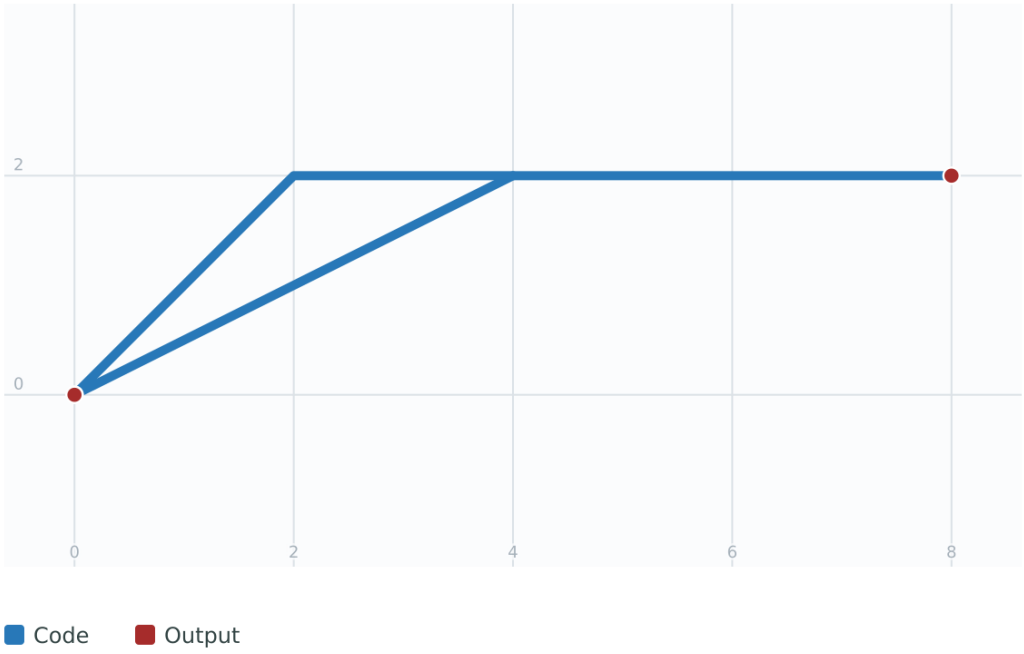
Code

```
SELECT ST_LocateAlong(
'MULTILINESTRINGM((1 2 3, 3 4 2, 9 4 3),(1 2 3, 5 4 5))'::geometry,
3);
```

Output

```
MULTIPOINT M ((1 2 3),(9 4 3),(1 2 3))
```

Figure



See Also

ST_LocateBetween, ST_LocateBetweenElevations, ST_InterpolatePoint

7.19.7 ST_LocateBetween

ST_LocateBetween — Returns the portions of a geometry that match a measure range.

Synopsis

geometry **ST_LocateBetween**(geometry geom, float8 measure_start, float8 measure_end, float8 offset = 0);

Description

Return a geometry (collection) with the portions of the input measured geometry that match the specified measure range (inclusively).

If the `offset` is provided, the result is offset to the left or right of the input line by the specified distance. A positive offset will be to the left, and a negative one to the right.

Clipping a non-convex POLYGON may produce invalid geometry.

The semantic is specified by the *ISO/IEC 13249-3 SQL/MM Spatial* standard.

Availability: 1.1.0 by old name ST_Locate_Between_Measures.

Changed: 2.0.0 - in prior versions this used to be called ST_Locate_Between_Measures.

Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE.

This function supports M coordinates.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

Examples

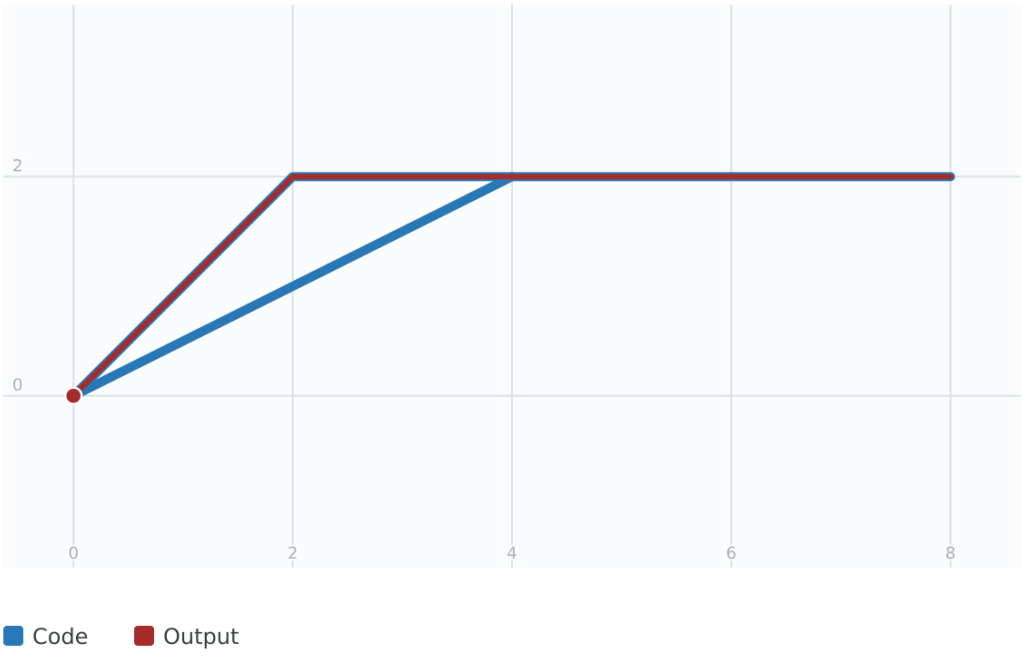
Code

```
SELECT ST_LocateBetween(
'MULTILINESTRING M ((1 2 3,3 4 2,9 4 3),(1 2 3,5 4 5))'::geometry,
1.5,
3);
```

Output

```
GEOMETRYCOLLECTION M (LINESTRING M (1 2 3,3 4 2,9 4 3),POINT M (1 2 3))
```

Figure



Extract the section between measures 2 and 8, offset to the left. Because the clipped line and the offset result share the same coordinate space, this example is intended for a single overlay panel rather than separate side-by-side figures.

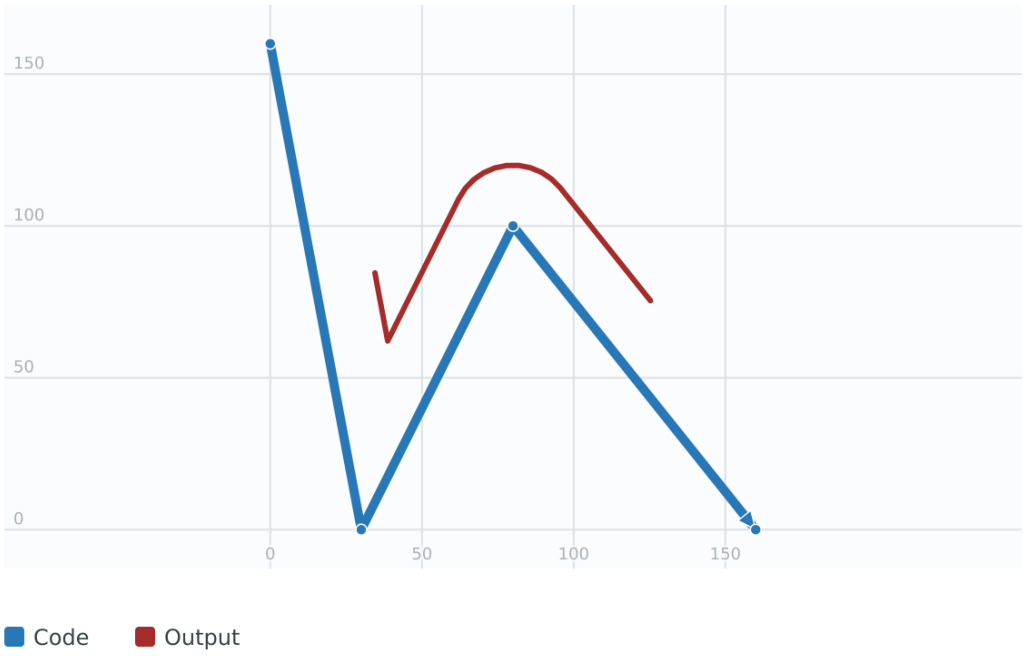
Code

```
SELECT ST_LocateBetween(
ST_AddMeasure(
    'LINESTRING (20 180,50 20,100 120,180 20)',
    0,
    10),
2,
8,
20);
```

Output

```
MULTILINESTRING((54.49835 104.53427,58.700561 82.122481,82.111456 128.944272,84.248937 132.324934,87.01637 ↩
    135.21267,90.303073 137.491987,93.977598 139.071724,97.892984 139.888702,101.892639 139.910247,105.816599 ↩
    139.135495,109.507928 137.595434,112.818995 135.351657,115.617376 132.493901,145.310173 95.377905))
```

Figure



See Also

ST_LocateAlong, ST_LocateBetweenElevations

7.19.8 ST_LocateBetweenElevations

ST_LocateBetweenElevations — Returns the portions of a geometry that lie in an elevation (Z) range.

Synopsis

geometry **ST_LocateBetweenElevations**(geometry geom, float8 elevation_start, float8 elevation_end);

Description

Returns a geometry (collection) with the portions of a geometry that lie in an elevation (Z) range.

Clipping a non-convex POLYGON may produce invalid geometry.

Availability: 1.4.0

Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE.

This function supports 3d and will not drop the z-index.

Examples

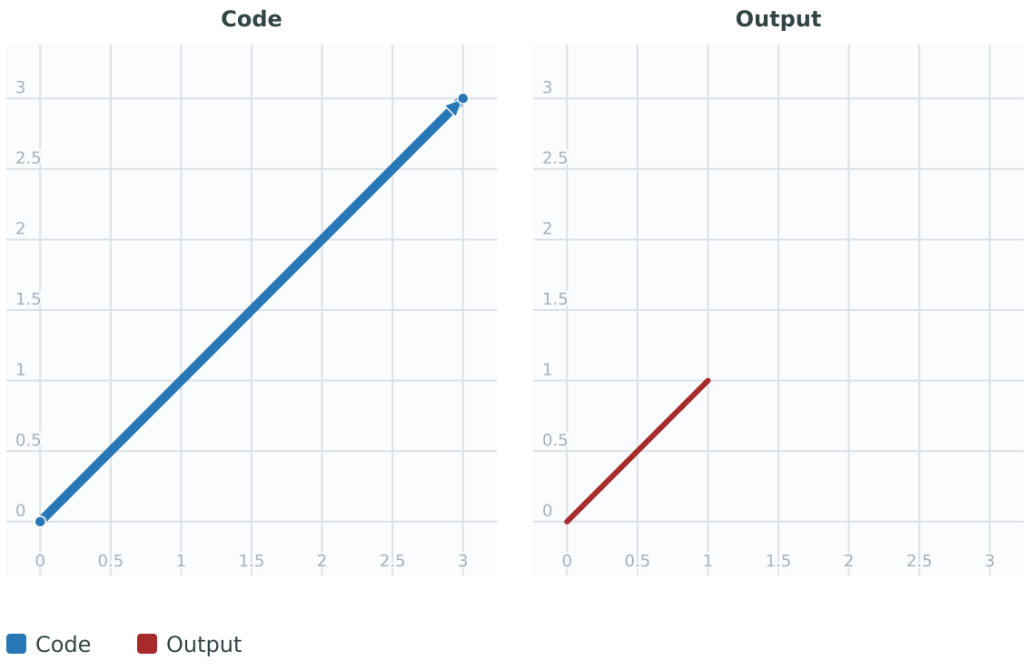
Code

```
SELECT ST_LocateBetweenElevations(
'LINESTRING(1 2 3,4 5 6)'::geometry,
2,
4);
```

Output

```
MULTILINESTRING Z ((1 2 3,2 3 4))
```

Figure



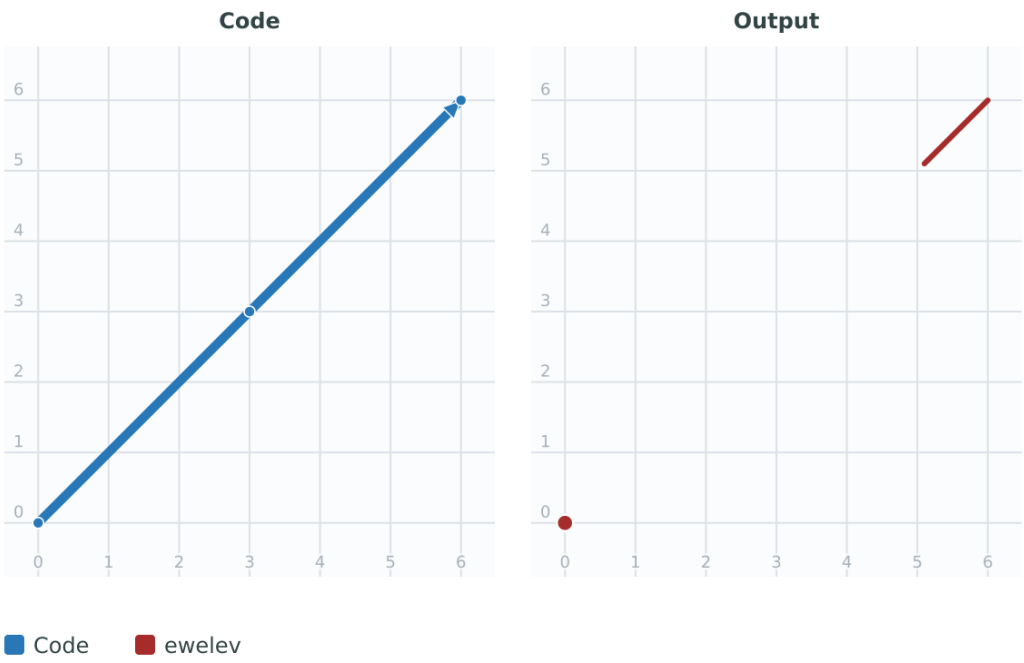
Code

```
SELECT ST_LocateBetweenElevations(
'LINESTRING(1 2 6,4 5 -1,7 8 9)',
6,
9) As ewelev;
```

Output

```
GEOMETRYCOLLECTION Z (POINT Z (1 2 6),LINESTRING Z (6.1 7.1 6,7 8 9))
```

Figure



See Also

ST_Dump, ST_LocateBetween

7.19.9 ST_InterpolatePoint

ST_InterpolatePoint — Returns the interpolated measure of a geometry closest to a point.

Synopsis

float8 **ST_InterpolatePoint**(geometry linear_geom_with_measure, geometry point);

Description

Returns an interpolated measure value of a linear measured geometry at the location closest to the given point.



Note
Use this function only for linear geometries with an M component

Availability: 2.0.0

 This function supports 3d and will not drop the z-index.

Examples

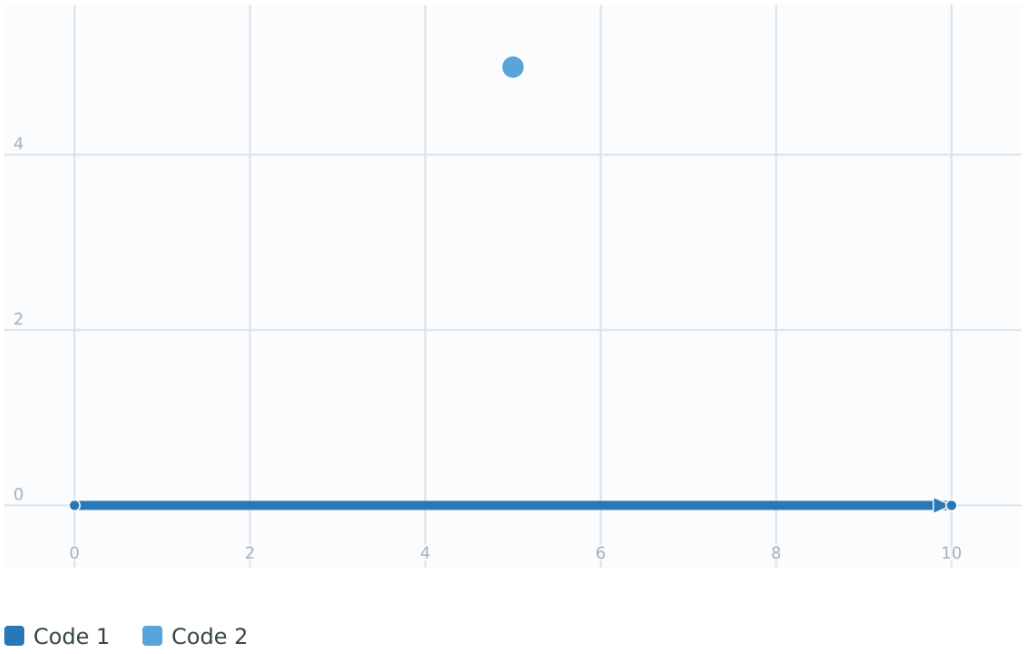
Code

```
SELECT ST_InterpolatePoint(
'LINESTRING M (0 0 0,10 0 20)',
'POINT(5 5)');
```

Output

```
10
```

Figure



See Also

ST_AddMeasure, ST_LocateAlong, ST_LocateBetween

7.19.10 ST_3DInterpolatePoint

ST_3DInterpolatePoint — Returns the interpolated measure of a geometry closest to a point in 3D.

Synopsis

float8 **ST_3DInterpolatePoint**(geometry linear_geom_with_measure, geometry point);

Description

Returns the interpolated measure value of a linear ZM geometry at the location closest to the given point, using 3D (XYZ) distance for the projection. Use this function when the geometry has significant Z variation, such as flight trajectories, where ST_InterpolatePoint would give incorrect results by ignoring the Z dimension.



Note
The line must have both Z and M dimensions. The point should have a Z dimension.

Availability: 3.7.0

 This function supports 3d and will not drop the z-index.

Examples

This example interpolates on a line rising diagonally in Z, using a point at the 3D midpoint.

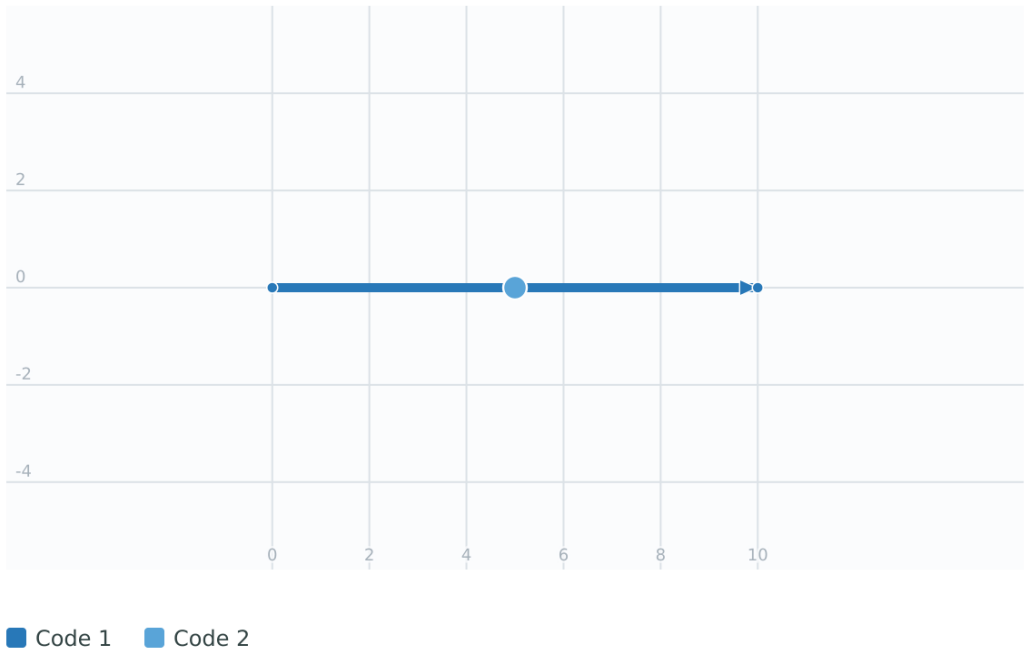
Code

```
SELECT ST_3DInterpolatePoint(
'LINESTRING ZM (0 0 0 0,10 0 10 20)',
'POINT Z (5 0 5)');
```

Output

```
10
```

Figure



This example uses a vertical line with zero XY extent, where 3D projection works but 2D projection does not.

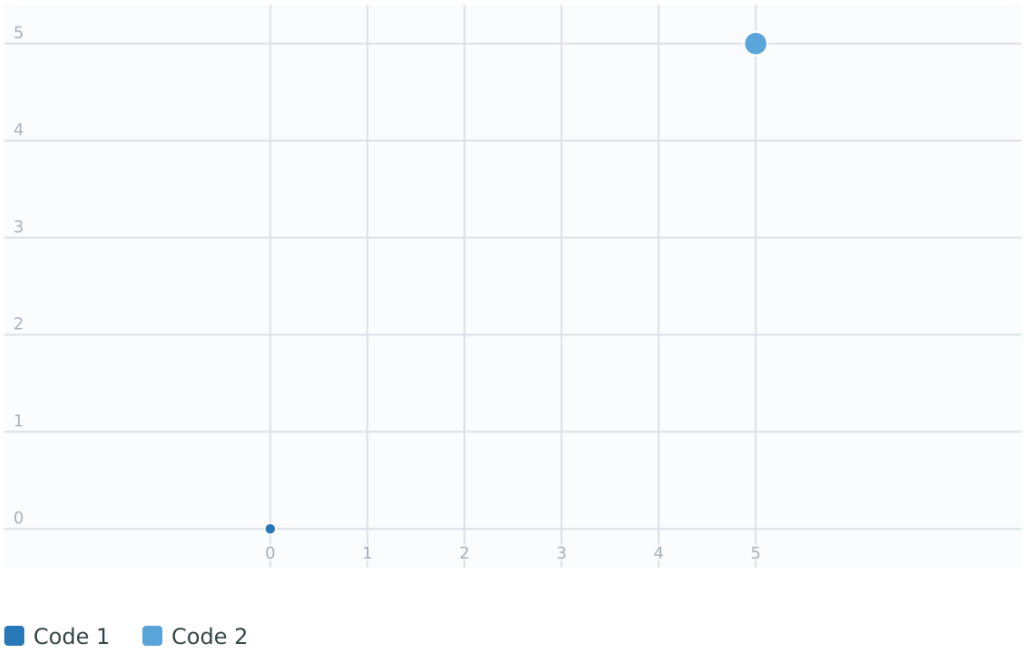
Code

```
SELECT ST_3DInterpolatePoint(
'LINESTRING ZM (0 0 0 0,0 0 10 100)',
'POINT Z (5 5 5)');
```

Output

```
50
```

Figure



See Also

ST_InterpolatePoint, ST_AddMeasure, ST_LocateAlong, ST_LocateBetween

7.19.11 ST_AddMeasure

ST_AddMeasure — Interpolates measures along a linear geometry.

Synopsis

geometry **ST_AddMeasure**(geometry geom_mline, float8 measure_start, float8 measure_end);

Description

Return a derived geometry with measure values linearly interpolated between the start and end points. If the geometry has no measure dimension, one is added. If the geometry has a measure dimension, it is over-written with new values. Only LINESTRINGS and MULTILINESTRINGS are supported.

Availability: 1.5.0

 This function supports 3d and will not drop the z-index.

Examples

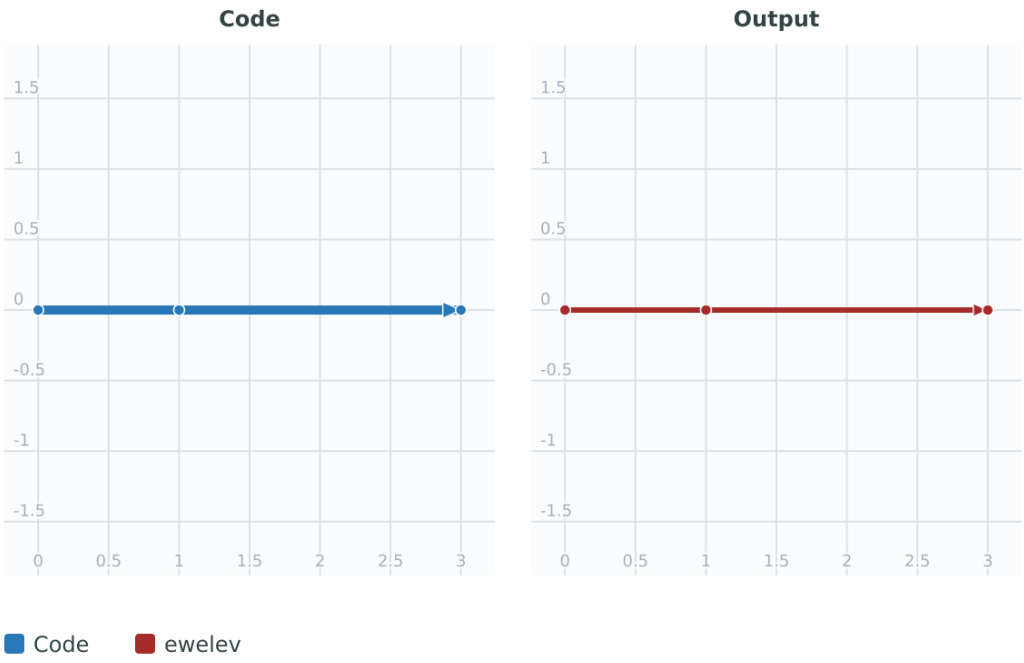
Code

```
SELECT ST_AddMeasure(
'LINESTRING(1 0,2 0,4 0)',
1,
4) As ewelev;
```

Output

```
LINESTRING M (1 0 1,2 0 2,4 0 4)
```

Figure



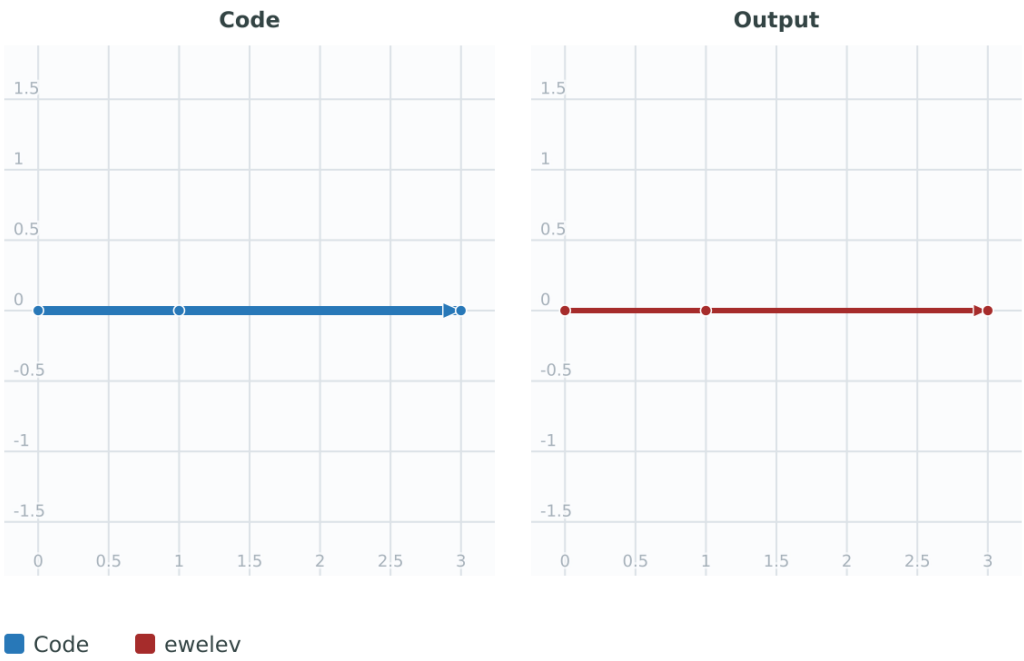
Code

```
SELECT ST_AddMeasure(
'LINESTRING(1 0 4,2 0 4,4 0 4)',
10,
40) As ewelev;
```

Output

```
LINESTRING ZM (1 0 4 10,2 0 4 20,4 0 4 40)
```

Figure



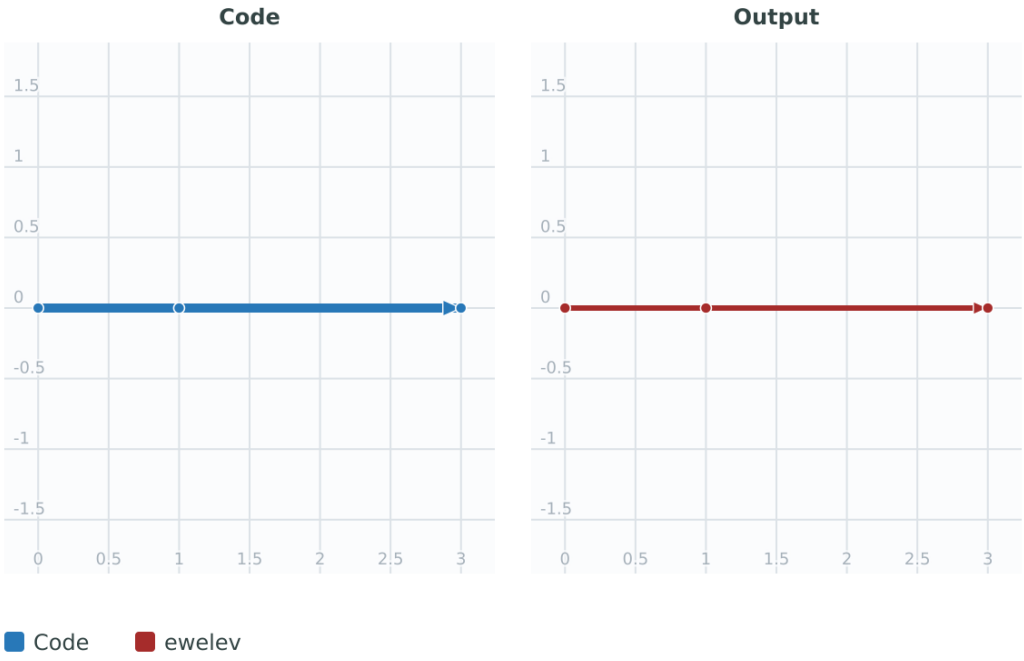
Code

```
SELECT ST_AddMeasure(
'LINESTRINGM(1 0 4, 2 0 4, 4 0 4)',
10,
40) As ewelev;
```

Output

```
LINESTRING M (1 0 10,2 0 20,4 0 40)
```

Figure



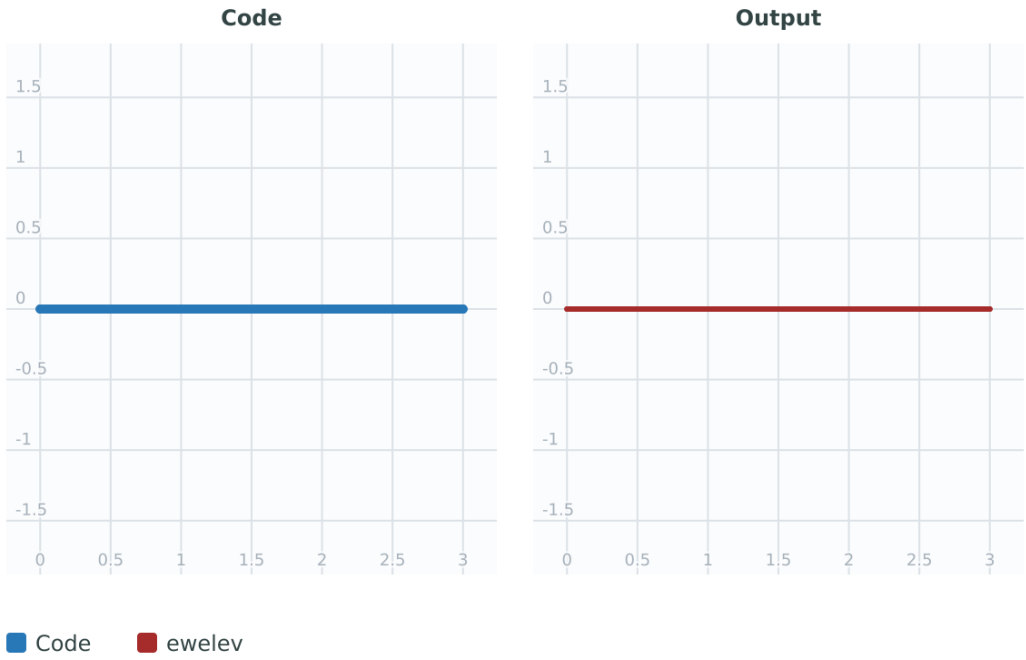
Code

```
SELECT ST_AddMeasure(
'MULTILINESTRINGM((1 0 4, 2 0 4, 4 0 4),(1 0 4, 2 0 4, 4 0 4))',
10,
70) As ewelev;
```

Output

```
MULTILINESTRING M ((1 0 10,2 0 20,4 0 40),(1 0 40,2 0 50,4 0 70))
```

Figure



7.20 Trajectory Functions

7.20.1 ST_IsValidTrajectory

ST_IsValidTrajectory — Tests if the geometry is a valid trajectory.

Synopsis

boolean **ST_IsValidTrajectory**(geometry line);

Description

Tests if a geometry encodes a valid trajectory. A valid trajectory is represented as a `LINESTRING` with measures (M values). The measure values must increase from each vertex to the next.

Valid trajectories are expected as input to spatio-temporal functions like ST_ClosestPointOfApproach

Availability: 2.2.0

 This function supports 3d and will not drop the z-index.

Examples

This example constructs a valid trajectory.

Code

```
SELECT ST_IsValidTrajectory(ST_MakeLine(ST_MakePointM(0, 0, 1),
ST_MakePointM(0, 1, 2)));
```

Output

```
t
```

This example constructs an invalid trajectory.

Code

```
SELECT ST_IsValidTrajectory(ST_MakeLine(ST_MakePointM(0, 0, 1), ST_MakePointM(0, 1, 0)));
```

Output

```
NOTICE:  Measure of vertex 1 (0) not bigger than measure of vertex 0 (1)
f
```

See Also

ST_ClosestPointOfApproach

7.20.2 ST_ClosestPointOfApproach

ST_ClosestPointOfApproach — Returns a measure at the closest point of approach of two trajectories.

Synopsis

float8 **ST_ClosestPointOfApproach**(geometry track1, geometry track2);

Description

Returns the smallest measure at which points interpolated along the given trajectories are the least distance apart.

Inputs must be valid trajectories as checked by ST_IsValidTrajectory. Null is returned if the trajectories do not overlap in their M ranges.

To obtain the actual points at the computed measure use ST_LocateAlong .

Availability: 2.2.0

 This function supports 3d and will not drop the z-index.

Examples

Return the time in which two objects moving between 10:00 and 11:00 are closest to each other and their distance at that point.

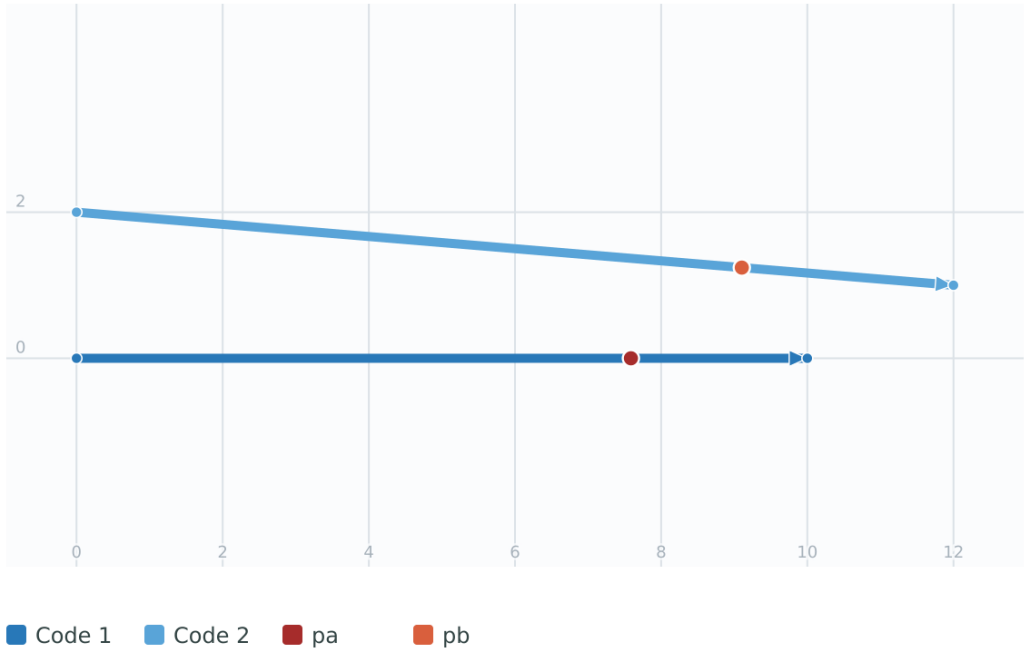
Code

```
WITH inp AS ( SELECT
  ST_AddMeasure('LINESTRING Z (0 0 0,10 0 5)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) a,
  ST_AddMeasure('LINESTRING Z (0 2 10,12 1 2)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) b
), cpa AS (
  SELECT ST_ClosestPointOfApproach(a, b) m FROM inp
), points AS (
  SELECT ST_GeometryN(ST_LocateAlong(a, m), 1) pa,
         ST_GeometryN(ST_LocateAlong(b, m), 1) pb
  FROM inp, cpa
)
SELECT to_timestamp(m) AT TIME ZONE 'UTC' t,
       ST_3DDistance(pa, pb) distance,
       pa AS pa, pb AS pb
FROM points, cpa;
```

Output

```
-[ RECORD 1 ]----------------------------
t        | 2015-05-26 10:45:31.034483
distance | 1.9652147377620688
pa       | POINT ZM (7.59 0 3.79 1432637131.03)
pb       | POINT ZM (9.1 1.24 3.93 1432637131.03)
```

Figure



See Also

ST_IsValidTrajectory, ST_DistanceCPA, ST_LocateAlong, ST_AddMeasure

7.20.3 ST_DistanceCPA

ST_DistanceCPA — Returns the distance between the closest point of approach of two trajectories.

Synopsis

float8 **ST_DistanceCPA**(geometry track1, geometry track2);

Description

Returns the distance (in 2D) between two trajectories at their closest point of approach.

Inputs must be valid trajectories as checked by ST_IsValidTrajectory. Null is returned if the trajectories do not overlap in their M ranges.

Availability: 2.2.0

 This function supports 3d and will not drop the z-index.

Examples

Return the minimum distance of two objects moving between 10:00 and 11:00.

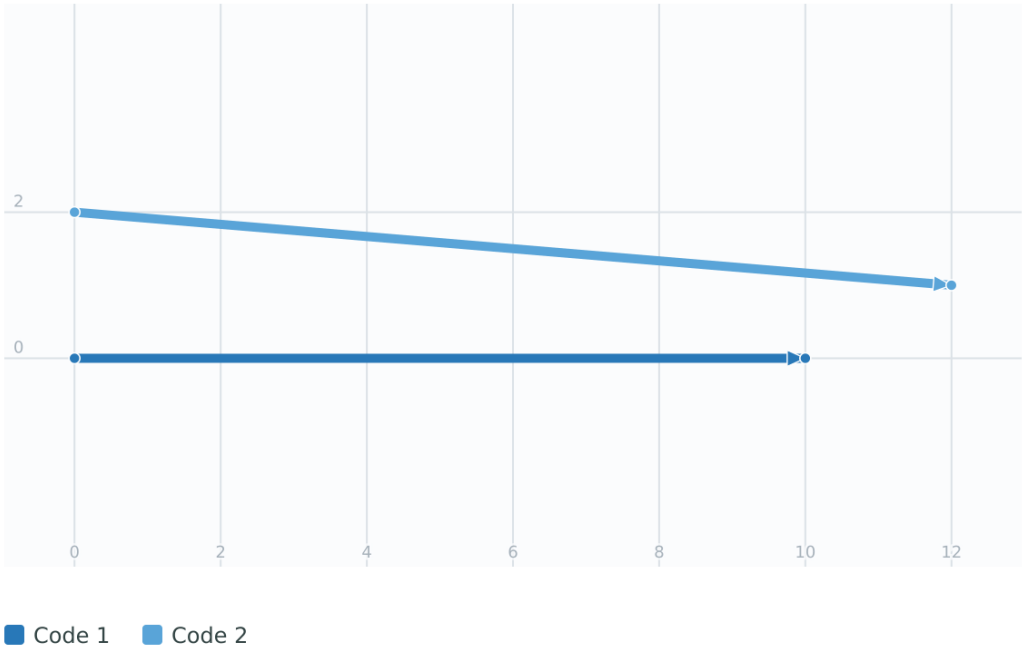
Code

```
WITH inp AS ( SELECT
  ST_AddMeasure('LINESTRING Z (0 0 0,10 0 5)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) a,
  ST_AddMeasure('LINESTRING Z (0 2 10,12 1 2)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) b
)
SELECT ST_DistanceCPA(a, b) distance FROM inp;
```

Output

```
1.9652147377620688
```

Figure



See Also

ST_IsValidTrajectory, ST_ClosestPointOfApproach, ST_AddMeasure, |=|

7.20.4 ST_CPAWithin

ST_CPAWithin — Tests if the closest point of approach of two trajectories is within the specified distance.

Synopsis

boolean **ST_CPAWithin**(geometry track1, geometry track2, float8 dist);

Description

Tests whether two moving objects have ever been closer than the specified distance.

Inputs must be valid trajectories as checked by ST_IsValidTrajectory. False is returned if the trajectories do not overlap in their M ranges.

Availability: 2.2.0

This function supports 3d and will not drop the z-index.

Examples

Code

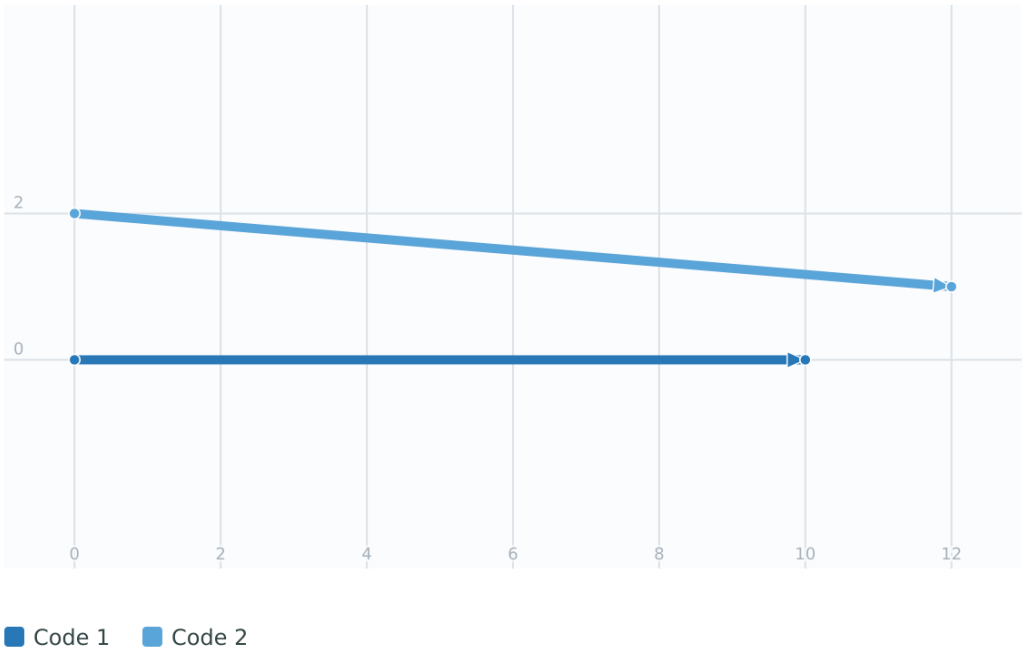
```
WITH inp AS ( SELECT
  ST_AddMeasure('LINESTRING Z (0 0 0,10 0 5)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
```

```
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) a,
  ST_AddMeasure('LINESTRING Z (0 2 10,12 1 2)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) b
)
SELECT ST_CPAWithin(a, b, 2), ST_DistanceCPA(a, b) distance FROM inp;
```

Output

```
 st_cpawithin |     distance
--------------+------------------
 t            | 1.9652147377620688
```

Figure



See Also

ST_IsValidTrajectory, ST_ClosestPointOfApproach, ST_DistanceCPA, |=|

7.21 Version Functions

7.21.1 PostGIS_Extensions_Upgrade

PostGIS_Extensions_Upgrade — Packages and upgrades PostGIS extensions (e.g. postgis_raster, postgis_topology, postgis_sfcgal) to given or latest version.

Synopsis

text **PostGIS_Extensions_Upgrade**(text target_version=null);

Description

Packages and upgrades PostGIS extensions to given or latest version. Only extensions you have installed in the database will be packaged and upgraded if needed. Reports full PostGIS version and build configuration infos after. This is short-hand for doing multiple CREATE EXTENSION .. FROM unpackaged and ALTER EXTENSION .. UPDATE for each PostGIS extension. Currently only tries to upgrade extensions postgis, postgis_raster, postgis_sfcgal, and postgis_topology.

Availability: 2.5.0

Changed: 3.4.0 to add target_version argument.

Changed: 3.3.0 support for upgrades from any PostGIS version. Does not work on all systems.

Changed: 3.0.0 to repackage loose extensions and support postgis_raster.

Examples

Code

```
SELECT PostGIS_Extensions_Upgrade();
```

Output

```
NOTICE:  Packaging extension postgis
NOTICE:  Packaging extension postgis_raster
NOTICE:  Packaging extension postgis_sfcgal
NOTICE:  Extension postgis_topology is not available or not packagable for some reason
```

See Also

Section 3.4, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.2 PostGIS_Full_Version

PostGIS_Full_Version — Reports full PostGIS version and build configuration infos.

Synopsis

text **PostGIS_Full_Version**();

Description

Reports full PostGIS version and build configuration infos. Also informs about synchronization between libraries and scripts suggesting upgrades as needed.

Enhanced: 3.4.0 now includes extra PROJ configurations NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location

Examples

Code

```
SELECT PostGIS_Full_Version();
```

Output

```
POSTGIS="3.7.0dev 3.7.0alpha1-477-ga9004e778" [EXTENSION] PGSQL="180" GEOS="3.15.0beta1" SFCGAL="2.3.0" PROJ="9.8.1 ↩
    NETWORK_ENABLED=OFF URL_ENDPOINT=https://cdn.proj.org DATABASE_PATH=/usr/share/proj/proj.db" GDAL="GDAL 3.13.1, ↩
    released 2026/06/01" LIBXML="2.15.3"
```

See Also

Section 3.4, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Wagyu_Ve
PostGIS_Version

7.21.3 PostGIS_GEOS_Version

PostGIS_GEOS_Version — Returns the version number of the GEOS library.

Synopsis

text **PostGIS_GEOS_Version**();

Description

Returns the version number of the GEOS library, or `NULL` if GEOS support is not enabled.

Examples

Code

```
SELECT PostGIS_GEOS_Version();
```

Output

```
3.12.0dev-CAPI-1.18.0
```

See Also

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.4 PostGIS_GEOS_Compiled_Version

PostGIS_GEOS_Compiled_Version — Returns the version number of the GEOS library against which PostGIS was built.

Synopsis

text **PostGIS_GEOS_Compiled_Version**();

Description

Returns the version number of the GEOS library, or against which PostGIS was built.

Availability: 3.4.0

Examples

Code

```
SELECT PostGIS_GEOS_Compiled_Version();
```

Output

```
3.12.0
```

See Also

PostGIS_GEOS_Version, PostGIS_Full_Version

7.21.5 PostGIS_Liblwgeom_Version

PostGIS_Liblwgeom_Version — Returns the version number of the liblwgeom library. This should match the version of PostGIS.

Synopsis

text **PostGIS_Liblwgeom_Version**();

Description

Returns the version number of the liblwgeom library/

Examples

Code

```
SELECT PostGIS_Liblwgeom_Version();
```

Output

```
3.4.0dev 3.3.0rc2-993-g61bdf43a7
```

See Also

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.6 PostGIS_LibXML_Version

PostGIS_LibXML_Version — Returns the version number of the libxml2 library.

Synopsis

text **PostGIS_LibXML_Version**();

Description

Returns the version number of the LibXML2 library.
Availability: 1.5

Examples

Code

```
SELECT PostGIS_LibXML_Version();
```

Output

```
2.9.10
```

See Also

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_PROJ_Version, PostGIS_GEOS_Version, PostGIS_Version

7.21.7 PostGIS_LibJSON_Version

PostGIS_LibJSON_Version — Returns the version number of the libjson-c library.

Synopsis

text **PostGIS_LibJSON_Version**();

Description

Returns the version number of the LibJSON-C library.

Examples

Code

```
SELECT PostGIS_LibJSON_Version();
```

Output

```
0.17
```

See Also

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_PROJ_Version, PostGIS_GEOS_Version, PostGIS_Version

7.21.8 PostGIS_Lib_Build_Date

PostGIS_Lib_Build_Date — Returns build date of the PostGIS library.

Synopsis

text **PostGIS_Lib_Build_Date**();

Description

Returns build date of the PostGIS library.

Examples

Code

```
SELECT PostGIS_Lib_Build_Date();
```

Output

```
2023-06-22 03:56:11
```

7.21.9 PostGIS_Lib_Version

PostGIS_Lib_Version — Returns the version number of the PostGIS library.

Synopsis

text **PostGIS_Lib_Version**();

Description

Returns the version number of the PostGIS library.

Examples

Code

```
SELECT PostGIS_Lib_Version();
```

Output

```
3.4.0dev
```

See Also

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.10 PostGIS_PROJ_Version

PostGIS_PROJ_Version — Returns the version number of the PROJ4 library.

Synopsis

text **PostGIS_PROJ_Version**();

Description

Returns the version number of the PROJ library and some configuration options of proj.

Enhanced: 3.4.0 now includes NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location

Examples

Code

```
SELECT PostGIS_PROJ_Version();
```

Output

```
7.2.1 NETWORK_ENABLED=OFF URL_ENDPOINT=https://cdn.proj.org USER_WRITABLE_DIRECTORY=/tmp/ ↩
    proj DATABASE_PATH=/usr/share/proj/proj.db
```

See Also

PostGIS_PROJ_Compiled_Version, PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Versio
PostGIS_Version

7.21.11 PostGIS_PROJ_Compiled_Version

PostGIS_PROJ_Compiled_Version — Returns the version number of the PROJ library against which PostGIS was built.

Synopsis

text **PostGIS_PROJ_Compiled_Version**();

Description

Returns the version number of the PROJ library, or against which PostGIS was built.

Availability: 3.5.0

Examples

Code

```
SELECT PostGIS_PROJ_Compiled_Version();
```

Output

```
9.1.1
```

See Also

PostGIS_PROJ_Version, PostGIS_Full_Version

7.21.12 PostGIS_Wagyu_Version

PostGIS_Wagyu_Version — Returns the version number of the internal Wagyu library.

Synopsis

text **PostGIS_Wagyu_Version**();

Description

Returns the version number of the internal Wagyu library, or `NULL` if Wagyu support is not enabled.

Examples

Code

```
SELECT PostGIS_Wagyu_Version();
```

Output

```
0.5.0 (Internal)
```

See Also

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_PROJ_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_Version

7.21.13 PostGIS_Scripts_Build_Date

PostGIS_Scripts_Build_Date — Returns build date of the PostGIS scripts.

Synopsis

text **PostGIS_Scripts_Build_Date**();

Description

Returns build date of the PostGIS scripts.

Availability: 1.0.0RC1

Examples

Code

```
SELECT PostGIS_Scripts_Build_Date();
```

Output

```
2023-06-22 03:56:11
```

See Also

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_Version

7.21.14 PostGIS_Scripts_Installed

PostGIS_Scripts_Installed — Returns version of the PostGIS scripts installed in this database.

Synopsis

text **PostGIS_Scripts_Installed**();

Description

Returns version of the PostGIS scripts installed in this database.



Note
If the output of this function doesn't match the output of PostGIS_Scripts_Released you probably missed to properly upgrade an existing database. See the Upgrading section for more info.

Availability: 0.9.0

Examples

Code

```
SELECT PostGIS_Scripts_Installed();
```

Output

```
3.4.0dev 3.3.0rc2-993-g61bdf43a7
```

See Also

PostGIS_Full_Version, PostGIS_Scripts_Released, PostGIS_Version

7.21.15 PostGIS_Scripts_Released

PostGIS_Scripts_Released — Returns the version number of the postgis.sql script released with the installed PostGIS lib.

Synopsis

text **PostGIS_Scripts_Released**();

Description

Returns the version number of the postgis.sql script released with the installed PostGIS lib.



Note
Starting with version 1.1.0 this function returns the same value of PostGIS_Lib_Version. Kept for backward compatibility.

Availability: 0.9.0

Examples

Code

```
SELECT PostGIS_Scripts_Released();
```

Output

```
3.4.0dev 3.3.0rc2-993-g61bdf43a7
```

See Also

PostGIS_Full_Version, PostGIS_Scripts_Installed, PostGIS_Lib_Version

7.21.16 PostGIS_Version

PostGIS_Version — Returns PostGIS version number and compile-time options.

Synopsis

text **PostGIS_Version**();

Description

Returns PostGIS version number and compile-time options.

Examples

Code

```
SELECT PostGIS_Version();
```

Output

```
3.4 USE_GEOS=1 USE_PROJ=1 USE_STATS=1
```

See Also

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version

7.22 Grand Unified Custom Variables (GUCs)

7.22.1 postgis.gdal_datapath

postgis.gdal_datapath — A configuration option to assign the value of GDAL's GDAL_DATA option. If not set, the environmentally set GDAL_DATA variable is used.

Description

A PostgreSQL GUC variable for setting the value of GDAL's GDAL_DATA option. The `postgis.gdal_datapath` value should be the complete physical path to GDAL's data files.

This configuration option is of most use for Windows platforms where GDAL's data files path is not hard-coded. This option should also be set when GDAL's data files are not located in GDAL's expected path.



Note
This option can be set in PostgreSQL's configuration file postgresql.conf. It can also be set by connection or transaction.

Availability: 2.2.0



Note
Additional information about GDAL_DATA is available at GDAL's Configuration Options.

Examples

Set and reset `postgis.gdal_datapath`

Code

```
SET postgis.gdal_datapath TO '/usr/local/share/gdal.hidden';
SET postgis.gdal_datapath TO default;
```

Setting on windows for a particular database

Code

```
ALTER DATABASE gisdb
SET postgis.gdal_datapath = 'C:/Program Files/PostgreSQL/9.3/gdal-data';
```

See Also

PostGIS_GDAL_Version, ST_Transform

7.22.2 postgis.gdal_enabled_drivers

postgis.gdal_enabled_drivers — A configuration option to set the enabled GDAL drivers in the PostGIS environment. Affects the GDAL configuration variable GDAL_SKIP.

Description

A configuration option to set the enabled GDAL drivers in the PostGIS environment. Affects the GDAL configuration variable GDAL_SKIP. This option can be set in PostgreSQL's configuration file: postgresql.conf. It can also be set by connection or transaction.

The initial value of `postgis.gdal_enabled_drivers` may also be set by passing the environment variable `POSTGIS_GDAL_EN` with the list of enabled drivers to the process starting PostgreSQL.

Enabled GDAL specified drivers can be specified by the driver's short-name or code. Driver short-names or codes can be found at GDAL Raster Formats. Multiple drivers can be specified by putting a space between each driver.



Note
There are three special codes available for `postgis.gdal_enabled_drivers`. The codes are case-sensitive.

- `DISABLE_ALL` disables all GDAL drivers. If present, `DISABLE_ALL` overrides all other values in `postgis.gdal_enabled_drivers`.
- `ENABLE_ALL` enables all GDAL drivers.
- `VSICURL` enables GDAL's `/vsicurl/` virtual file system.

When `postgis.gdal_enabled_drivers` is set to DISABLE_ALL, attempts to use out-db rasters, ST_FromGDALRaster(), ST_AsGDALRaster(), ST_AsTIFF(), ST_AsJPEG() and ST_AsPNG() will result in error messages.



Note
In the standard PostGIS installation, `postgis.gdal_enabled_drivers` is set to DISABLE_ALL.



Note
Additional information about GDAL_SKIP is available at GDAL's Configuration Options.

Availability: 2.2.0

Examples

Set and reset `postgis.gdal_enabled_drivers` for the current session.

Code

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';
SET postgis.gdal_enabled_drivers = default;
```

Set specific drivers for all new connections to a database.

Code

```
ALTER DATABASE mygisdb SET postgis.gdal_enabled_drivers TO 'GTiff PNG JPEG';
```

Set the default for the whole database cluster to enable all drivers. This requires superuser access. Also note that database, session, and user settings override this.

This example writes the setting to postgres.auto.conf.

Code

```
ALTER SYSTEM SET postgis.gdal_enabled_drivers TO 'ENABLE_ALL';
```

Reload the PostgreSQL configuration so the changed setting is applied.

Code

```
SELECT pg_reload_conf();
```

See Also

ST_FromGDALRaster, ST_AsGDALRaster, ST_AsTIFF, ST_AsPNG, ST_AsJPEG, postgis.enable_outdb_rasters

7.22.3 postgis.enable_outdb_rasters

postgis.enable_outdb_rasters — A boolean configuration option to enable access to out-db raster bands.

Description

A boolean configuration option to enable access to out-db raster bands. This option can be set in PostgreSQL's configuration file: postgresql.conf. It can also be set by connection or transaction.

The initial value of `postgis.enable_outdb_rasters` may also be set by passing the environment variable `POSTGIS_ENABLE_` with a non-zero value to the process starting PostgreSQL.



Note
Even if `postgis.enable_outdb_rasters` is True, the GUC `postgis.gdal_enabled_drivers` determines the accessible raster formats.



Note
In the standard PostGIS installation, `postgis.enable_outdb_rasters` is set to False.

Availability: 2.2.0

Examples

Set and reset `postgis.enable_outdb_rasters` for the current session.

Code

```
SET postgis.enable_outdb_rasters TO True;
SET postgis.enable_outdb_rasters = default;
SET postgis.enable_outdb_rasters = True;
SET postgis.enable_outdb_rasters = False;
```

Set the default for all new connections to a database.

Code

```
ALTER DATABASE gisdb SET postgis.enable_outdb_rasters = true;
```

Set the default for the whole database cluster. This requires superuser access. Also note that database, session, and user settings override this.

This example writes the setting to postgres.auto.conf.

Code

```
ALTER SYSTEM SET postgis.enable_outdb_rasters = true;
```

Reload the PostgreSQL configuration so the changed setting is applied.

Code

```
SELECT pg_reload_conf();
```

See Also

postgis.gdal_enabled_drivers postgis.gdal_vsi_options

7.22.4 postgis.gdal_vsi_options

postgis.gdal_vsi_options — A string configuration to set options used when working with an out-db raster.

Description

A string configuration to set options used when working with an out-db raster. Configuration options control things like how much space GDAL allocates to local data cache, whether to read overviews, and what access keys to use for remote out-db data sources.

Availability: 3.2.0

Examples

Set `postgis.gdal_vsi_options` for current session:

Code

```
SET postgis.gdal_vsi_options = 'AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxx AWS_SECRET_ACCESS_KEY= ↩
    yyyyyyyyyyyyyyyyyyyyyyyyyy';
```

Set `postgis.gdal_vsi_options` just for the *current transaction* using the `LOCAL` keyword:

Code

```
SET LOCAL postgis.gdal_vsi_options = 'AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxx  ↩
    AWS_SECRET_ACCESS_KEY=yyyyyyyyyyyyyyyyyyyyyyyyyy';
```

See Also

postgis.enable_outdb_rasters postgis.gdal_enabled_drivers

7.22.5 postgis.gdal_cpl_debug

postgis.gdal_cpl_debug — A boolean configuration to turn logging of GDAL debug messages on and off.

Description

By default, GDAL logging is printed to stderr, and lower level debug messages are not printed at all. Turning this GUC to true will cause detailed GDAL logging to be sent into the PostgreSQL server log. PostgreSQL clients receive only generic GDAL notice, debug, warning, or error messages.

Availability: 3.6.0

See Also

postgis.enable_outdb_rasters postgis.gdal_enabled_drivers

7.23 Troubleshooting Functions

7.23.1 PostGIS_AddBBox

PostGIS_AddBBox — Add bounding box to the geometry.

Synopsis

geometry **PostGIS_AddBBox**(geometry geomA);

Description

Add bounding box to the geometry. This would make bounding box based queries faster, but will increase the size of the geometry.



Note
Bounding boxes are automatically added to geometries so in general this is not needed unless the generated bounding box somehow becomes corrupted or you have an old install that is lacking bounding boxes. Then you need to drop the old and re-add.

 This method supports Circular Strings and Curves.

Examples

Code

```
UPDATE sometable
SET geom =  PostGIS_AddBBox(geom)
WHERE PostGIS_HasBBox(geom) = false;
```

See Also

PostGIS_DropBBox, PostGIS_HasBBox

7.23.2 PostGIS_DropBBox

PostGIS_DropBBox — Drop the bounding box cache from the geometry.

Synopsis

geometry **PostGIS_DropBBox**(geometry geomA);

Description

Drop the bounding box cache from the geometry. This reduces geometry size, but makes bounding-box based queries slower. It is also used to drop a corrupt bounding box. A tale-tell sign of a corrupt cached bounding box is when your ST_Intersects and other relation queries leave out geometries that rightfully should return true.



Note
Bounding boxes are automatically added to geometries and improve speed of queries so in general this is not needed unless the generated bounding box somehow becomes corrupted or you have an old install that is lacking bounding boxes. Then you need to drop the old and re-add. This kind of corruption has been observed in 8.3-8.3.6 series whereby cached bboxes were not always recalculated when a geometry changed and upgrading to a newer version without a dump reload will not correct already corrupted boxes. So one can manually correct using below and re-add the bbox or do a dump reload.

 This method supports Circular Strings and Curves.

Examples

This example drops cached bounding boxes where the stored box is wrong. Applying `Box2D` to `ST_AsBinary` forces a fresh box calculation, while `Box2D` on the table geometry reads the cached box.

Code

```
UPDATE sometable
SET geom =  PostGIS_DropBBox(geom)
WHERE Not (Box2D(ST_AsBinary(geom)) = Box2D(geom));
```

Output

```
UPDATE ...
```

After dropping corrupt cached boxes, re-add them where needed.

Code

```
UPDATE sometable
SET geom =  PostGIS_AddBBox(geom)
WHERE Not PostGIS_HasBBOX(geom);
```

Output

```
UPDATE ...
```

See Also

PostGIS_AddBBox, PostGIS_HasBBox, Box2D

7.23.3 PostGIS_HasBBox

PostGIS_HasBBox — Returns TRUE if the bbox of this geometry is cached, FALSE otherwise.

Synopsis

boolean **PostGIS_HasBBox**(geometry geomA);

Description

Returns TRUE if the bbox of this geometry is cached, FALSE otherwise. Use PostGIS_AddBBox and PostGIS_DropBBox to control caching.

 This method supports Circular Strings and Curves.

Examples

Code

```
SELECT geom
FROM sometable WHERE PostGIS_HasBBox(geom) = false;
```

See Also

PostGIS_AddBBox, PostGIS_DropBBox

Chapter 8

SFCGAL Functions Reference

SFCGAL is a C++ wrapper library around CGAL that provides advanced 2D and 3D spatial functions. For robustness, geometry coordinates have an exact rational number representation.

Installation instructions for the library can be found on the SFCGAL home page (http://www.sfcgal.org). To enable the functions use `create extension postgis_sfcgal`.

8.1 SFCGAL Management Functions

8.1.1 postgis_sfcgal_version

postgis_sfcgal_version — Returns the version of SFCGAL in use

Synopsis

text **postgis_sfcgal_version**(void);

Description

Returns the version of SFCGAL in use

Availability: 2.1.0

 This method needs SFCGAL backend.

See Also

postgis_sfcgal_full_version

8.1.2 postgis_sfcgal_full_version

postgis_sfcgal_full_version — Returns the full version of SFCGAL in use including CGAL and Boost versions

Synopsis

text **postgis_sfcgal_full_version**(void);

Description

Returns the full version of SFCGAL in use including CGAL and Boost versions

Availability: 3.3.0

 This method needs SFCGAL backend.

See Also

postgis_sfcgal_version

8.2 SFCGAL Accessors and Setters

8.2.1 CG_ForceLHR

CG_ForceLHR — Force LHR orientation

Synopsis

geometry **CG_ForceLHR**(geometry geom);

Description

Returns an equivalent geometry whose polygonal components follow the left-hand rule: exterior rings are counter-clockwise and interior rings are clockwise. Rings are reversed as needed; coordinates and dimensionality are preserved. The input must be valid.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Reorient a polygon and its hole to the left-hand rule.

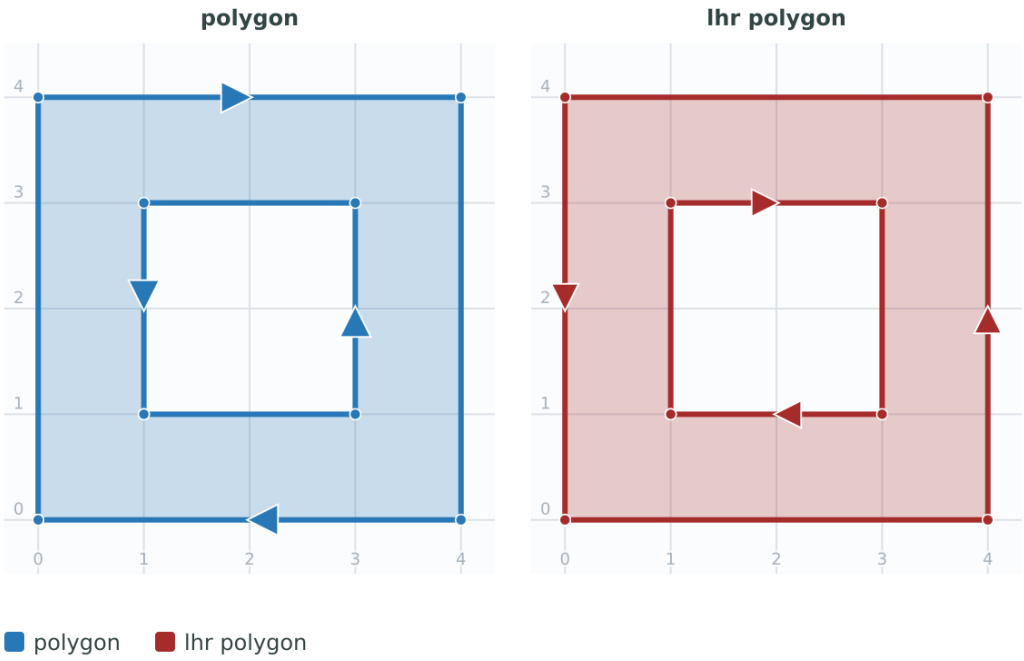
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,0 4,4 4,4 0,0 0),
                  (1 1,3 1,3 3,1 3,1 1))'::geometry AS geom
)
SELECT geom AS input_polygon,
       CG_ForceLHR(geom) AS lhr_polygon,
       ST_IsPolygonCCW(geom) AS input_is_lhr,
       ST_IsPolygonCCW(CG_ForceLHR(geom)) AS output_is_lhr
FROM data;
```

Output

```
POLYGON((0 0,0 4,4 4,4 0,0 0),(1 1,3 1,3 3,1 3,1 1)) | POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,1 3,3 3,3 1,1 1)) | f | t
```

Figure



8.2.2 CG_IsPlanar

CG_IsPlanar — Check if a surface is or not planar

Synopsis

boolean **CG_IsPlanar**(geometry geom);

Description

Tests whether the input geometry lies in a single plane in 3D.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

8.2.3 CG_IsSolid

CG_IsSolid — Test if the geometry is a solid. No validity check is performed.

Synopsis

boolean **CG_IsSolid**(geometry geom1);

Description

Returns `true` when the input carries the solid flag, such as a geometry produced by CG_MakeSolid. A closed `PolyhedralSurface` remains a surface until it is explicitly converted. This tests the representation, not shell closure or validity; no validity check is performed.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Compare the same closed shell before and after converting it to a solid.

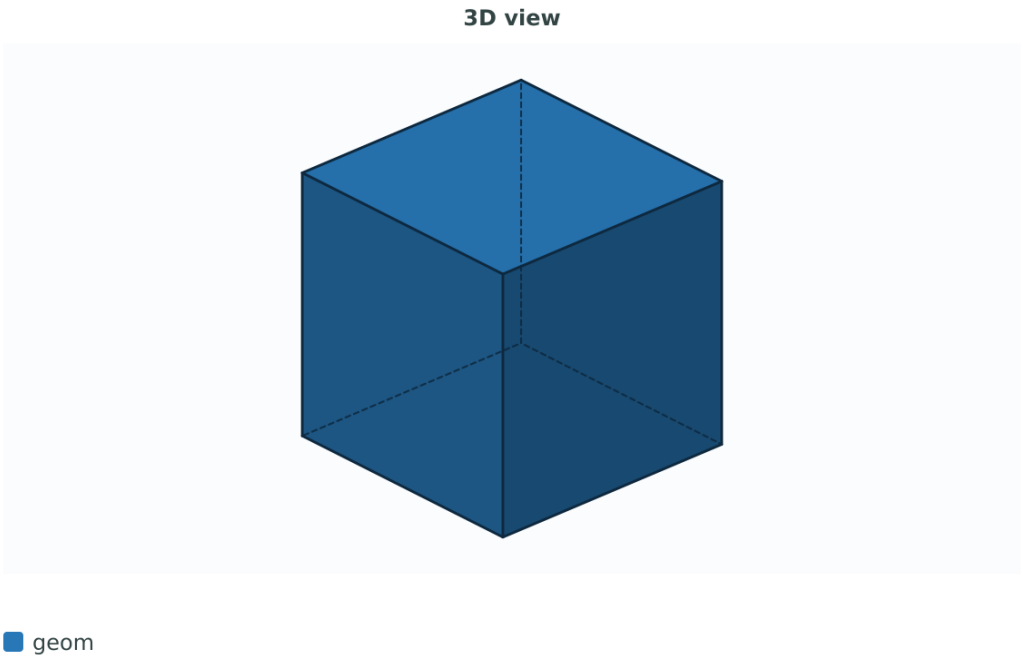
Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z (
    ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
    ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
    ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
  )'::geometry AS geom
)
SELECT CG_IsSolid(geom) AS shell_is_solid,
       CG_IsSolid(CG_MakeSolid(geom)) AS solid_is_solid
FROM data;
```

Output

```
f | t
```

Figure



8.2.4 CG_MakeSolid

CG_MakeSolid — Cast the geometry into a solid. No check is performed. To obtain a valid solid, the input geometry must be a closed Polyhedral Surface or a closed TIN.

Synopsis

geometry **CG_MakeSolid**(geometry geom1);

Description

Marks a PolyhedralSurface or TIN as a solid. The input is expected to describe a closed shell; the function sets the solid representation and does not validate the shell.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

8.2.5 CG_Orientation

CG_Orientation — Determine surface orientation

Synopsis

integer **CG_Orientation**(geometry geom);

Description

The function only applies to polygons. It returns -1 if the polygon is counterclockwise oriented and 1 if the polygon is clockwise oriented.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

8.2.6 CG_Area

CG_Area — Calculates the area of a geometry

Synopsis

double precision **CG_Area**(geometry geom);

Description

Calculates the area of a geometry.

Performed by the SFCGAL module



Note
NOTE: this function returns a double precision value representing the area.

Availability: 3.5.0

 This method needs SFCGAL backend.

Geometry Examples

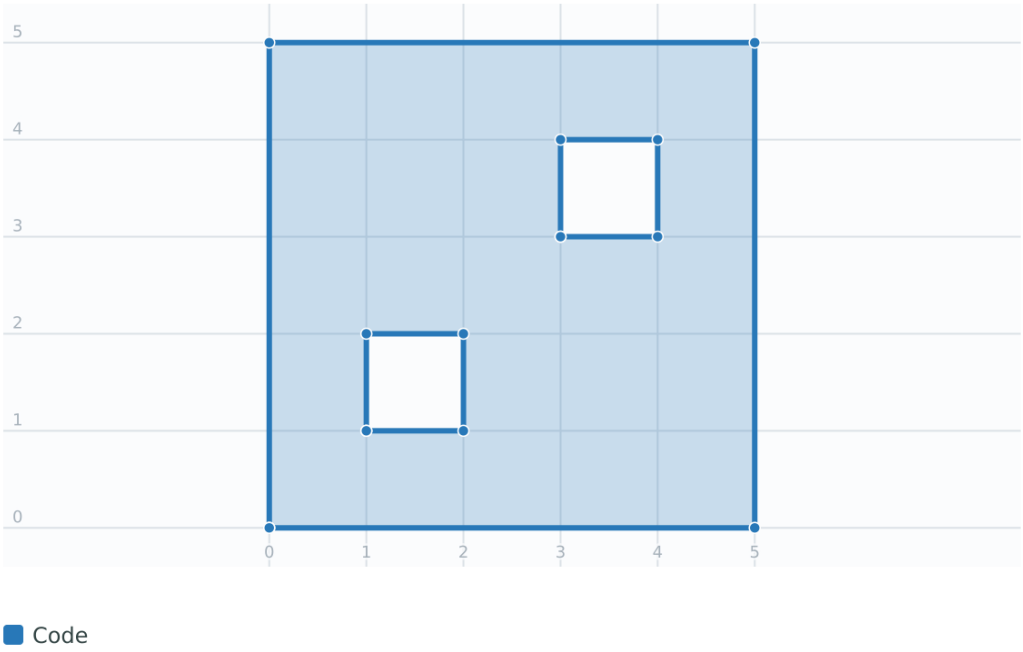
Code

```
SELECT CG_Area('Polygon ((0 0,0 5,5 5,5 0,0 0),(1 1,2 1,2 2,1 2,1 1),(3 3,4 3,4 4,3 4,3 3))' ↩
    );
```

Output

```
23
```

Figure



See Also

CG_3DArea, ST_Area

8.2.7 CG_3DArea

CG_3DArea — Computes area of 3D surface geometries. Will return 0 for solids.

Synopsis

float**CG_3DArea**(geometry geom1);

Description

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 8.1, 10.5

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Note: By default a PolyhedralSurface built from WKT is a surface geometry, not solid. It therefore has surface area. Once converted to a solid, no area.

Code

```
SELECT CG_3DArea(geom) As cube_surface_area,
CG_3DArea(CG_MakeSolid(geom)) As solid_surface_area
FROM (SELECT 'POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry) As f(geom);
```

Output

```
cube_surface_area | solid_surface_area
------------------+--------------------
6                 | 0
```

See Also

CG_Area, CG_MakeSolid, CG_IsSolid, CG_Area

8.2.8 CG_Volume

CG_Volume — Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.

Synopsis

float **CG_Volume**(geometry geom1);

Description

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 9.1 (same as CG_3DVolume)

Closed surface geometries can have non-zero volume. Use CG_MakeSolid when the geometry value itself must carry the solid flag.

Example

A closed polyhedral surface has volume. CG_MakeSolid makes the solid representation explicit.

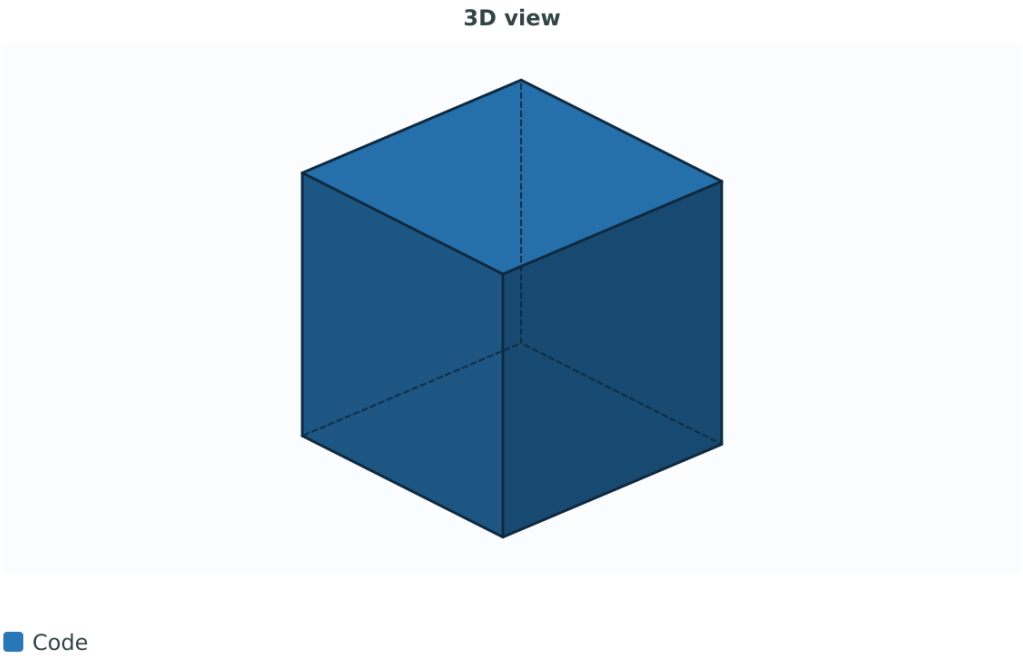
Code

```
SELECT CG_Volume(geom) As cube_surface_vol,
CG_Volume(CG_MakeSolid(geom)) As solid_surface_vol
FROM (SELECT 'POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry) As f(geom);
```

Output

```
cube_surface_vol | solid_surface_vol
-----------------+------------------
1                | 1
```

Figure



See Also

CG_3DArea, CG_MakeSolid, CG_IsSolid

8.3 SFCGAL Processing and Relationship Functions

8.3.1 CG_Intersection

CG_Intersection — Computes the intersection of two geometries

Synopsis

geometry **CG_Intersection**(geometry geomA , geometry geomB);

Description

Computes the intersection of two geometries.

Performed by the SFCGAL module



Note

NOTE: this function returns a geometry representing the intersection.

Availability: 3.5.0



This method needs SFCGAL backend.

Geometry Examples

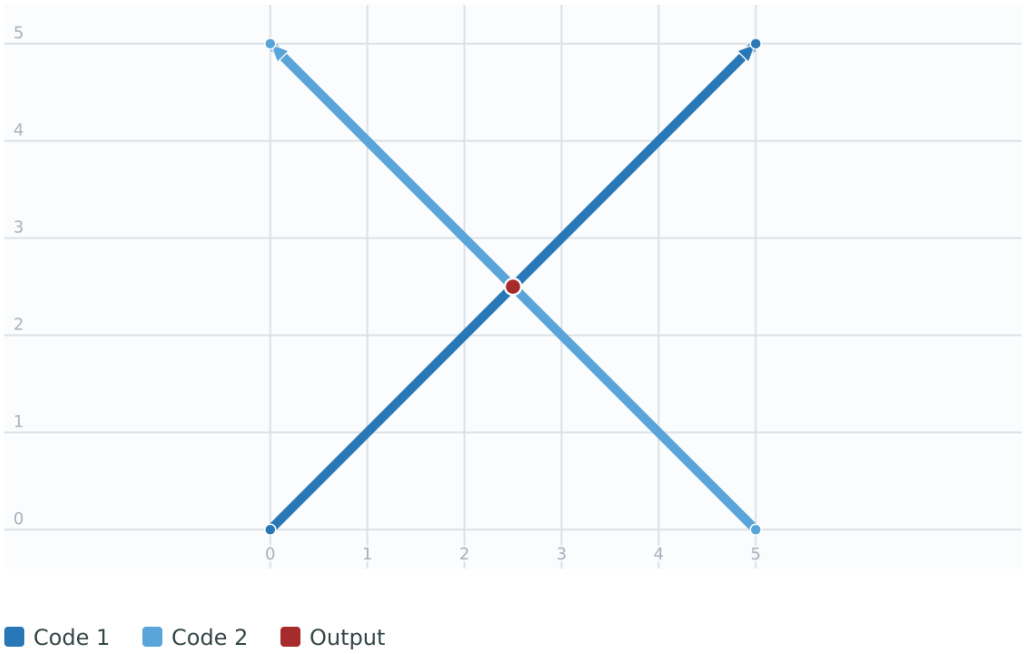
Code

```
SELECT CG_Intersection('LINESTRING(0 0,5 5)', 'LINESTRING(5 0,0 5)');
```

Output

```
POINT(2.5 2.5)
```

Figure



See Also

CG_3DIntersection, ST_Intersection

8.3.2 CG_Intersects

CG_Intersects — Tests if two geometries intersect (they have at least one point in common)

Synopsis

boolean **CG_Intersects**(geometry geomA , geometry geomB);

Description

Returns `true` if two geometries intersect. Geometries intersect if they have any point in common.

Performed by the SFCGAL module



Note
NOTE: this is the "allowable" version that returns a boolean, not an integer.

Availability: 3.5.0


This method needs SFCGAL backend.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Geometry Examples

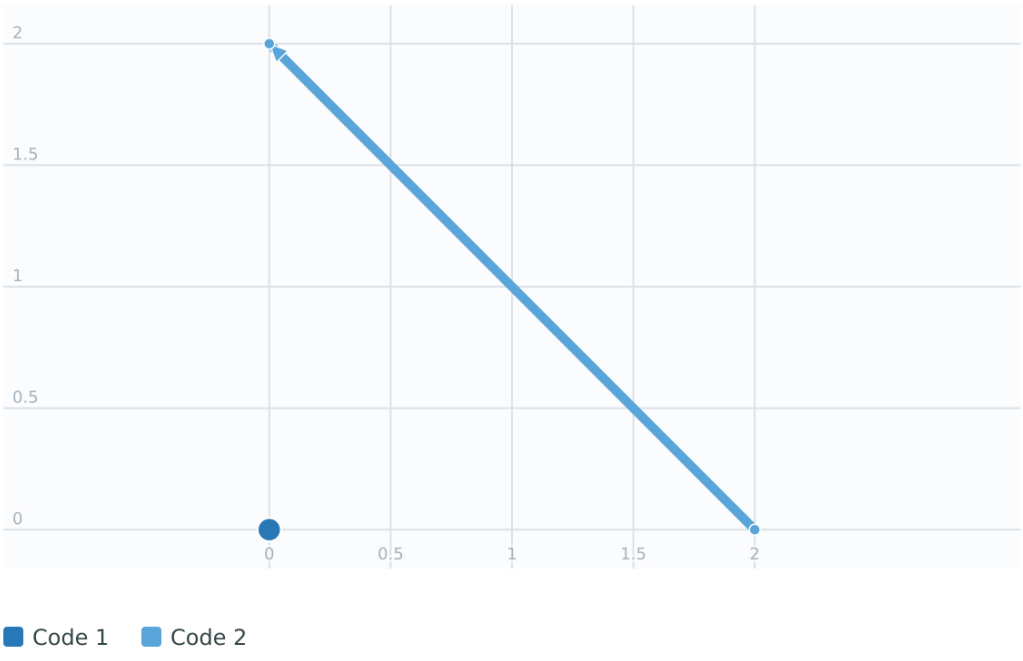
Code

```
SELECT CG_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
f
```

Figure



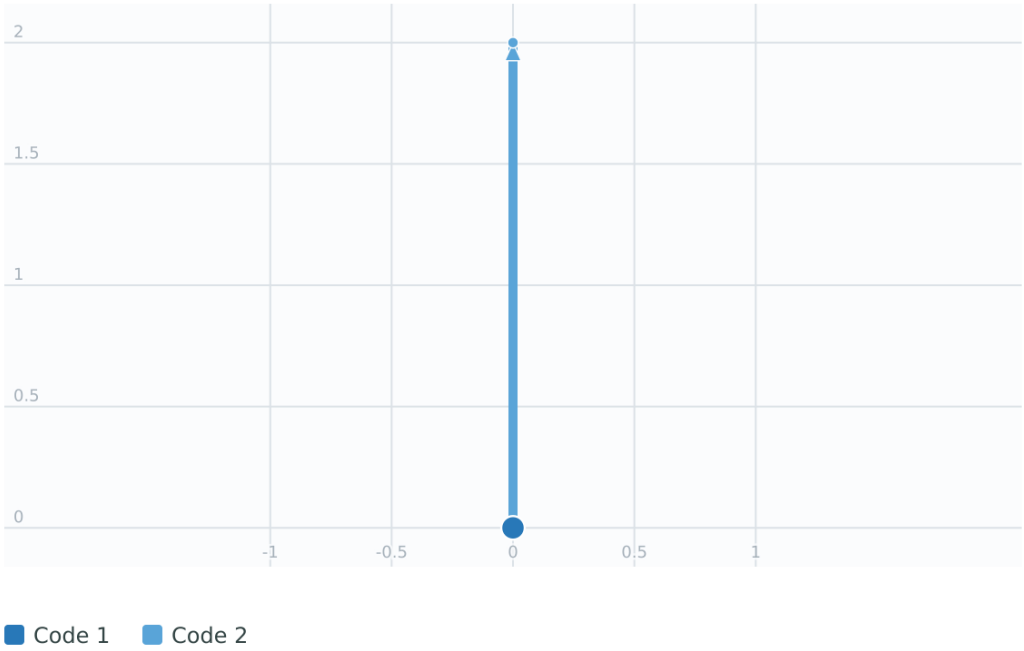
Code

```
SELECT CG_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Output

```
t
```

Figure



See Also

CG_3DIntersects, ST_3DIntersects, ST_Intersects, ST_Disjoint

8.3.3 CG_3DIntersects

CG_3DIntersects — Tests if two 3D geometries intersect

Synopsis

boolean **CG_3DIntersects**(geometry geomA , geometry geomB);

Description

Tests if two 3D geometries intersect. 3D geometries intersect if they have any point in common in the three-dimensional space. Performed by the SFCGAL module

> **Note**
> NOTE: this is the "allowable" version that returns a boolean, not an integer.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Geometry Examples

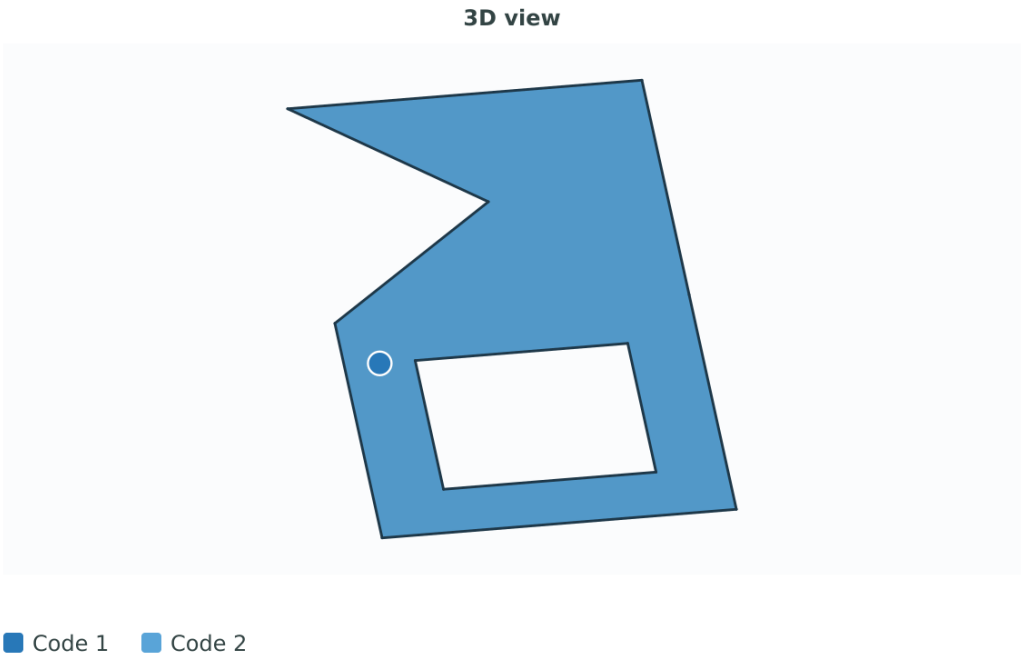
Code

```
SELECT CG_3DIntersects('POINT(1.2 0.1 0)', 'POLYHEDRALSURFACE(((0 0 0,0.5 0.5 0,1 0 0,1 1 ↩
    0,0 1 0,0 0 0)),((1 0 0,2 0 0,2 1 0,1 1 0,1 0 0),(1.2 0.2 0,1.2 0.8 0,1.8 0.8 0,1.8 0.2 ↩
    0,1.2 0.2 0)))');
```

Output

```
t
```

Figure



See Also

CG_Intersects, ST_3DIntersects, ST_Intersects, ST_Disjoint

8.3.4 CG_Difference

CG_Difference — Computes the geometric difference between two geometries

Synopsis

geometry **CG_Difference**(geometry geomA , geometry geomB);

Description

Computes the geometric difference between two geometries. The resulting geometry is a set of points that are present in geomA but not in geomB.

Performed by the SFCGAL module



Note
NOTE: this function returns a geometry.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Geometry Examples

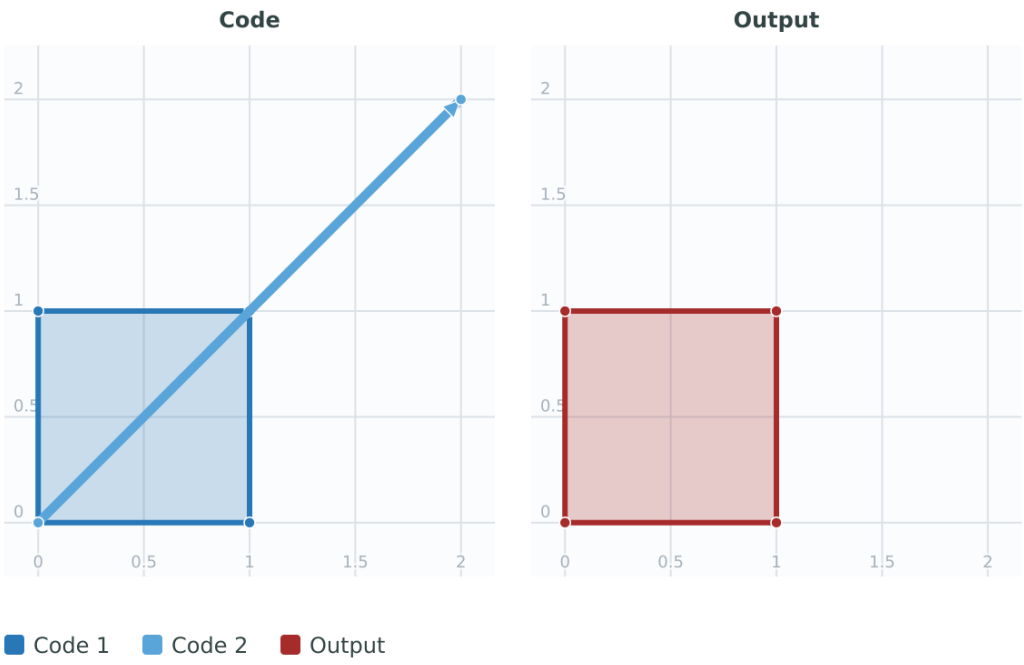
Code

```
SELECT CG_Difference('POLYGON((0 0,0 1,1 1,1 0,0 0))'::geometry, 'LINESTRING(0 0,2 2)':: ↩
    geometry);
```

Output

```
POLYGON((0 0,1 0,1 1,0 1,0 0))
```

Figure



See Also

CG_3DDifference, ST_Difference

8.3.5 CG_3DDifference

CG_3DDifference — Perform 3D difference

Synopsis

geometry **CG_3DDifference**(geometry geom1, geometry geom2);

Description

Returns that part of geom1 that is not part of geom2.

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

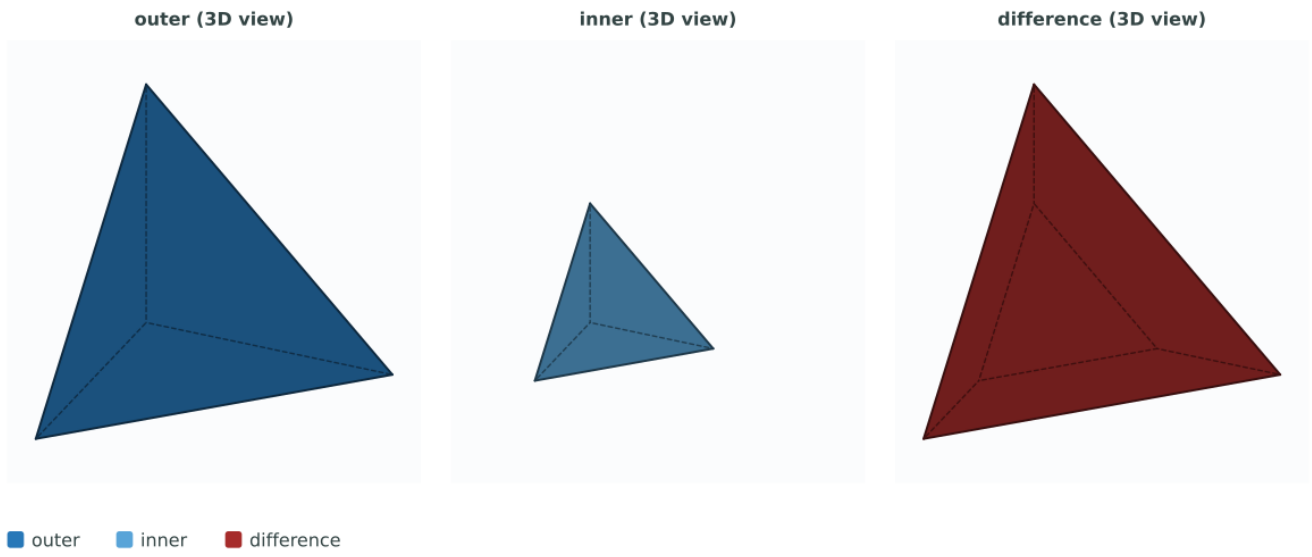
Examples

Subtract a smaller tetrahedral solid and render the two inputs and the actual remaining solid separately.

Code

```
WITH data AS (
  SELECT CG_MakeSolid('POLYHEDRALSURFACE Z (
    ((0 0 0,0 4 0,4 0 0,0 0 0)),
    ((0 0 0,4 0 0,0 0 4,0 0 0)),
    ((0 0 0,0 0 4,0 4 0,0 0 0)),
    ((4 0 0,0 4 0,0 0 4,4 0 0))
  )'::geometry) AS outer_solid
), solids AS (
  SELECT outer_solid,
         ST_Scale(outer_solid, 0.5, 0.5, 0.5) AS inner_solid
  FROM data
)
SELECT outer_solid AS input_outer,
       inner_solid AS input_inner,
       CG_3DDifference(outer_solid, inner_solid) AS difference
FROM solids;
```

Figure



See Also

CG_Extrude, ST_AsX3D, CG_3DIntersection CG_3DUnion

8.3.6 CG_Distance

CG_Distance — Computes the minimum distance between two geometries

Synopsis

double precision **CG_Distance**(geometry geomA , geometry geomB);

Description

Computes the minimum distance between two geometries.

Performed by the SFCGAL module

> **Note**
> NOTE: this function returns a double precision value representing the distance.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Geometry Examples

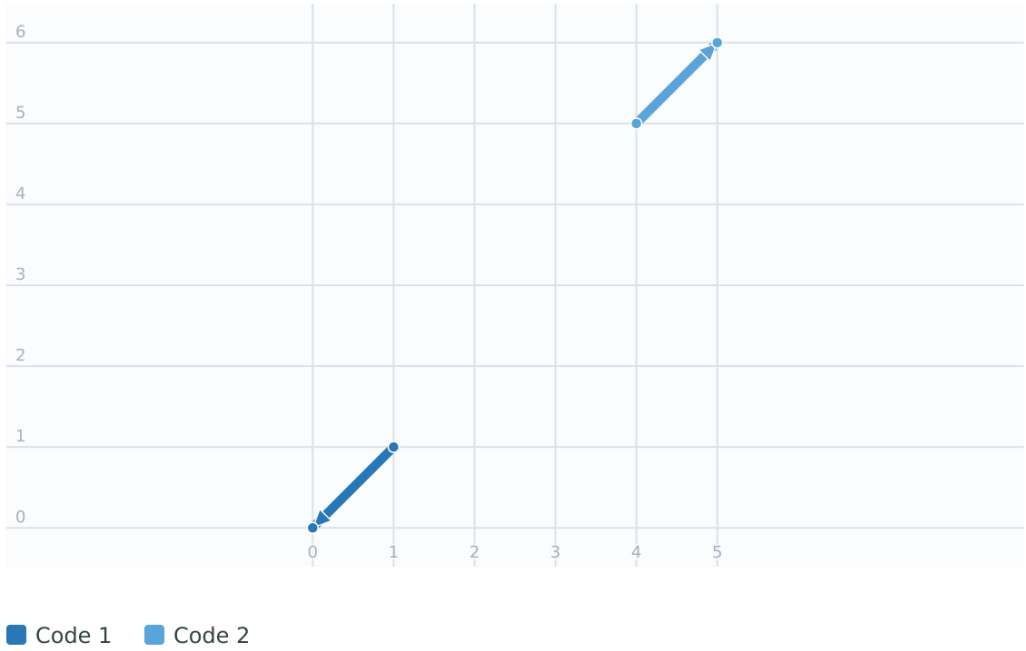
Code

```
SELECT CG_Distance('LINESTRING(0.0 0.0,-1.0 -1.0)', 'LINESTRING(3.0 4.0,4.0 5.0)');
```

Output

```
5
```

Figure



See Also

CG_3DDistance, CG_Distance

8.3.7 CG_3DDistance

CG_3DDistance — Computes the minimum 3D distance between two geometries

Synopsis

double precision **CG_3DDistance**(geometry geomA , geometry geomB);

Description

Computes the minimum 3D distance between two geometries.

Performed by the SFCGAL module



Note
NOTE: this function returns a double precision value representing the 3D distance.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Geometry Examples

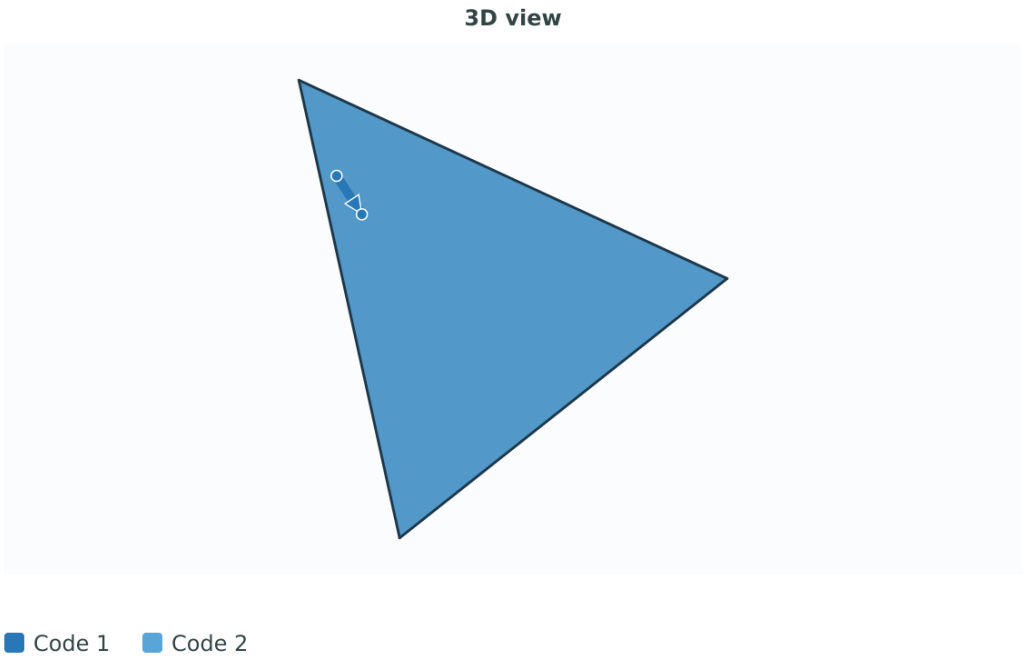
Code

```
SELECT CG_3DDistance('LINESTRING(-1.0 0.0 2.0,1.0 0.0 3.0)', 'TRIANGLE((-4.0 0.0 1.0,4.0 0.0 ↩
    1.0,0.0 4.0 1.0,-4.0 0.0 1.0))');
```

Output

```
1
```

Figure



See Also

CG_Distance, ST_3DDistance

8.3.8 CG_3DConvexHull

CG_3DConvexHull — Computes the 3D convex hull of a geometry.

Synopsis

geometry **CG_3DConvexHull**(geometry geom1);

Description

Computes the smallest convex three-dimensional geometry that contains the input. Unlike ST_ConvexHull, this function uses Z coordinates when constructing the hull.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Compute the 3D convex hull of a 3D LineString.

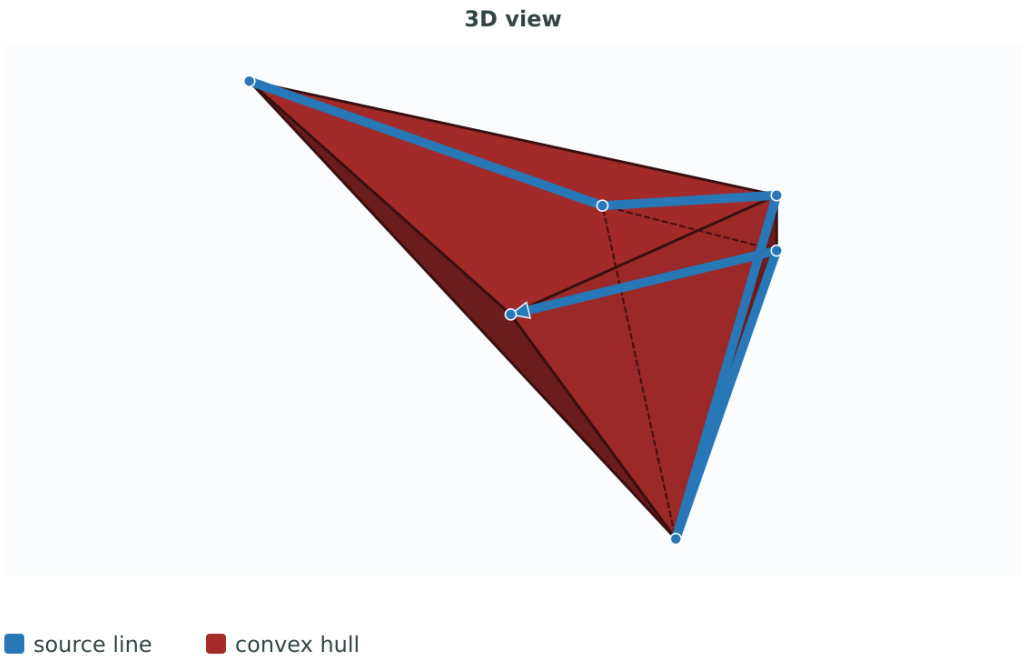
Code

```
WITH data AS (
  SELECT 'LINESTRING Z(0 0 5,1 5 3,5 7 6,9 5 3,5 7 5,6 3 5)'::geometry AS source_line
)
SELECT CG_3DConvexHull(source_line) AS convex_hull
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((1 5 3,9 5 3,0 0 5,1 5 3)),((1 5 3,0 0 5,5 7 6,1 5 3)),((5 7 6,5 7 5,1 5 3,5 7 6)),((0 0 5,6 3 ↩
    5,5 7 6,0 0 5)),((6 3 5,9 5 3,5 7 6,6 3 5)),((0 0 5,9 5 3,6 3 5,0 0 5)),((9 5 3,5 7 5,5 7 6,9 5 3)),((1 5 3,5 7 ↩
    5,9 5 3,1 5 3)))
```

Figure



Compare the extruded source geometry with its 3D convex hull.

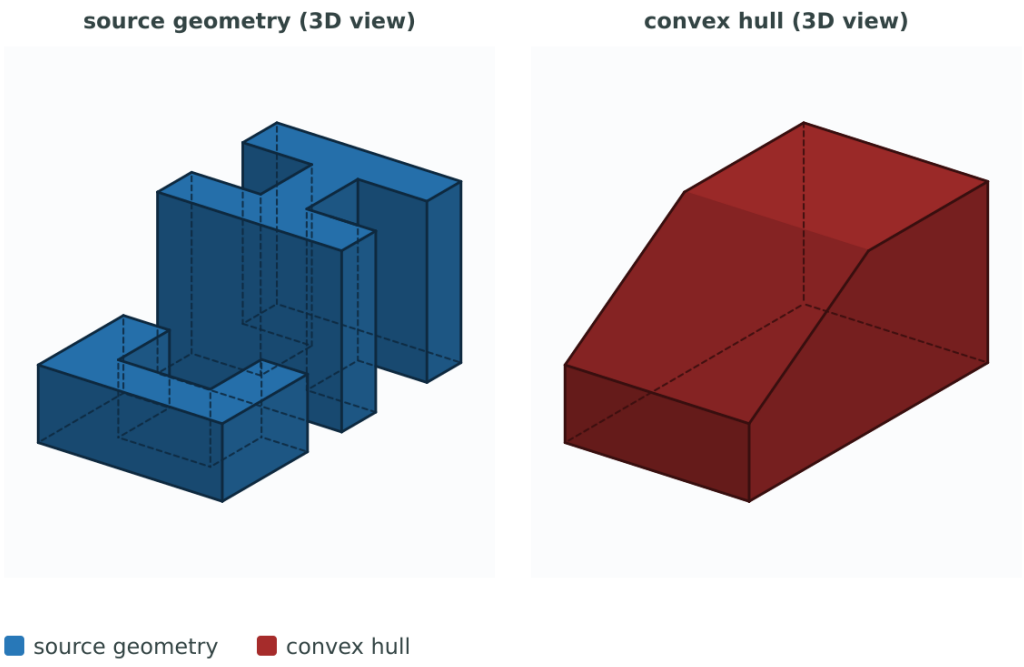
Code

```
WITH footprints(label, geom, height) AS (VALUES
  ('C', 'POLYGON((0 0,0 8,5 8,5 6,2 6,2 2,5 2,5 0,0 0))'::geometry, 3),
  ('H', 'POLYGON((7 0,9 0,9 3,12 3,12 0,14 0,14 8,12 8,12 5,9 5,9 8,7 8,7 0))'::geometry, 7)
), parts AS (
  SELECT label, CG_Extrude(geom, 0, 0, height) AS geom
  FROM footprints
), shapes AS (
  SELECT ST_Collect(geom ORDER BY label) AS source_geometry
  FROM parts
)
SELECT
  source_geometry AS input_source_geometry,
  CG_3DConvexHull(source_geometry) AS convex_hull
FROM shapes;
```

Output

```
-[ RECORD 1 ]----------
input_source_geometry | GEOMETRYCOLLECTION(POLYHEDRALSURFACE(((0 0 0,0 8 0,5 8 0,5 6 0,2 6 0,2 2 0,5 2 0,5 0 0,0 0 0)) ↩
     ,((0 0 3,5 0 3,5 2 3,2 2 3,2 6 3,5 6 3,5 8 3,0 8 3,0 0 3)),((0 0 0,0 0 3,0 8 3,0 8 0,0 0 0)),((0 8 0,0 8 3,5 8 3,5 ↩
      8 0,0 8 0)),((5 8 0,5 8 3,5 6 3,5 6 0,5 8 0)),((5 6 0,5 6 3,2 6 3,2 6 0,5 6 0)),((2 6 0,2 6 3,2 2 3,2 2 0,2 6 0)) ↩
     ,((2 2 0,2 2 3,5 2 3,5 2 0,2 2 0)),((5 2 0,5 2 3,5 0 3,5 0 0,5 2 0)),((5 0 0,5 0 3,0 0 3,0 0 0,5 0 0))),  ↩
     POLYHEDRALSURFACE(((7 0 0,7 8 0,9 8 0,9 5 0,12 5 0,12 8 0,14 8 0,14 0 0,12 0 0,12 3 0,9 3 0,9 0 0,7 0 0)),((7 0  ↩
     7,9 0 7,9 3 7,12 3 7,12 0 7,14 0 7,14 8 7,12 8 7,12 5 7,9 5 7,9 8 7,7 8 7,7 0 7)),((7 0 0,7 0 7,7 8 7,7 8 0,7 0 0) ↩
     ),((7 8 0,7 8 7,9 8 7,9 8 0,7 8 0)),((9 8 0,9 8 7,9 5 7,9 5 0,9 8 0)),((9 5 0,9 5 7,12 5 7,12 5 0,9 5 0)),((12 5  ↩
     0,12 5 7,12 8 7,12 8 0,12 5 0)),((12 8 0,12 8 7,14 8 7,14 8 0,12 8 0)),((14 8 0,14 8 7,14 0 7,14 0 0,14 8 0)),((14 ↩
      0 0,14 0 7,12 0 7,12 0 0,14 0 0)),((12 0 0,12 0 7,12 3 7,12 3 0,12 0 0)),((12 3 0,12 3 7,9 3 7,9 3 0,12 3 0)),((9 ↩
      3 0,9 3 7,9 0 7,9 0 0,9 3 0)),((9 0 0,9 0 7,7 0 7,7 0 0,9 0 0))))
convex_hull           | POLYHEDRALSURFACE(((14 8 7,14 0 0,14 8 0,14 8 7)),((14 0 7,7 8 7,7 0 7,14 0 7)),((0 8 3,7 8 7,0 ↩
      8 0,0 8 3)),((0 0 3,7 8 7,0 8 3,0 0 3)),((0 0 3,0 8 3,0 0 0,0 0 3)),((7 0 7,7 8 7,0 0 3,7 0 7)),((14 0 7,14 0  ↩
     0,14 8 7,14 0 7)),((7 0 7,14 0 0,14 0 7,7 0 7)),((14 8 7,7 8 7,14 0 7,14 8 7)),((0 0 0,0 8 3,0 8 0,0 0 0)),((0 0  ↩
     0,14 0 0,0 0 3,0 0 0)),((0 0 3,14 0 0,7 0 7,0 0 3)),((0 8 0,14 0 0,0 0 0,0 8 0)),((14 8 0,14 0 0,0 8 0,14 8 0)) ↩
     ,((14 8 0,7 8 7,14 8 7,14 8 0)),((0 8 0,7 8 7,14 8 0,0 8 0)))
```

Figure



See Also

ST_Letters, ST_AsX3D

8.3.9 CG_3DIntersection

CG_3DIntersection — Perform 3D intersection

Synopsis

geometry **CG_3DIntersection**(geometry geom1, geometry geom2);

Description

Return a geometry that is the shared portion between geom1 and geom2.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

3D linestrings and polygons can intersect in a 3D segment.

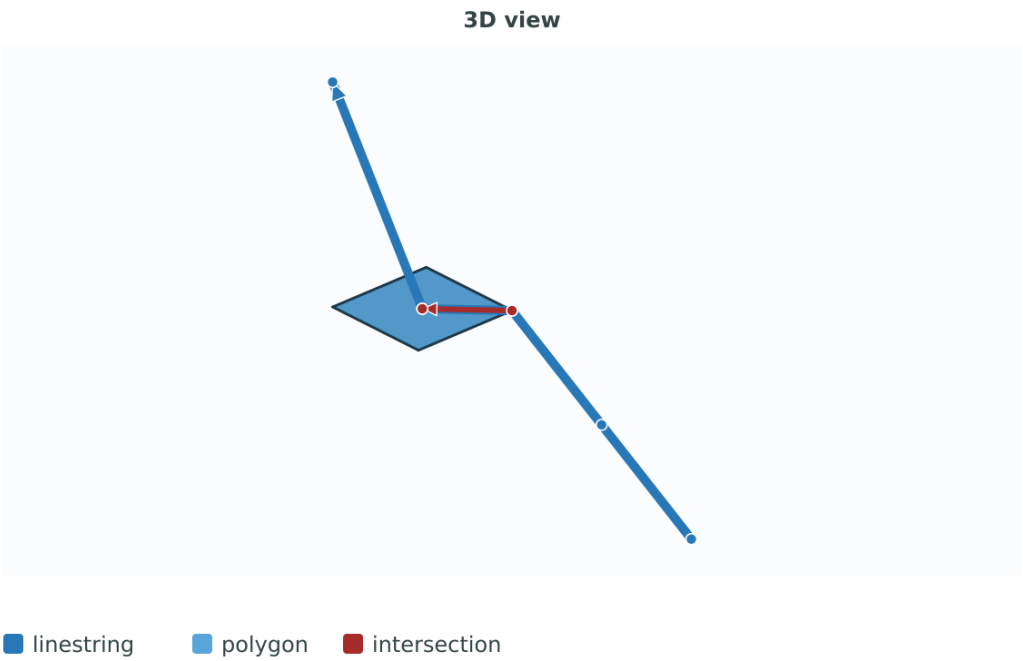
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10)'::geometry AS ↩
    input_linestring,
         'POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8))'::geometry AS input_polygon
)
SELECT input_linestring AS input_linestring,
       input_polygon AS input_polygon,
       CG_3DIntersection(input_linestring, input_polygon) AS intersection
FROM data;
```

Output

```
LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10) | POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8)) | LINESTRING Z (1 1 ↩
     8,0.5 0.5 8)
```

Figure



Intersect a cube shell with a vertical polygon.

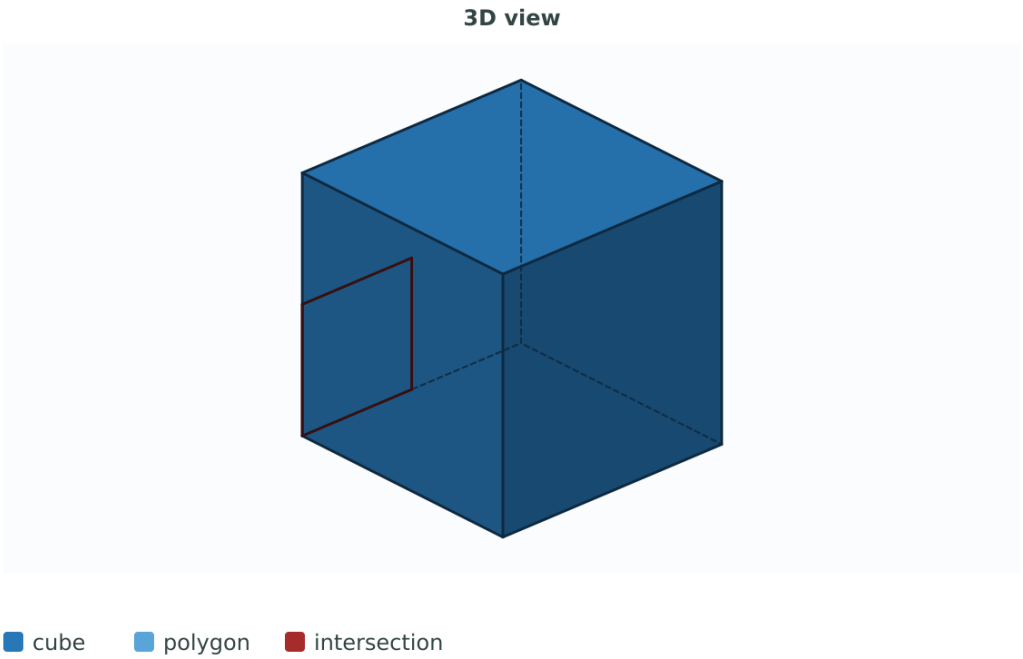
Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z(
    ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
    ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
    ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
  )'::geometry AS input_cube,
  'POLYGON Z ((0 0 0,0 0 0.5,0 0.5 0.5,0 0.5 0,0 0 0))'::geometry AS input_polygon
)
SELECT input_cube AS input_cube,
       input_polygon AS input_polygon,
       CG_3DIntersection(input_cube, input_polygon) AS intersection
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 ↩
    0)),((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))) | ↩
    POLYGON Z ((0 0 0,0 0 0.5,0 0.5 0.5,0 0.5 0,0 0 0)) | TIN Z (((0 0 0,0 0 0.5,0 0.5 0.5,0 0 0)),((0 0.5 0,0 0 0,0 ↩
    0.5 0.5,0 0.5 0)))
```

Figure



A smaller solid contained by a larger solid remains volumetric after intersection.

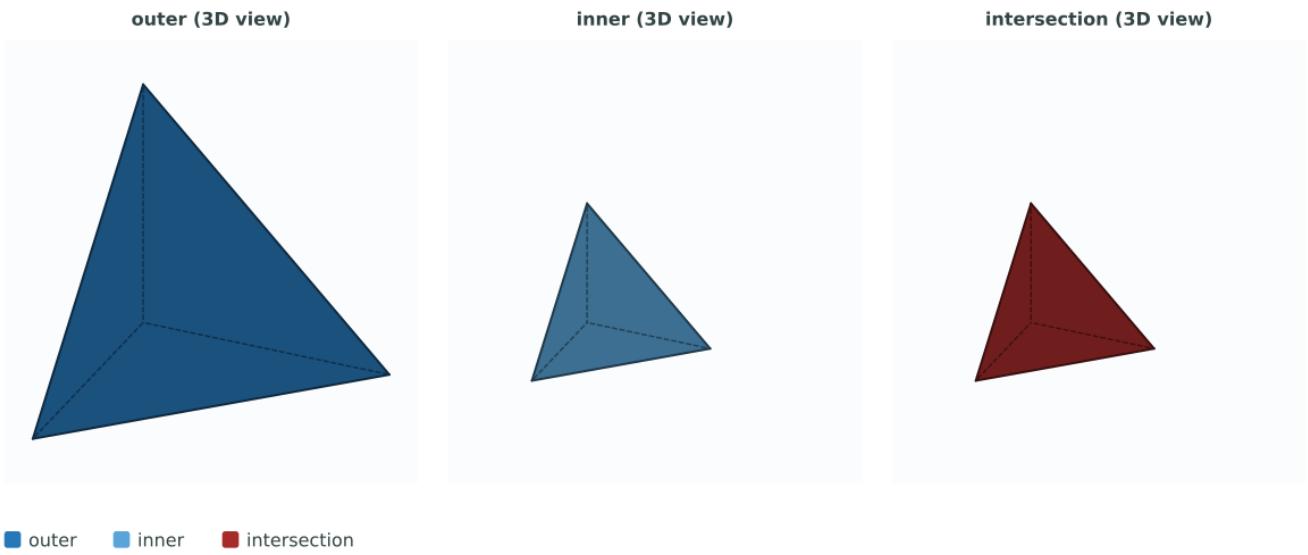
Code

```
WITH data AS (
  SELECT CG_MakeSolid('POLYHEDRALSURFACE Z (
    ((0 0 0,0 4 0,4 0 0,0 0 0)),
    ((0 0 0,4 0 0,0 0 4,0 0 0)),
    ((0 0 0,0 0 4,0 4 0,0 0 0)),
    ((4 0 0,0 4 0,0 0 4,4 0 0))
  )'::geometry) AS outer_solid
), solids AS (
  SELECT outer_solid,
         ST_Scale(outer_solid, 0.5, 0.5, 0.5) AS inner_solid
  FROM data
)
SELECT outer_solid AS input_outer,
       inner_solid AS input_inner,
       CG_3DIntersection(outer_solid, inner_solid) AS intersection
FROM solids;
```

Output

```
POLYHEDRALSURFACE Z (((0 0 0,0 4 0,4 0 0,0 0 0)),((0 0 0,4 0 0,0 0 4,0 0 0)),((0 0 0,0 0 4,0 4 0,0 0 0)),((4 0 0,0 4 ↩
    0,0 0 4,4 0 0))) | POLYHEDRALSURFACE Z (((0 0 0,0 2 0,2 0 0,0 0 0)),((0 0 0,2 0 0,0 0 2,0 0 0)),((0 0 0,0 0 2,0 2 ↩
    0,0 0 0)),((2 0 0,0 2 0,0 0 2,2 0 0))) | POLYHEDRALSURFACE Z (((0 0 0,0 2 0,2 0 0,0 0 0)),((0 0 0,0 0 2,0 2 0,0 0 ↩
    0)),((2 0 0,0 0 2,0 0 0,2 0 0)),((2 0 0,0 2 0,0 0 2,2 0 0)))
```

Figure



8.3.10 CG_Union

CG_Union — Computes the union of two geometries

Synopsis

geometry **CG_Union**(geometry geomA , geometry geomB);

Description

Computes the union of two geometries.

Performed by the SFCGAL module



Note
NOTE: this function returns a geometry representing the union.

Availability: 3.5.0


This method needs SFCGAL backend.

Geometry Examples

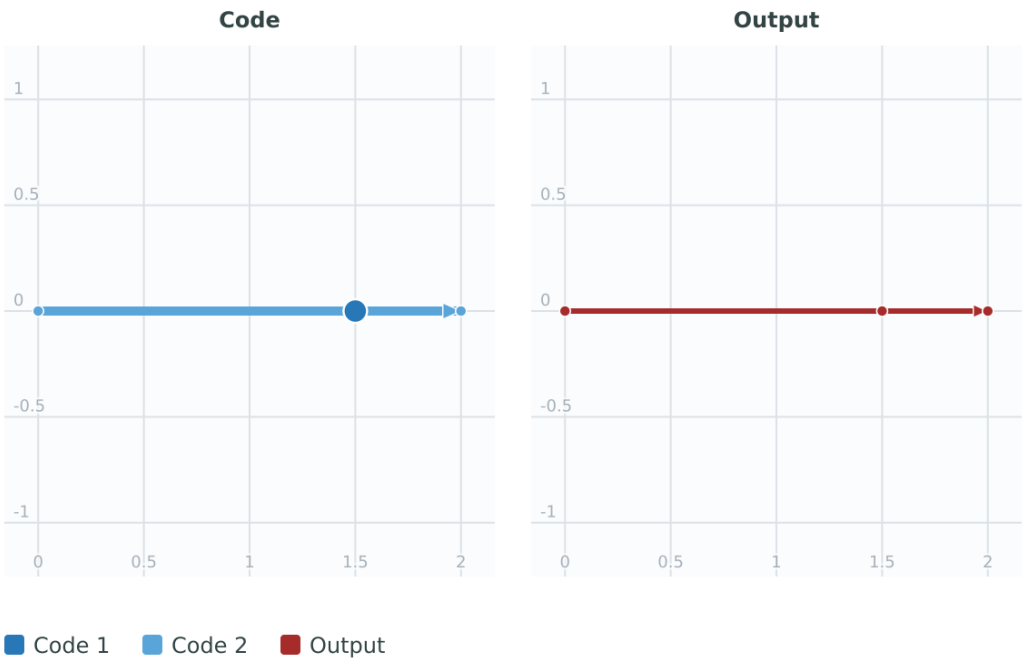
Code

```
SELECT CG_Union('POINT(.5 0)', 'LINESTRING(-1 0,1 0)');
```

Output

```
LINESTRING(-1 0,0.5 0,1 0)
```

Figure



See Also

CG_3DUnion, ST_Union

8.3.11 CG_3DUnion

CG_3DUnion — Perform 3D union using postgis_sfcgal.

Synopsis

geometry **CG_3DUnion**(geometry geom1, geometry geom2);
geometry **CG_3DUnion**(geometry set g1field);

Description

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Aggregate variant: returns a geometry that is the 3D union of a rowset of geometries. The CG_3DUnion() function is an "aggregate" function in the terminology of PostgreSQL. That means that it operates on rows of data, in the same way the SUM() and AVG() functions do and like most aggregates, it also ignores NULL geometries.

Examples

Union two overlapping tetrahedral solids.

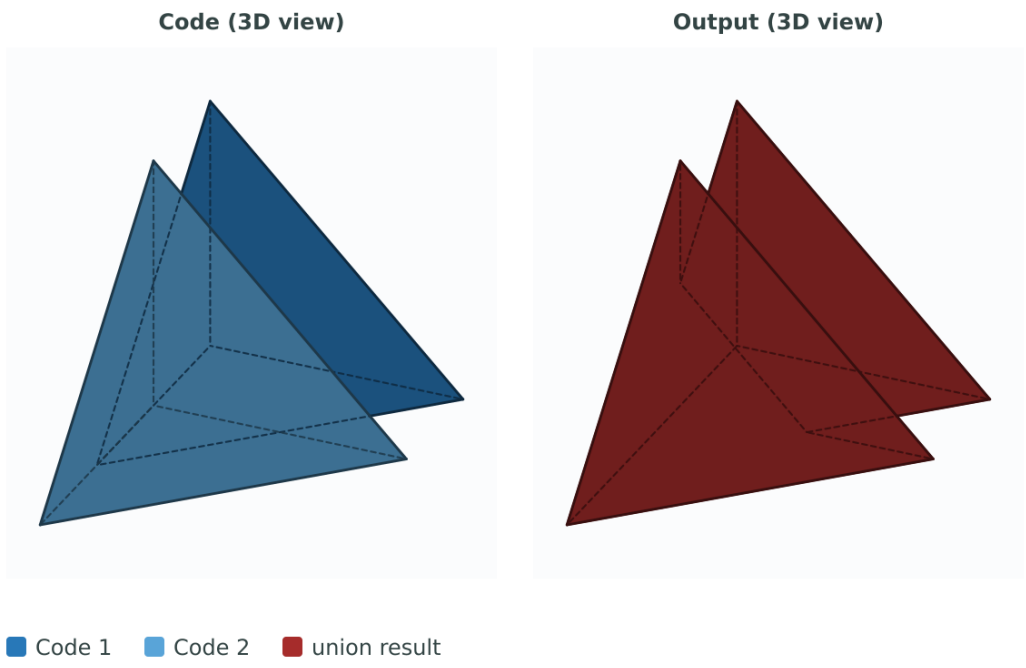
Code

```
WITH solids AS (
  SELECT CG_MakeSolid('POLYHEDRALSURFACE Z (
            ((0 0 0,4 0 0,0 4 0,0 0 0)),
            ((0 0 0,0 0 4,4 0 0,0 0 0)),
            ((0 0 0,0 4 0,0 0 4,0 0 0)),
            ((4 0 0,0 0 4,0 4 0,4 0 0)))'::geometry) AS geom1,
         CG_MakeSolid('POLYHEDRALSURFACE Z (
            ((2 0 0,6 0 0,2 4 0,2 0 0)),
            ((2 0 0,2 0 4,6 0 0,2 0 0)),
            ((2 0 0,2 4 0,2 0 4,2 0 0)),
            ((6 0 0,2 0 4,2 4 0,6 0 0)))'::geometry) AS geom2
)
SELECT CG_3DUnion(geom1, geom2) AS union_result
FROM solids;
```

Output

```
POLYHEDRALSURFACE Z (((2 0 0,0 0 0,2 2 0,2 0 0)),((2 0 2,0 0 0,2 0 0,2 0 2)),((0 0 4,0 4 0,0 0 0,0 0 4)),((2 0 2,0 4 ↩
    0,0 0 4,2 0 2)),((0 4 0,2 2 0,0 0 0,0 4 0)),((2 2 0,0 4 0,2 0 2,2 2 0)),((0 0 0,2 0 2,0 0 4,0 0 0)),((4 0 0,2 0 ↩
    0,2 2 0,4 0 0)),((4 0 0,2 0 2,2 0 0,4 0 0)),((4 0 0,2 2 0,6 0 0,4 0 0)),((2 0 2,4 0 0,2 0 4,2 0 2)),((2 0 2,2 4 ↩
    0,2 2 0,2 0 2)),((2 4 0,2 0 4,6 0 0,2 4 0)),((2 2 0,2 4 0,6 0 0,2 2 0)),((2 4 0,2 0 2,2 0 4,2 4 0)),((4 0 0,6 0 ↩
    0,2 0 4,4 0 0)))
```

Figure



See Also

CG_Extrude, ST_AsX3D, CG_3DIntersection CG_3DDifference

8.3.12 CG_AlphaShape

CG_AlphaShape — Computes an Alpha-shape enclosing a geometry

Synopsis

geometry **CG_AlphaShape**(geometry geom, float alpha, boolean allow_holes = false);

Description

Computes the Alpha-Shape of the points in a geometry. An alpha-shape is a (usually) concave polygonal geometry which contains all the vertices of the input, and whose vertices are a subset of the input vertices. An alpha-shape provides a closer fit to the shape of the input than the shape produced by the convex hull.

The "closeness of fit" is controlled by the `alpha` parameter, which can have values from 0 to infinity. Smaller alpha values produce more concave results. Alpha values greater than some data-dependent value produce the convex hull of the input.



Note
Following the CGAL implementation, the alpha value is the *square* of the radius of the disc used in the Alpha-Shape algorithm to "erode" the Delaunay Triangulation of the input points. See CGAL Alpha-Shapes for more information. This is different from the original definition of alpha-shapes, which defines alpha as the radius of the eroding disc.

The computed shape does not contain holes unless the optional `allow_holes` argument is specified as true.

This function effectively computes a concave hull of a geometry in a similar way to ST_ConcaveHull, but uses CGAL and a different algorithm.

Availability: 3.5.0 - requires SFCGAL >= 1.4.1.

 This method needs SFCGAL backend.

Examples

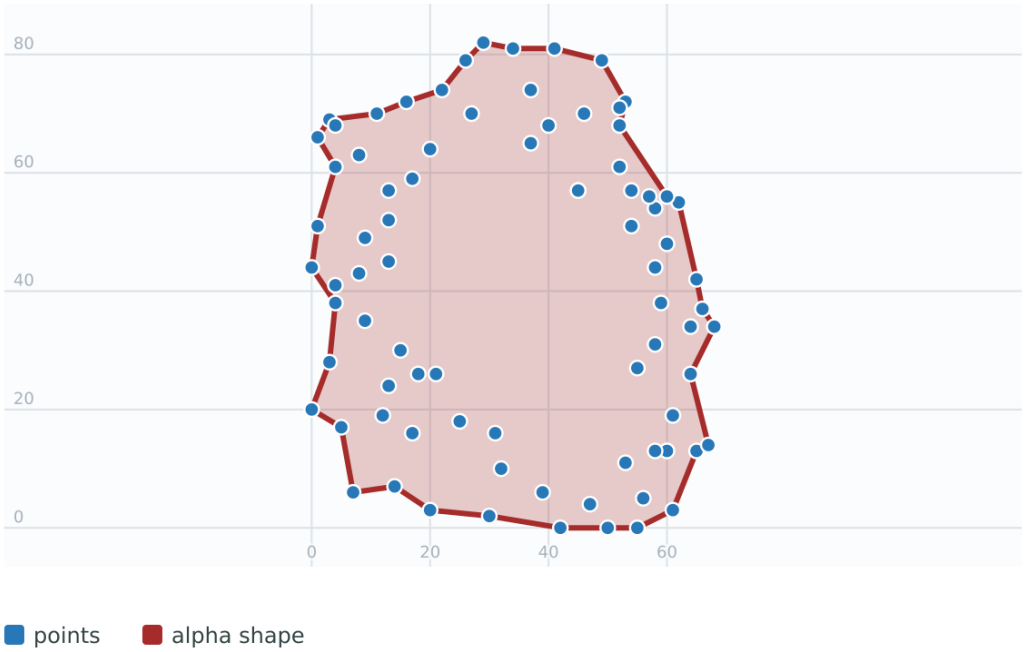
The original point cloud and the alpha-shape share one comparison frame.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),
(88 29),(24 82),(32 51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 ↩
    65),
(81 47),(88 58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 ↩
    98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 ↩
    88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 ↩
    90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 ↩
    59),(34 86),(45 90),(64 97))'::geometry AS input_points
)
SELECT input_points AS input_points,

         CG_AlphaShape(input_points, 80.2) AS alpha_shape
FROM data;
```

Figure



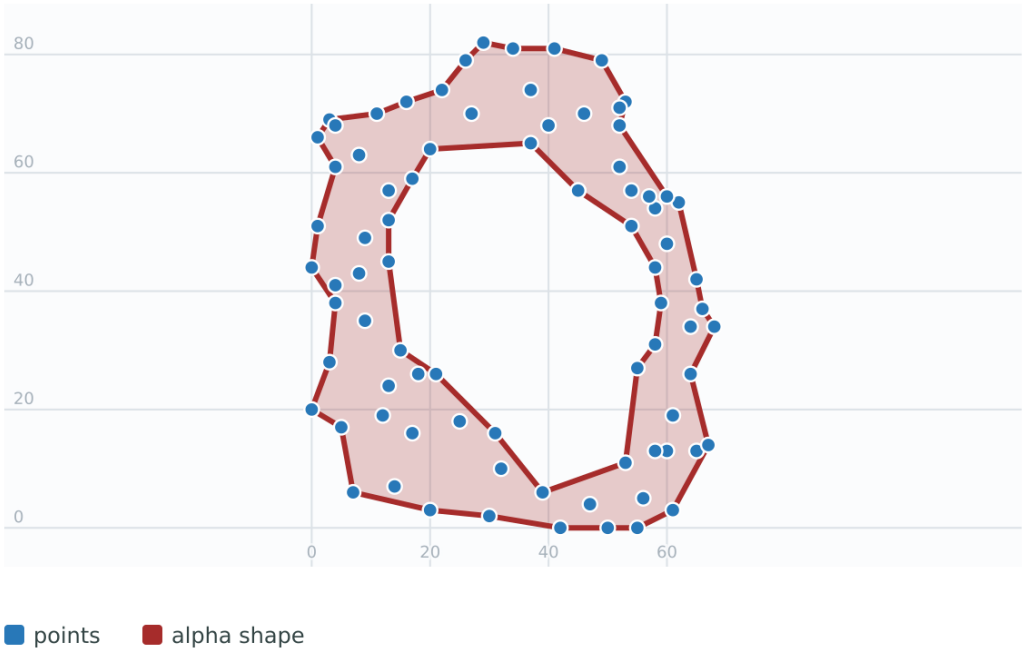
The same point cloud can produce a shape with a hole when holes are allowed.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),(88 29),(24 82),(32 51)
    ,(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 65),(81 47),(88 58)
    ,(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52
    98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39
    88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60
    90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31
    59),(34 86),(45 90),(64 97))'::geometry AS input_points
)
SELECT input_points AS input_points,

         CG_AlphaShape(input_points, 100.1, true) AS alpha_shape
FROM data;
```

Figure



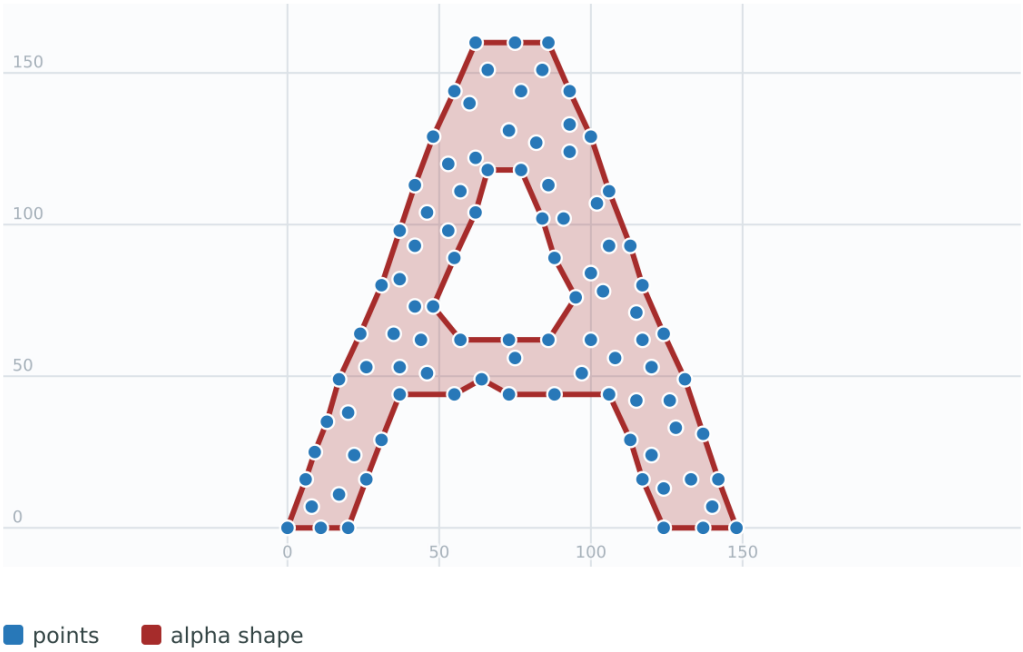
The second documented point cloud also produces a holed alpha-shape.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT ((132 64),(114 64),(99 64),(81 64),(63 64),(57 49),(52 36),(46 20),(37
    20),(26 20),(32 36),(39 55),(43 69),(50 84),(57 100),(63 118),(68 133),(74 149),(81 164)
    ,(88 180),(101 180),(112 180),(119 164),(126 149),(132 131),(139 113),(143 100),(150 84)
    ,(157 69),(163 51),(168 36),(174 20),(163 20),(150 20),(143 36),(139 49),(132 64),(99
    151),(92 138),(88 124),(81 109),(74 93),(70 82),(83 82),(99 82),(112 82),(126 82),(121
    96),(114 109),(110 122),(103 138),(99 151),(34 27),(43 31),(48 44),(46 58),(52 73),(63
    73),(61 84),(72 71),(90 69),(101 76),(123 71),(141 62),(166 27),(150 33),(159 36),(146
    44),(154 53),(152 62),(146 73),(134 76),(143 82),(141 91),(130 98),(126 104),(132 113)
    ,(128 127),(117 122),(112 133),(119 144),(108 147),(119 153),(110 171),(103 164),(92
    171),(86 160),(88 142),(79 140),(72 124),(83 131),(79 118),(68 113),(63 102),(68 93),(35
     45))'::geometry AS input_points
)
SELECT input_points AS input_points,

         CG_AlphaShape(input_points, 102.2, true) AS alpha_shape
FROM data;
```

Figure



See Also

ST_ConcaveHull, CG_OptimalAlphaShape

8.3.13 CG_ApproxConvexPartition

CG_ApproxConvexPartition — Computes approximal convex partition of the polygon geometry

Synopsis

geometry **CG_ApproxConvexPartition**(geometry geom);

Description

Computes approximal convex partition of the polygon geometry (using a triangulation).



Note
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. CG_ApproxConvexPartition and CG_GreeneApproxConvexPartition functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; CG_ApproxConvexPartition uses a triangulation and CG_GreeneApproxConvexPartition a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. Though the triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0


This method needs SFCGAL backend.

Examples

Approximate convex partition of the same polygon used by the other partition examples.

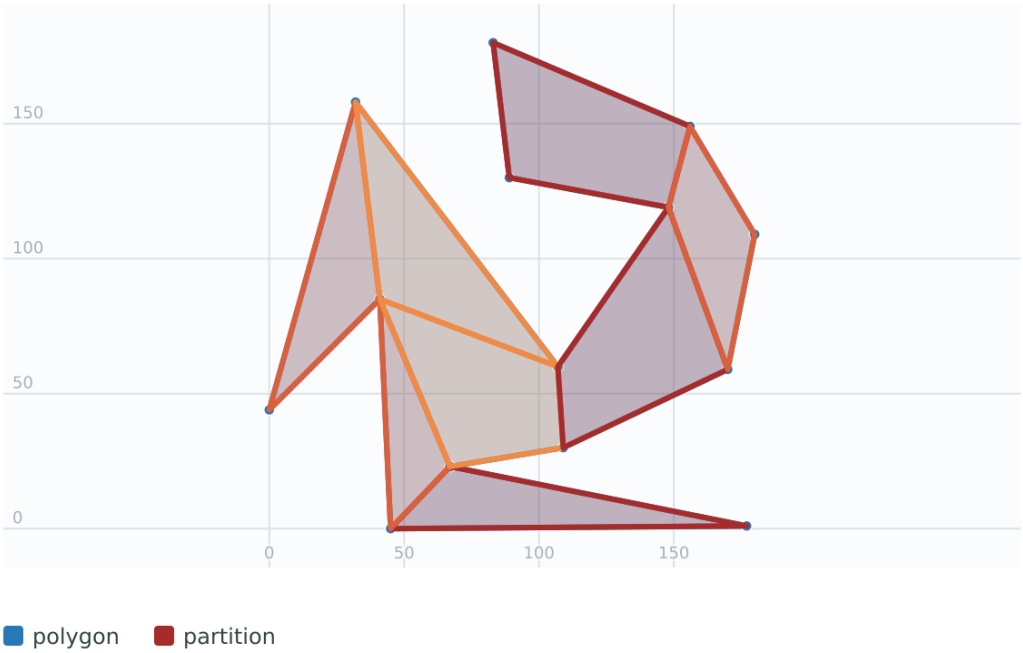
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_ApproxConvexPartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((156 150,83 181,89 131,148 120,156 150)), POLYGON((32 159,0 45,41 86,32 159)), POLYGON ↩
    ((107 61,32 159,41 86,107 61)), POLYGON((45 1,177 2,67 24,45 1)), POLYGON((41 86,45 1,67 24,41 86)), POLYGON((107 ↩
    61,41 86,67 24,109 31,107 61)), POLYGON((148 120,107 61,109 31,170 60,148 120)), POLYGON((156 150,148 120,170 ↩
    60,180 110,156 150)))
```

Figure



See Also

CG_YMonotonePartition, CG_GreeneApproxConvexPartition, CG_OptimalConvexPartition

8.3.14 CG_ApproximateMedialAxis

CG_ApproximateMedialAxis — Compute the approximate medial axis of an areal geometry.

Synopsis

geometry **CG_ApproximateMedialAxis**(geometry geom);
geometry **CG_ApproximateMedialAxis**(geometry geom, boolean projected);

Description

Return an approximate medial axis for the areal input based on its straight skeleton. Uses an SFCGAL specific API when built against a capable version (1.2.0+). Otherwise the function is just a wrapper around CG_StraightSkeleton (slower case).

When `projected` is `true`, free endpoints of the medial axis are extended to reach the polygon boundary (projected medial axis). Requires SFCGAL 2.3.0+. When built against an older SFCGAL version, a notice is emitted and the non-projected result is returned instead.

Availability: 3.5.0

Availability: 3.7.0 - `projected` parameter. Requires SFCGAL >= 2.3.0 for projected result; falls back to non-projected with a notice on older versions.



Note
This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

 This method needs SFCGAL backend.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

A polygon and its approximate medial axis.

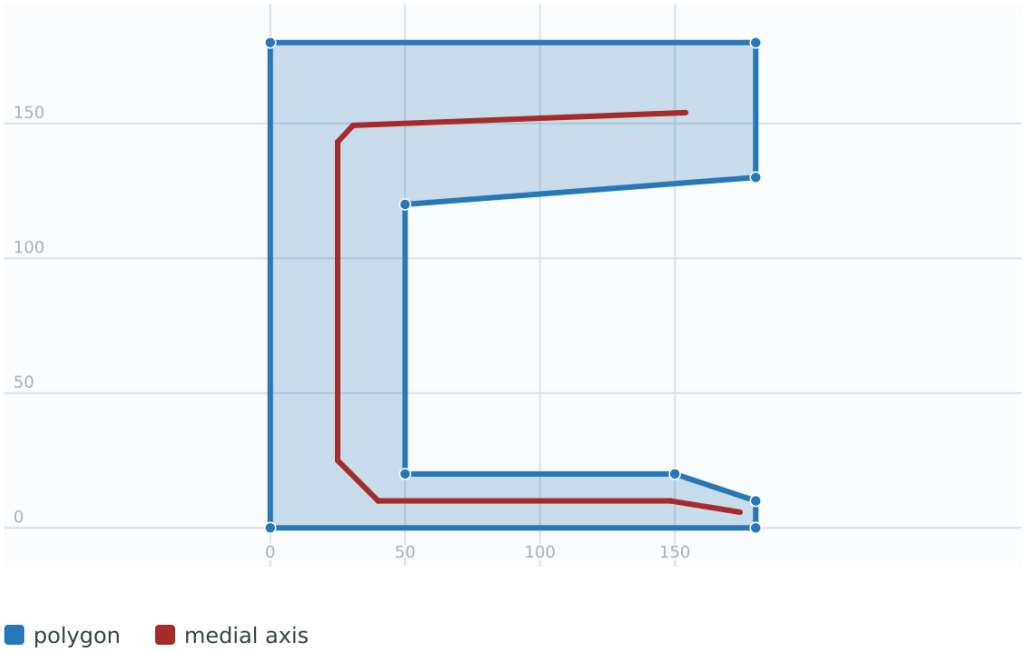
Code

```
WITH data AS (
  SELECT 'POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))' ↩
    ::geometry AS polygon
)
SELECT polygon AS input_polygon,
       CG_ApproximateMedialAxis(polygon) AS medial_axis
FROM data;
```

Output

```
POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190)) | MULTILINESTRING((184.189 ↩
    15.811,158.377 20),(50 20,158.377 20),(50 20,35 35),(35 153.151,35 35),(35 153.151,40.697 159.303),(164.04 ↩
    164.04,40.697 159.303))
```

Figure



This example computes the projected medial axis with free endpoints extended to the polygon boundary.

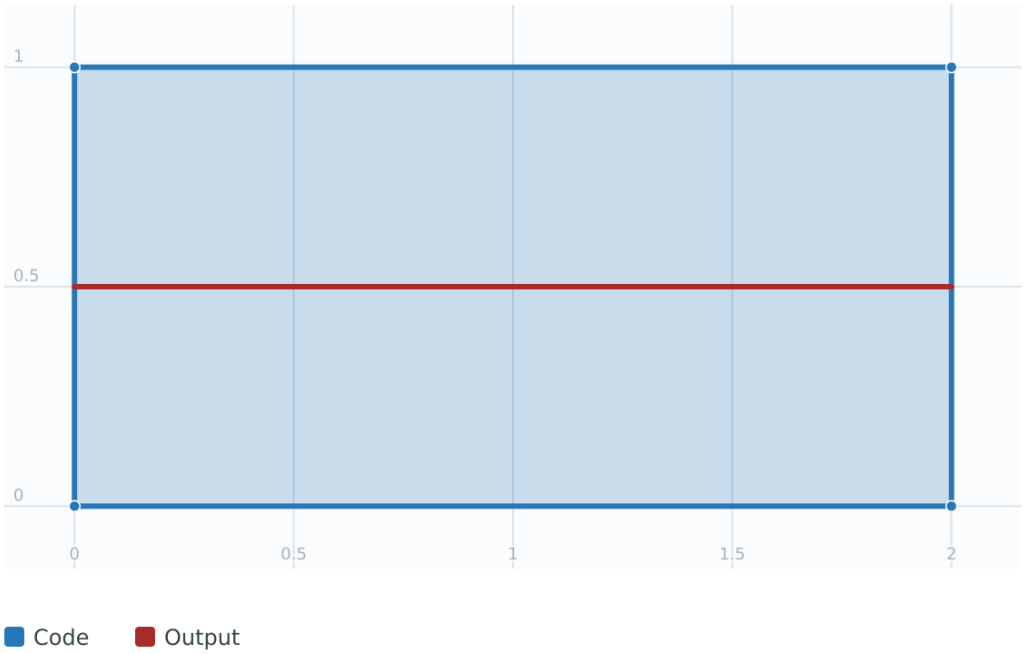
Code

```
SELECT CG_ApproximateMedialAxis('POLYGON((0 0,2 0,2 1,0 1,0 0))', true);
```

Output

```
MULTILINESTRING((0 0.5,0.5 0.5,1.5 0.5,2 0.5))
```

Figure



See Also

CG_StraightSkeleton, CG_StraightSkeletonPartition

8.3.15 CG_GenerateFlatRoof

CG_GenerateFlatRoof — Generate a flat (box) roof from a footprint polygon.

Synopsis

geometry **CG_GenerateFlatRoof**(geometry geom, float8 height);

Description

Generate a flat (box-shaped) roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. The roof height above the ground plane is given by `height` (default 3.0).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

 This method needs SFCGAL backend.

 This function supports Polyhedral surfaces.

Examples

Generate a flat roof two units above a rectangular footprint.

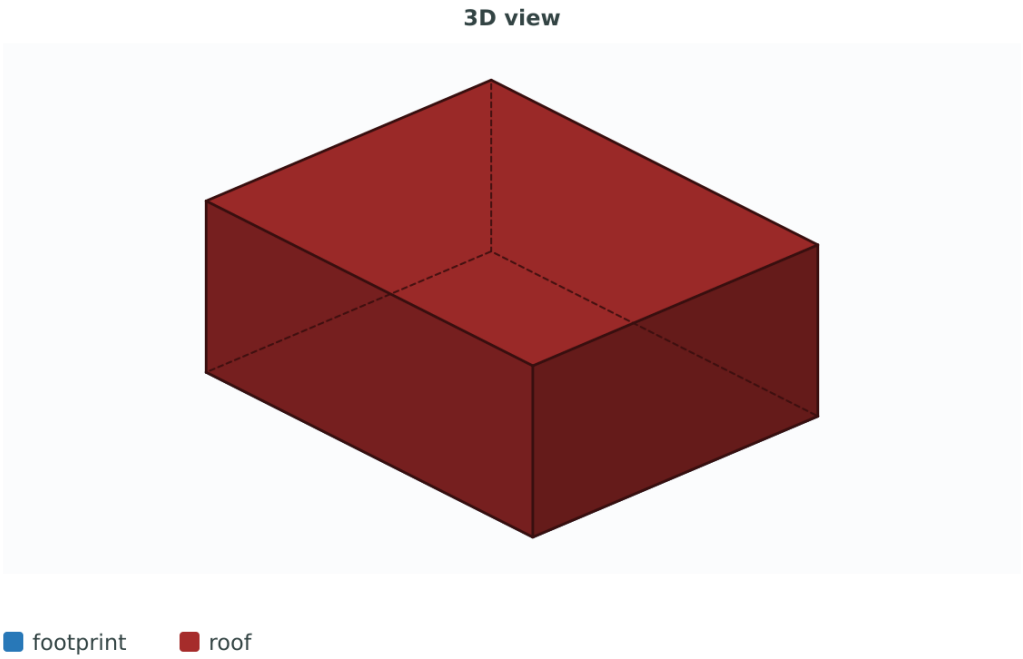
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS footprint
)
SELECT CG_GenerateFlatRoof(footprint, 2.0) AS roof
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((0 0 0,0 4 0,5 4 0,5 0 0,0 0 0)),((0 0 2,5 0 2,5 4 2,0 4 2,0 0 2)),((0 0 0,0 0 2,0 4 2,0 4 0,0 0 ↩
    0)),((0 4 0,0 4 2,5 4 2,5 4 0,0 4 0)),((5 4 0,5 4 2,5 0 2,5 0 0,5 4 0)),((5 0 0,5 0 2,0 0 2,0 0 0,5 0 0)))
```

Figure



See Also

CG_GenerateHippedRoof, CG_GenerateGableRoof, CG_GenerateSkillionRoof, CG_GenerateRoof

8.3.16 CG_GenerateHippedRoof

CG_GenerateHippedRoof — Generate a hipped roof from a footprint polygon.

Synopsis

geometry **CG_GenerateHippedRoof**(geometry geom, float8 height);

Description

Generate a hipped roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. All sides slope upward to meet at a ridge or apex. The roof height is given by `height` (default 3.0).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

Examples

The footprint and hipped roof are shown side-by-side so the generated 3D faces do not hide behind the input footprint.

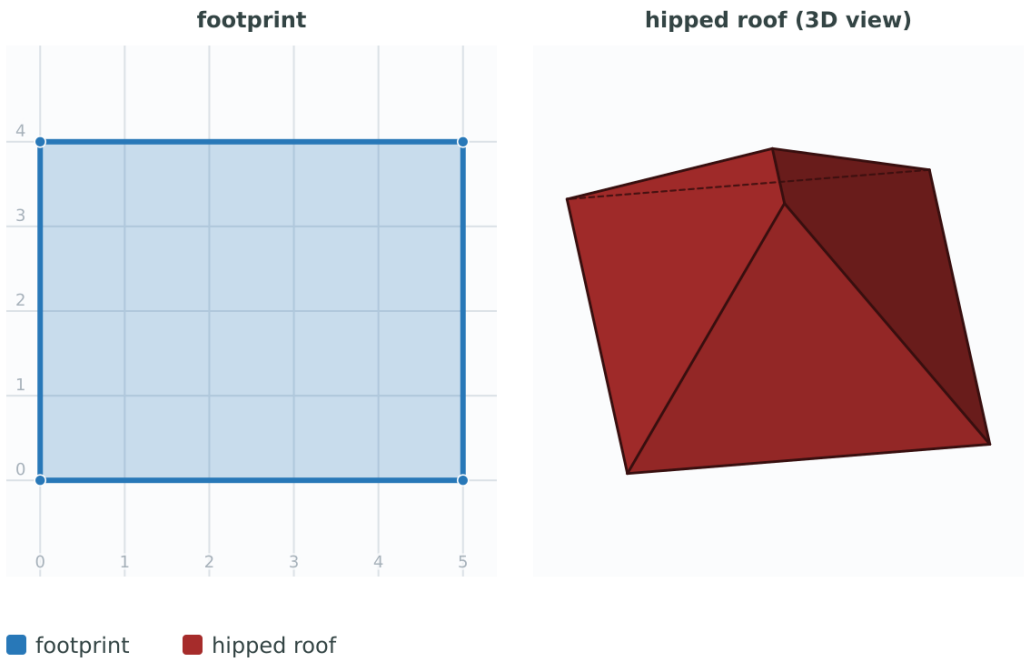
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS input_footprint
)
SELECT input_footprint AS input_footprint,
       CG_GenerateHippedRoof(input_footprint, 2.0) AS hipped_roof
FROM data;
```

Output

```
POLYGON((0 0,5 0,5 4,0 4,0 0)) | POLYHEDRALSURFACE Z (((0 4 0,5 4 0,5 0 0,0 0 0,0 4 0)),((0 4 0,0 0 0,2 2 2,0 4 0)),((0 ↩
      0 0,5 0 0,3 2 2,2 2 2,0 0 0)),((5 0 0,5 4 0,3 2 2,5 0 0)),((5 4 0,0 4 0,2 2 2,3 2 2,5 4 0)))
```

Figure



See Also

CG_GenerateFlatRoof, CG_GenerateGableRoof, CG_GenerateSkillionRoof, CG_GenerateRoof

8.3.17 CG_GenerateGableRoof

CG_GenerateGableRoof — Generate a gable roof from a footprint polygon.

Synopsis

geometry **CG_GenerateGableRoof**(geometry geom, float8 height, float8 slope_angle);

Description

Generate a gable roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. Two opposite sides are vertical gable ends; the other two sides slope to a horizontal ridge. `height` sets the ridge height above the ground (default 3.0) and `slope_angle` controls the pitch in degrees (default 30.0). The SRID of the input geometry is preserved.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

Examples

Generate a gable roof using a height of 2 and a 30 degree slope.

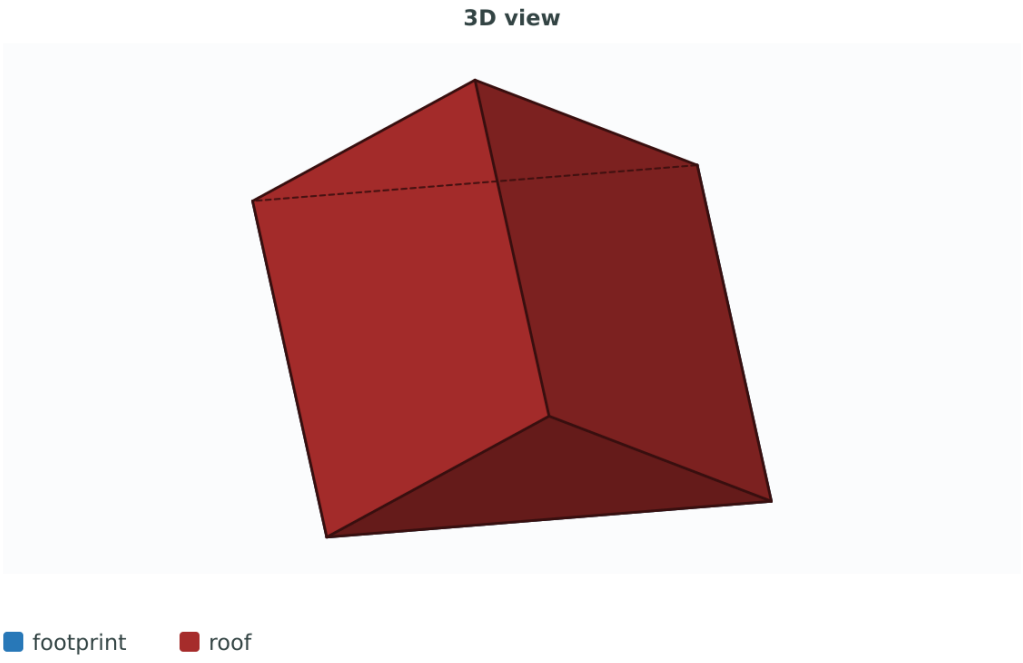
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS footprint
)
SELECT CG_GenerateGableRoof(footprint, 2.0, 30.0) AS roof
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((5 4 0,5 0 0,0 0 0,0 4 0,5 4 0)),((0 4 0,0 0 0,0 2 1.155,0 4 0)),((0 0 0,5 0 0,5 2 1.155,0 2 ↩
    1.155,0 0 0)),((5 0 0,5 4 0,5 2 1.155,5 0 0)),((5 4 0,0 4 0,0 2 1.155,5 2 1.155,5 4 0)))
```

Figure



See Also

CG_GenerateFlatRoof, CG_GenerateHippedRoof, CG_GenerateSkillionRoof, CG_GenerateRoof

8.3.18 CG_GenerateSkillionRoof

CG_GenerateSkillionRoof — Generate a skillion (single-slope) roof from a footprint polygon.

Synopsis

geometry **CG_GenerateSkillionRoof**(geometry geom, float8 height, float8 slope_angle, integer primary_edge_index);

Description

Generate a skillion (single-slope, shed-style) roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. The entire roof surface slopes in one direction. `height` sets the maximum ridge height (default 3.0), `slope_angle` sets the pitch in degrees (default 30.0), and `primary_edge_index` selects which polygon edge defines the high side (0-based, default 0).

When the combination of `height` and `slope_angle` would cause a side wall to vanish (the slope reaches the far edge before the full height is achieved), that face is omitted from the result. This means the number of output patches can vary depending on the parameters — a lower height relative to the footprint depth produces more faces than a taller, steeper configuration.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

 This method needs SFCGAL backend.

 This function supports Polyhedral surfaces.

Examples

Height 2.0 produces 6 faces (all walls present); height 4.0 produces 5 faces (one side wall is filtered out because the slope reaches the far edge before the full height).

Code

```
SELECT
  ST_NumPatches(CG_GenerateSkillionRoof(
    'POLYGON((0 0,5 0,5 4,0 4,0 0))', 2.0)) AS "height 2",
  ST_NumPatches(CG_GenerateSkillionRoof(
    'POLYGON((0 0,5 0,5 4,0 4,0 0))', 4.0)) AS "height 4";
```

Output

```
6 | 5
```

The height 2.0 result is shown with its footprint.

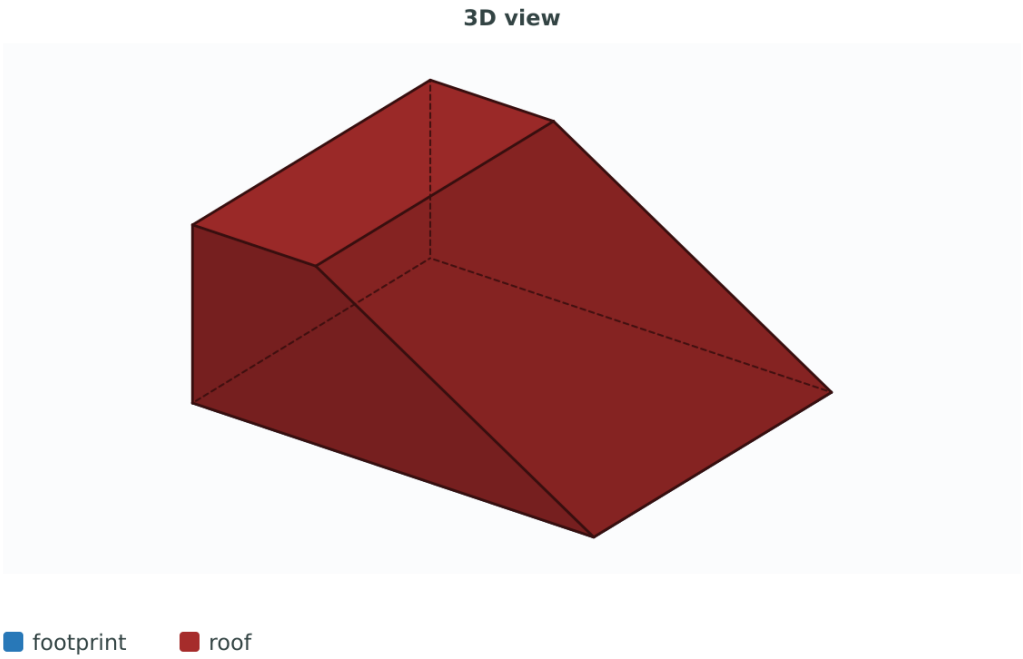
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS footprint
)
SELECT CG_GenerateSkillionRoof(footprint, 2.0) AS roof
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((0 4 0,5 4 0,5 0 0,0 0 0,0 4 0)),((5 0 2,5 4 2,3.464 4 2,3.464 0 2,5 0 2)),((0 4 0,0 0 0,3.464 0 ↩
    2,3.464 4 2,0 4 0)),((0 0 0,5 0 0,5 0 2,3.464 0 2,0 0 0)),((5 0 0,5 4 0,5 4 2,5 0 2,5 0 0)),((5 4 0,0 4 0,3.464 4 ↩
    2,5 4 2,5 4 0)))
```

Figure



See Also

CG_GenerateFlatRoof, CG_GenerateHippedRoof, CG_GenerateGableRoof, CG_GenerateRoof

8.3.19 CG_GenerateRoof

CG_GenerateRoof — Generate a roof of the requested type from a footprint polygon.

Synopsis

geometry **CG_GenerateRoof**(geometry geom, text roof_type, float8 height, float8 slope_angle, integer primary_edge_index);

Description

Generate a roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. The `roof_type` parameter selects the roof style (default `'HIPPED'`):

- `FLAT` — flat box roof.
- `HIPPED` — hipped roof (all sides slope to apex).
- `GABLE` — gable roof (two sloping sides, two vertical ends).
- `SKILLION` — single-slope (shed) roof.

`height` sets the roof height (default 3.0), `slope_angle` sets the pitch in degrees (default 30.0), and `primary_edge_index` selects the reference edge for skillion roofs (default 0).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

Examples

The generic dispatcher returns the same 3D roofs as the specialized helpers for the same footprint.

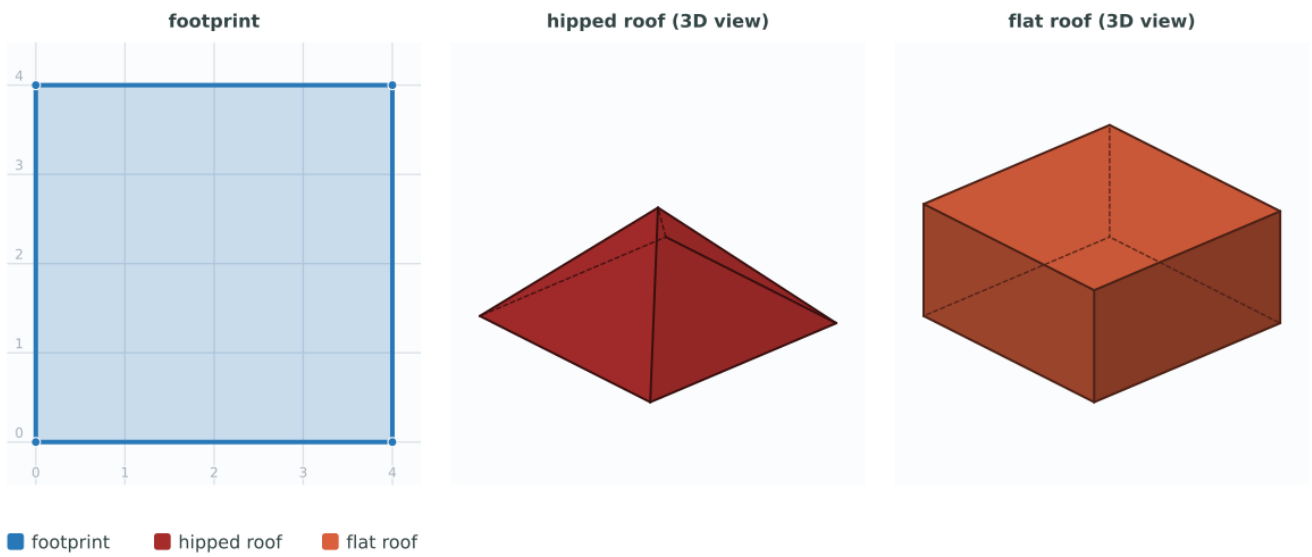
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,4 0,4 4,0 4,0 0))'::geometry AS input_footprint
)
SELECT input_footprint AS input_footprint,
       CG_GenerateRoof(input_footprint, 'HIPPED', 2.0, 30.0, 0) AS hipped_roof,
       CG_GenerateRoof(input_footprint, 'FLAT', 2.0, 30.0, 0) AS flat_roof
FROM data;
```

Output

```
POLYGON((0 0,4 0,4 4,0 4,0 0)) | POLYHEDRALSURFACE Z (((0 4 0,4 4 0,4 0 0,0 0 0,0 4 0)),((0 4 0,0 0 0,2 2 2,0 4 0)),((0 ↩
      0 0,4 0 0,2 2 2,0 0 0)),((4 0 0,4 4 0,2 2 2,4 0 0)),((4 4 0,0 4 0,2 2 2,4 4 0))) | POLYHEDRALSURFACE Z (((0 0 0,0 ↩
      4 0,4 4 0,4 0 0,0 0 0)),((0 0 2,4 0 2,4 4 2,0 4 2,0 0 2)),((0 0 0,0 0 2,0 4 2,0 4 0,0 0 0)),((0 4 0,0 4 2,4 4 2,4 ↩
      4 0,0 4 0)),((4 4 0,4 4 2,4 0 2,4 0 0,4 4 0)),((4 0 0,4 0 2,0 0 2,0 0 0,4 0 0)))
```

Figure



See Also

CG_GenerateFlatRoof, CG_GenerateHippedRoof, CG_GenerateGableRoof, CG_GenerateSkillionRoof

8.3.20 CG_PolygonRepair

CG_PolygonRepair — Repair an invalid polygon or multipolygon.

Synopsis

geometry **CG_PolygonRepair**(geometry geom, text rule);

Description

Repair an invalid polygon or multipolygon using CGAL's 2D Polygon Repair algorithm. Returns a `MultiPolygon`.

The `rule` parameter controls the repair strategy:

- `EVEN_ODD` (default) — areas covered an odd number of times are kept. Available with CGAL 6.0+.
- `NON_ZERO` — areas with non-zero winding number are kept. Requires CGAL 6.1+.
- `UNION` — union of all input polygons. Requires CGAL 6.1+.
- `INTERSECTION` — intersection of all input polygons. Requires CGAL 6.1+.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0 and CGAL >= 6.0.


This method needs SFCGAL backend.


This function supports Polyhedral surfaces.

Examples

Repair a self-intersecting bowtie polygon with the default `EVEN_ODD` rule.

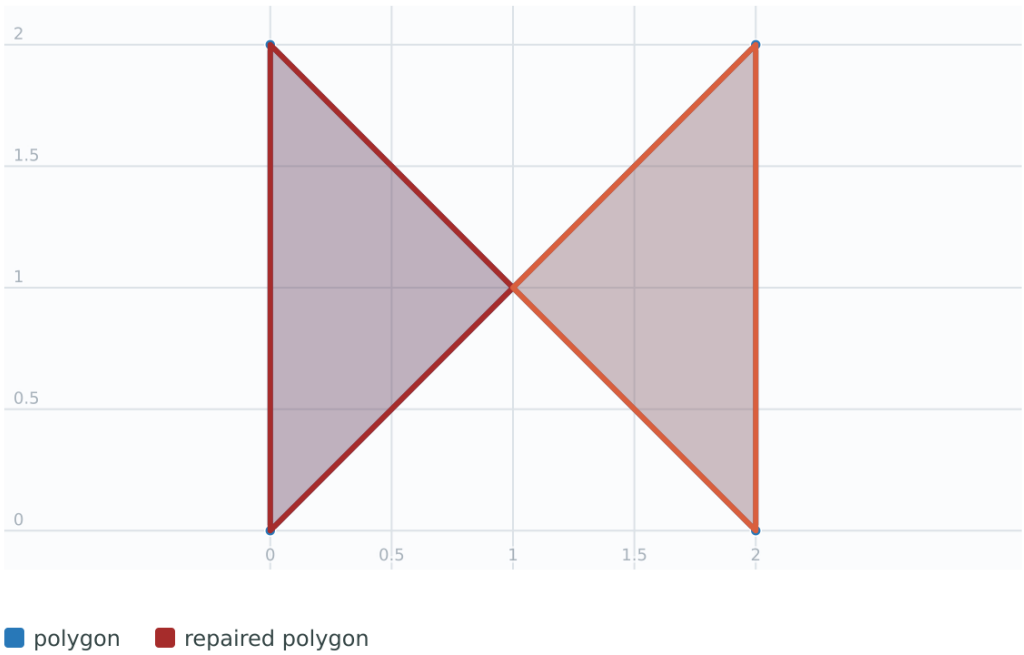
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,2 2,2 0,0 2,0 0))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_PolygonRepair(input_polygon) AS repaired_polygon
FROM data;
```

Output

```
POLYGON((0 0,2 2,2 0,0 2,0 0)) | MULTIPOLYGON(((0 0,1 1,0 2,0 0)),((1 1,2 0,2 2,1 1)))
```

Figure



With the `UNION` rule, all winding area is kept. This rule requires CGAL 6.1 or later.

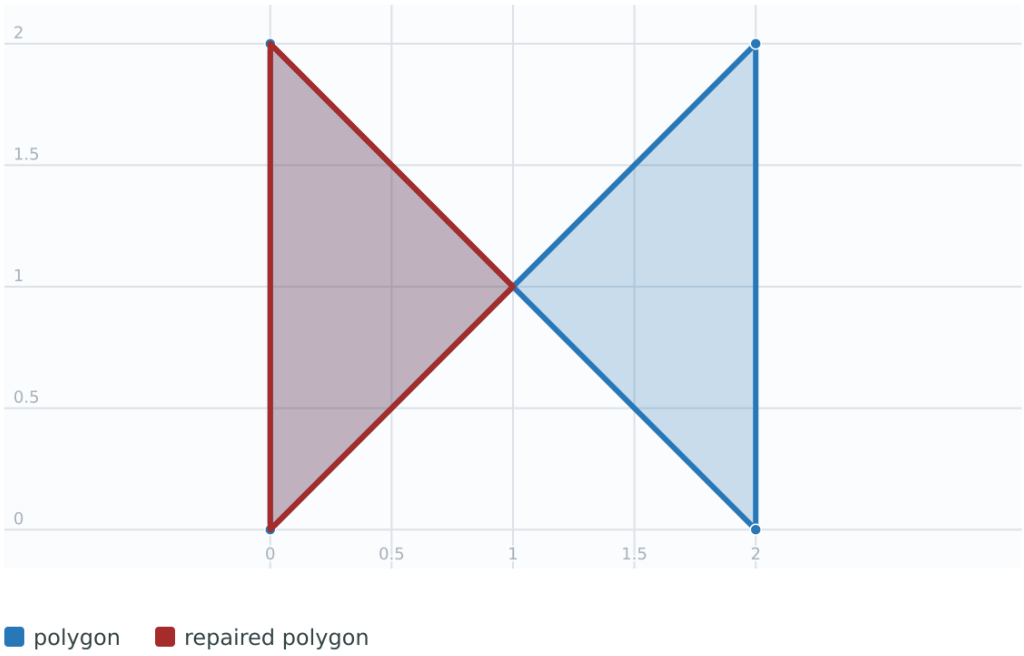
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,2 2,2 0,0 2,0 0))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_PolygonRepair(input_polygon, 'UNION') AS repaired_polygon
FROM data;
```

Output

```
POLYGON((0 0,2 2,2 0,0 2,0 0)) | MULTIPOLYGON(((0 0,1 1,0 2,0 0)))
```

Figure



See Also

ST_IsValid, ST_MakeValid

8.3.21 CG_ConstrainedDelaunayTriangles

CG_ConstrainedDelaunayTriangles — Return a constrained Delaunay triangulation around the given input geometry.

Synopsis

geometry **CG_ConstrainedDelaunayTriangles**(geometry g1);

Description

Return a Constrained Delaunay triangulation around the vertices of the input geometry. Output is a TIN.

This method needs SFCGAL backend.

Availability: 3.0.0

This function supports 3d and will not drop the z-index.

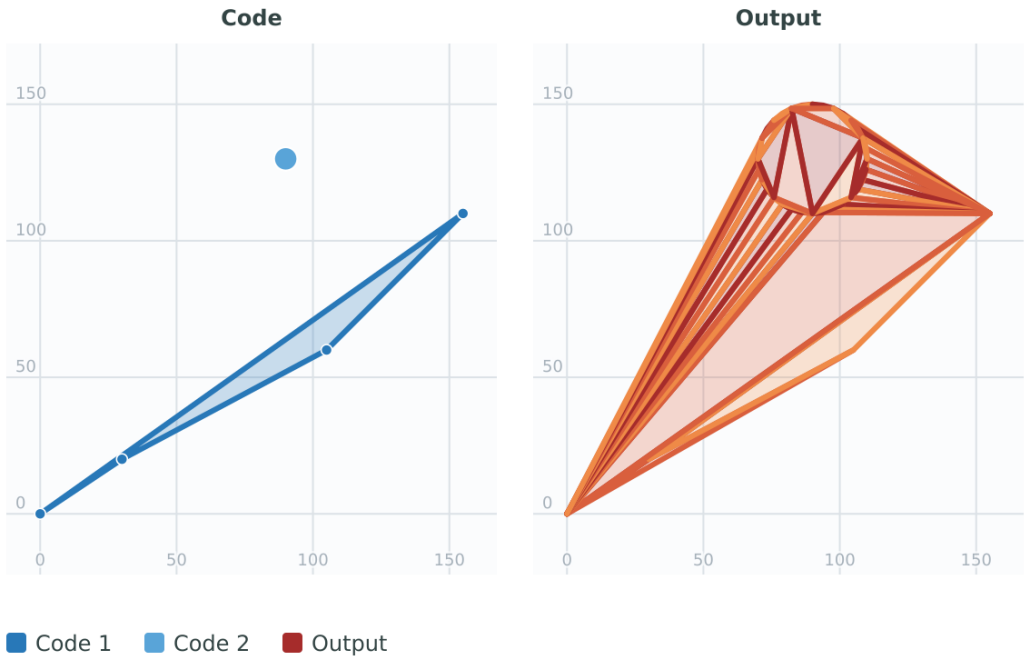
Examples

Constrained Delaunay triangulation of the polygon and buffered point.

Code

```
SELECT
  CG_ConstrainedDelaunayTriangles(
    ST_Union(
      'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry,
      ST_Buffer('POINT(110 170)'::geometry, 20)
    )
  );
```

Figure

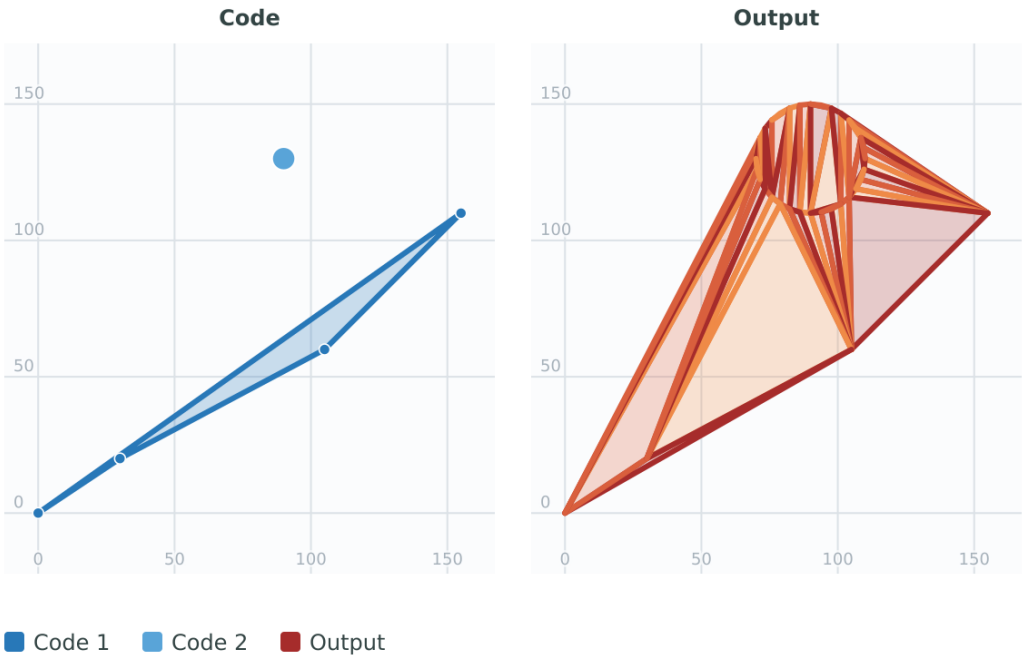


ST_DelaunayTriangles applied to the same input. Triangle edges cross polygon boundaries.

Code

```
SELECT
  ST_DelaunayTriangles(
    ST_Union(
      'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry,
      ST_Buffer('POINT(110 170)'::geometry, 20)
    )
  );
```

Figure



See Also

ST_DelaunayTriangles, ST_TriangulatePolygon, CG_Tesselate, ST_ConcaveHull, ST_Dump

8.3.22 CG_Extrude

CG_Extrude — Extrude a surface to a related volume

Synopsis

geometry **CG_Extrude**(geometry geom, float x, float y, float z);

Description

Extrudes the input along the vector (`x`, `y`, `z`) and connects it to its translated copy. Extrusion can raise the dimensionality of the result, for example from a surface to a volume.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

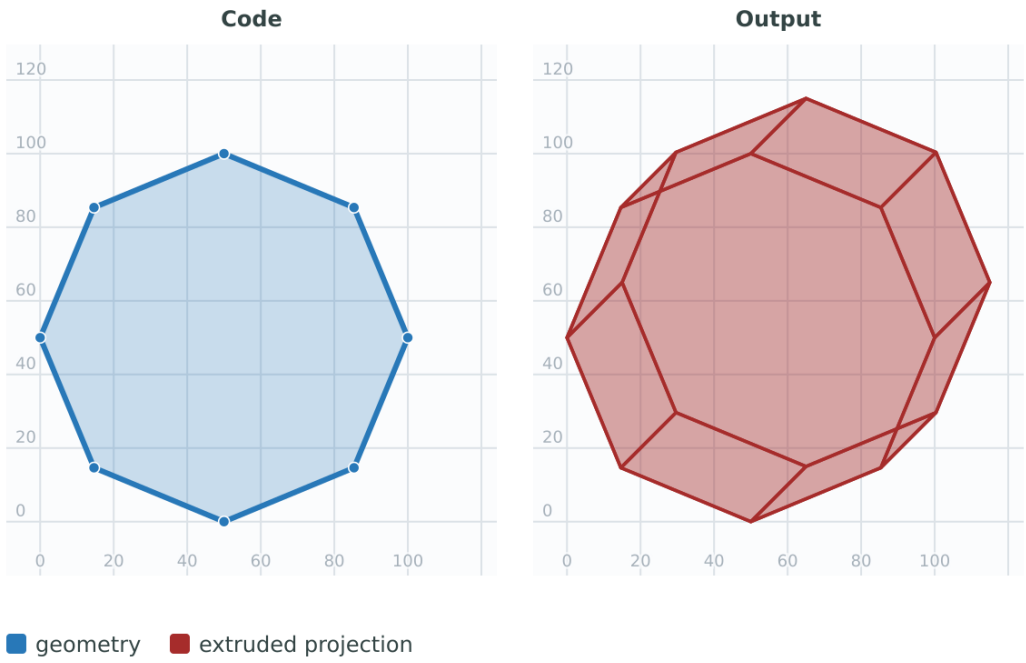
The figures use ST_Affine to project Z diagonally into XY after extrusion. The returned geometry from `CG_Extrude` remains a 3D PolyhedralSurface.

Extruding an octagonal Polygon 30 units along Z.

Code

```
WITH data AS (
  SELECT ST_Buffer('POINT(100 90)'::geometry, 50, 'quad_segs=2') AS geom
), projected AS (
  SELECT ST_Force2D(ST_Affine(geom,
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS input_geom,
         ST_Force2D(ST_Affine(CG_Extrude(geom, 0, 0, 30),
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS output_geom
  FROM data
)
SELECT input_geom AS input_geometry,
       output_geom AS extruded_projection
FROM projected;
```

Figure



Extruding a LineString 10 units along Z.

Code

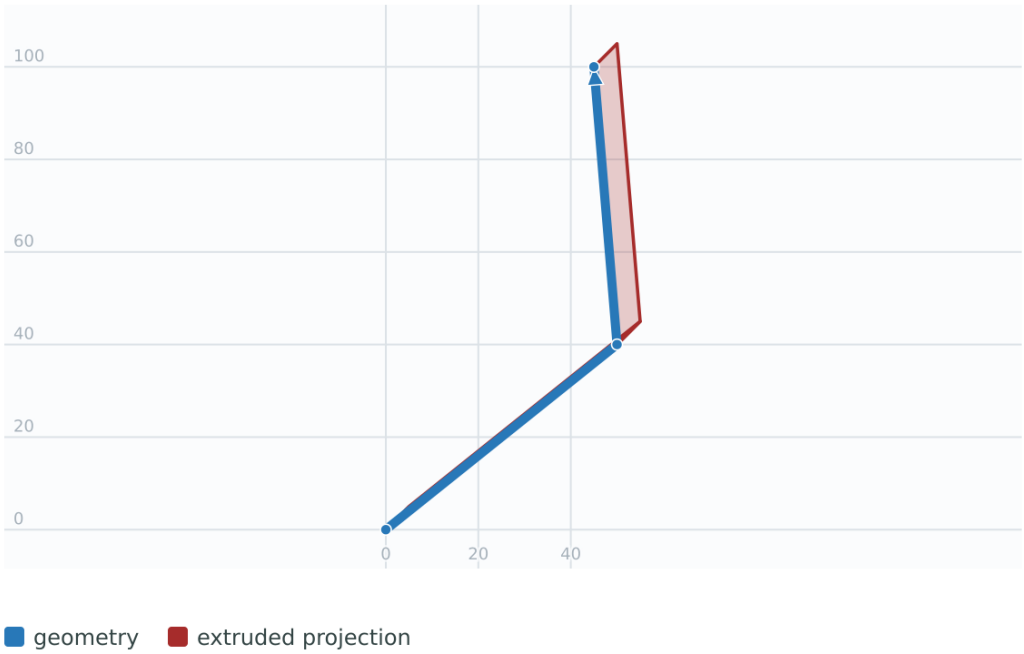
```
WITH data AS (
  SELECT 'LINESTRING(50 50,100 90,95 150)'::geometry AS geom
), projected AS (
  SELECT ST_Force2D(ST_Affine(geom,
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS input_geom,
         ST_Force2D(ST_Affine(CG_Extrude(geom, 0, 0, 10),
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS output_geom
  FROM data
)
SELECT input_geom AS input_geometry,
       output_geom AS extruded_projection
```

```
FROM projected;
```

Output

```
LINESTRING(50 50,100 90,95 150) | POLYHEDRALSURFACE(((50 50,100 90,105 95,55 55,50 50)),((100 90,95 150,100 155,105 ↩
    95,100 90)))
```

Figure



See Also

ST_AsX3D, CG_ExtrudeStraightSkeleton

8.3.23 CG_ExtrudeStraightSkeleton

CG_ExtrudeStraightSkeleton — Straight Skeleton Extrusion

Synopsis

geometry **CG_ExtrudeStraightSkeleton**(geometry geom, float roof_height, float body_height = 0);

Description

Computes an extrusion with a maximal height of the polygon geometry.



Note

Perhaps the first (historically) use-case of straight skeletons: given a polygonal roof, the straight skeleton directly gives the layout of each tent. If each skeleton edge is lifted from the plane a height equal to its offset distance, the resulting roof is "correct" in that water will always fall down to the contour edges (the roof's border), regardless of where it falls on the roof. The function computes this extrusion aka "roof" on a polygon. If the argument body_height > 0, so the polygon is extruded like with CG_Extrude(polygon, 0, 0, body_height). The result is an union of these polyhedralsurfaces.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

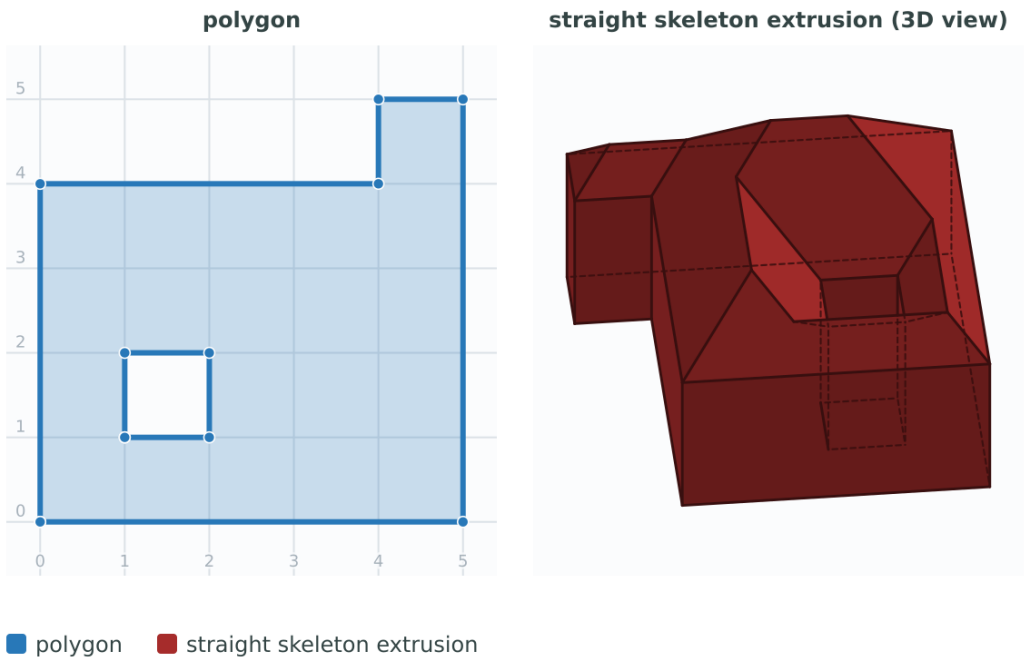
Examples

Extrude a polygon with a hole into a body and straight-skeleton roof.

Code

```
WITH data AS (
  SELECT 'POLYGON ((0 0,5 0,5 5,4 5,4 4,0 4,0 0),(1 1,1 2,2 2,2 1,1 1))'::geometry AS ↩
    input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_ExtrudeStraightSkeleton(input_polygon, 3.0, 2.0) AS straight_skeleton_extrusion
FROM data;
```

Figure



See Also

CG_Extrude, CG_ExtrudeStraightSkeleton, CG_StraightSkeleton, CG_StraightSkeletonPartition

8.3.24 CG_GreeneApproxConvexPartition

CG_GreeneApproxConvexPartition — Computes approximal convex partition of the polygon geometry

Synopsis

geometry **CG_GreeneApproxConvexPartition**(geometry geom);

Description

Computes approximal monotone convex partition of the polygon geometry.



Note

A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. CG_ApproxConvexPartition and CG_GreeneApproxConvexPartition functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; CG_ApproxConvexPartition uses a triangulation and CG_GreeneApproxConvexPartition a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. Though the triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

 This method needs SFCGAL backend.

Examples

Greene's monotone-based approximation uses the same polygon and stays directly comparable in one panel.

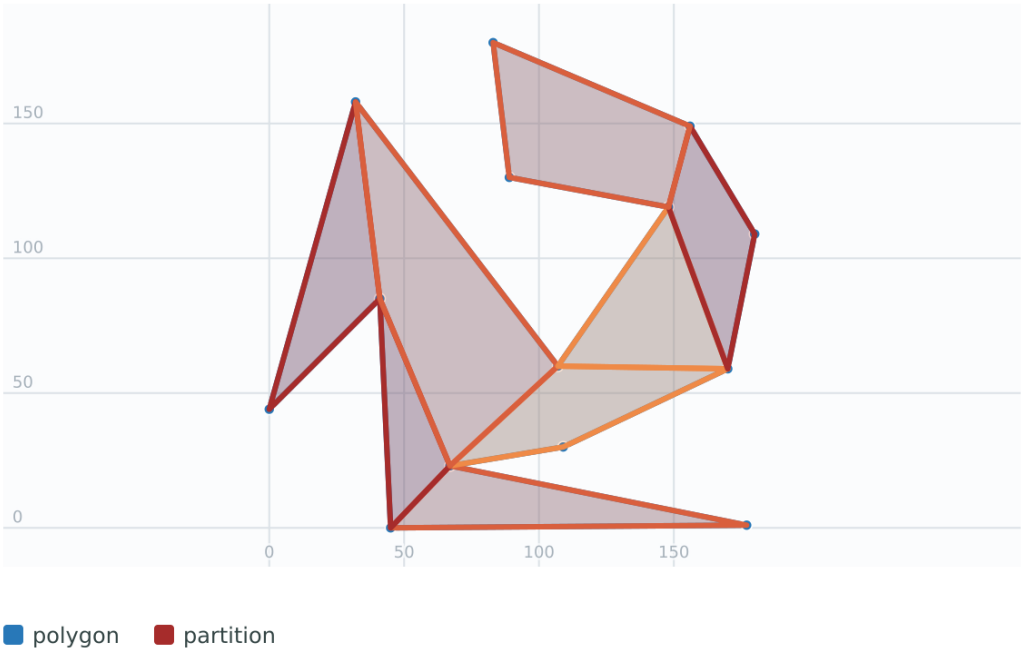
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67  ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_GreeneApproxConvexPartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((32 159,0 45,41 86,32 159)), POLYGON((45 1,177 2,67 24,45 1)), POLYGON((67 24,109 ↩
    31,170 60,107 61,67 24)), POLYGON((41 86,45 1,67 24,41 86)), POLYGON((107 61,32 159,41 86,67 24,107 61)), POLYGON ↩
    ((148 120,107 61,170 60,148 120)), POLYGON((148 120,170 60,180 110,156 150,148 120)), POLYGON((156 150,83 181,89 ↩
    131,148 120,156 150)))
```

Figure



See Also

CG_YMonotonePartition, CG_ApproxConvexPartition, CG_OptimalConvexPartition

8.3.25 CG_MinkowskiSum

CG_MinkowskiSum — Performs Minkowski sum

Synopsis

geometry **CG_MinkowskiSum**(geometry geom1, geometry geom2);

Description

This function performs a 2D minkowski sum of a point, line or polygon with a polygon.

A minkowski sum of two geometries A and B is the set of all points that are the sum of any point in A and B. Minkowski sums are often used in motion planning and computer-aided design. More details on Wikipedia Minkowski addition.

The first parameter can be any 2D geometry (point, linestring, polygon). If a 3D geometry is passed, it will be converted to 2D by forcing Z to 0, leading to possible cases of invalidity. The second parameter must be a 2D polygon.

Implementation utilizes CGAL 2D Minkowskisum.

Availability: 3.5.0

This method needs SFCGAL backend.

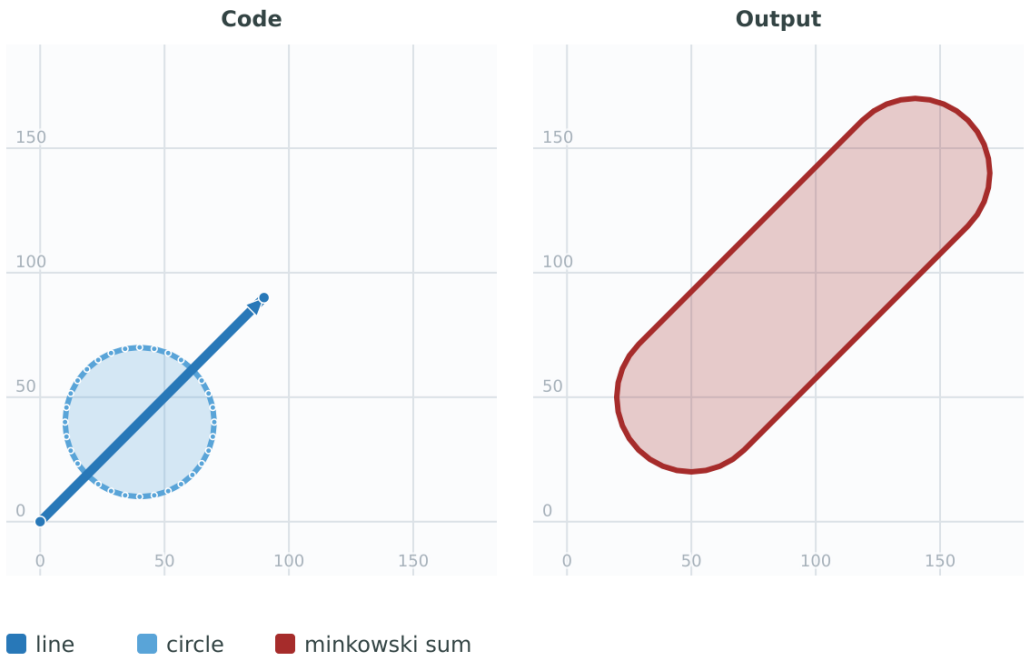
Examples

Minkowski sum of a LineString and a circular polygon. The line crosses the polygon, and the result sweeps the polygon along the line.

Code

```
WITH data AS (
  SELECT ST_MakeLine(ST_Point(10, 10), ST_Point(100, 100)) AS line,
         ST_Buffer('POINT(50 50)'::geometry, 30) AS circle
)
SELECT line AS input_line,
       circle AS input_circle,
       CG_MinkowskiSum(line, circle) AS minkowski_sum
FROM data;
```

Figure



Minkowski sum of a MultiPoint and a polygon. The result contains one translated copy of the polygon for each point.

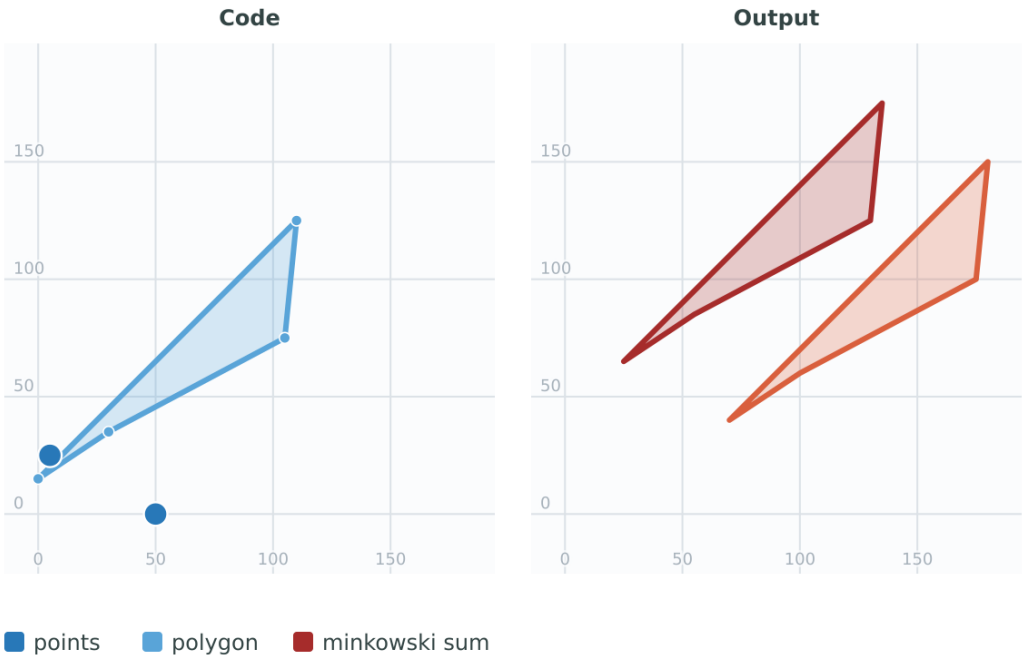
Code

```
WITH data AS (
  SELECT 'MULTIPOINT(25 50,70 25)'::geometry AS points,
         'POLYGON((130 150,20 40,50 60,125 100,130 150))'::geometry AS polygon
)
SELECT points AS input_points,
       polygon AS input_polygon,
       CG_MinkowskiSum(points, polygon) AS minkowski_sum
FROM data;
```

Output

```
MULTIPOINT((25 50),(70 25)) | POLYGON((130 150,20 40,50 60,125 100,130 150)) | MULTIPOLYGON(((45 90,75 110,150 150,155 ↩
    200,45 90)),((90 65,120 85,195 125,200 175,90 65)))
```

Figure



8.3.26 CG_OptimalAlphaShape

CG_OptimalAlphaShape — Computes an Alpha-shape enclosing a geometry using an "optimal" alpha value.

Synopsis

geometry **CG_OptimalAlphaShape**(geometry geom, boolean allow_holes = false, integer nb_components = 1);

Description

Computes the "optimal" alpha-shape of the points in a geometry. The alpha-shape is computed using a value of α chosen so that:

1. the number of polygon elements is equal to or smaller than `nb_components` (which defaults to 1)
2. all input points are contained in the shape

The result will not contain holes unless the optional `allow_holes` argument is specified as true.

Availability: 3.5.0 - requires SFCGAL >= 1.4.1.

This method needs SFCGAL backend.

Examples

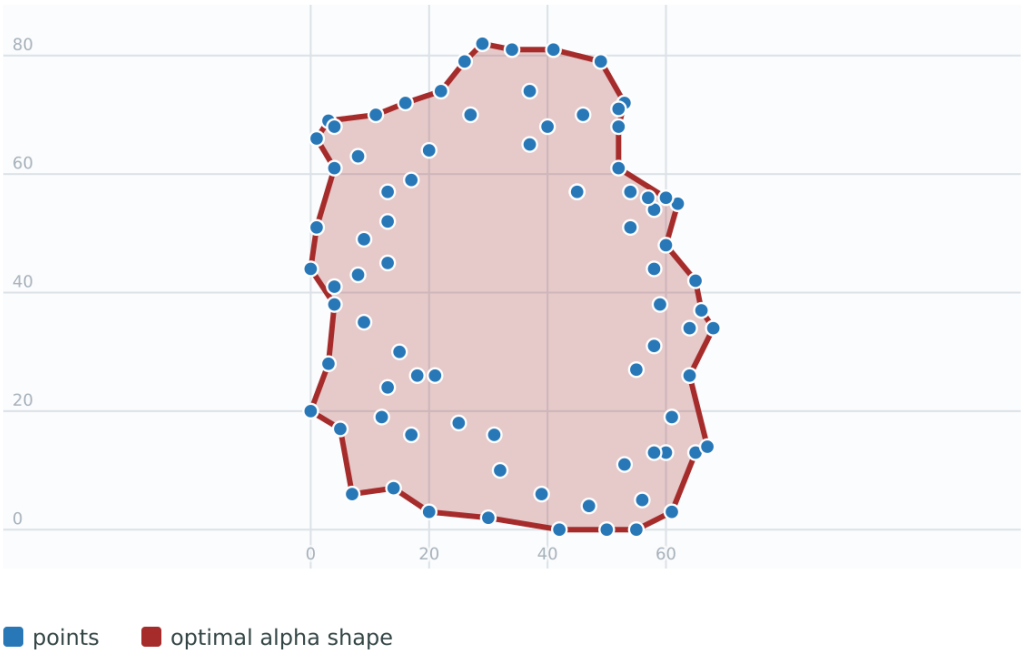
The original point cloud and the optimal alpha-shape share one comparison frame.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),
(88 29),(24 82),(32 51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 65),
(81 47),(88 58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 59),(34 86),(45 90),(64 97))'::geometry AS input_points
)
SELECT input_points AS input_points,

         CG_OptimalAlphaShape(input_points) AS optimal_alpha_shape
FROM data;
```

Figure



Allowing holes keeps the same outer shell but carves a central hole.

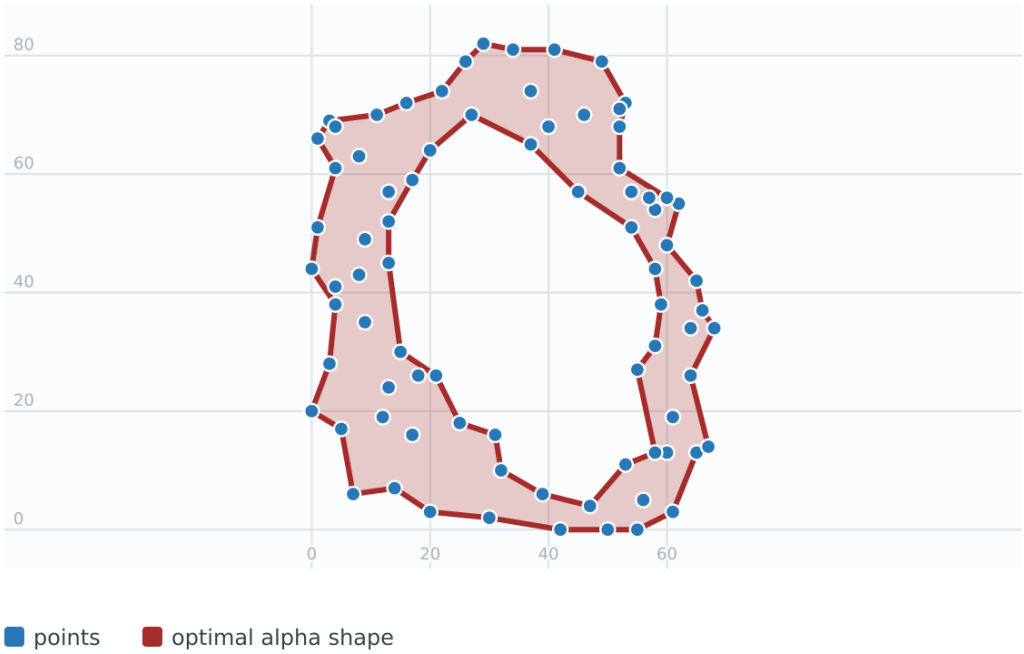
Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),(88 29),(24 82),(32 51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 65),(81 47),(88 58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 59),(34 86),(45 90),(64 97))'::geometry AS input_points
```

```
)
SELECT input_points AS input_points,

         CG_OptimalAlphaShape(input_points, allow_holes => true) AS optimal_alpha_shape
FROM data;
```

Figure



See Also

ST_ConcaveHull, CG_AlphaShape

8.3.27 CG_OptimalConvexPartition

CG_OptimalConvexPartition — Computes an optimal convex partition of the polygon geometry

Synopsis

geometry **CG_OptimalConvexPartition**(geometry geom);

Description

Computes an optimal convex partition of the polygon geometry.



Note
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. CG_OptimalConvexPartition produces a partition that is optimal in the number of pieces.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

Examples

The optimal convex partition uses the same polygon and produces the smallest documented piece count.

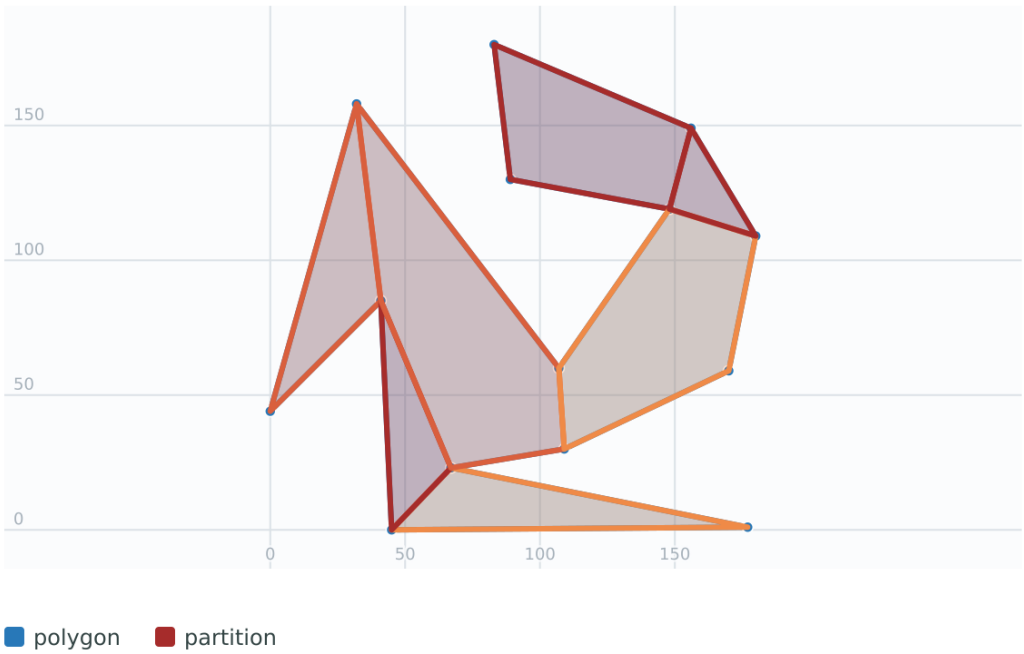
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_OptimalConvexPartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((156 150,83 181,89 131,148 120,156 150)), POLYGON((32 159,0 45,41 86,32 159)), POLYGON ↩
    ((45 1,177 2,67 24,45 1)), POLYGON((41 86,45 1,67 24,41 86)), POLYGON((107 61,32 159,41 86,67 24,109 31,107 61)), ↩
    POLYGON((148 120,107 61,109 31,170 60,180 110,148 120)), POLYGON((156 150,148 120,180 110,156 150)))
```

Figure



See Also

CG_YMonotonePartition, CG_ApproxConvexPartition, CG_GreeneApproxConvexPartition

8.3.28 CG_StraightSkeleton

CG_StraightSkeleton — Compute a straight skeleton from a geometry

Synopsis

geometry **CG_StraightSkeleton**(geometry geom, boolean use_distance_as_m = false);

Description

Computes the straight skeleton of a polygon by tracing the inward propagation of its edges. When *use_distance_as_m* is `true`, the M coordinate of each output point stores its distance from the input boundary.

Availability: 3.5.0

Requires SFCGAL >= 1.3.8 for option use_distance_as_m



Note

This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.



This method needs SFCGAL backend.



This function supports Polyhedral surfaces.



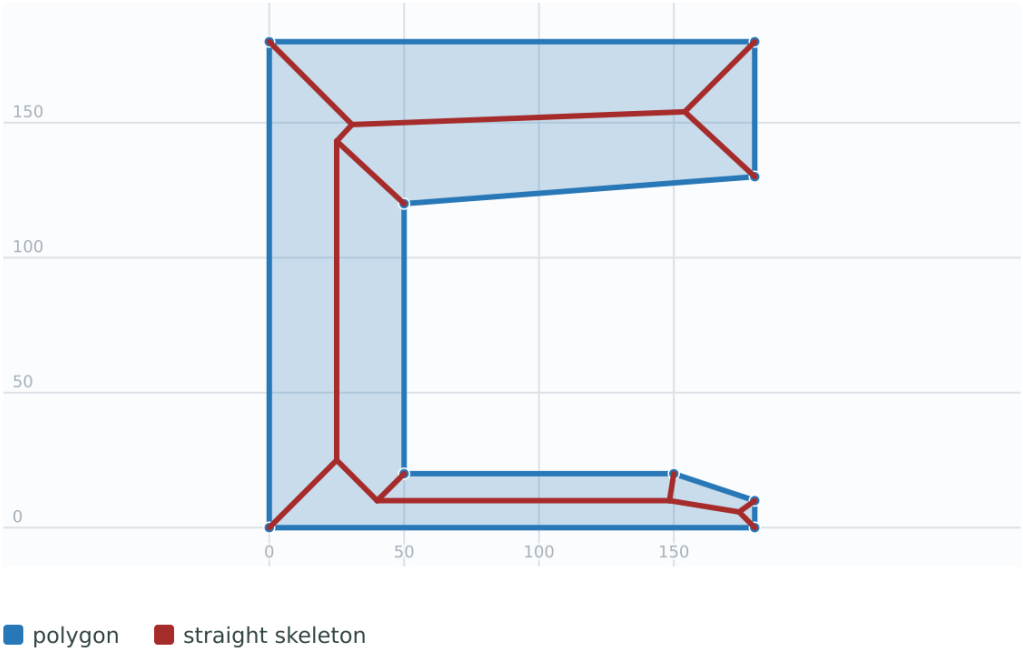
This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Code

```
WITH data AS (
  SELECT 'POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))' ↩
    ::geometry AS polygon
)
SELECT CG_StraightSkeleton(polygon) AS straight_skeleton
FROM data;
```

Figure



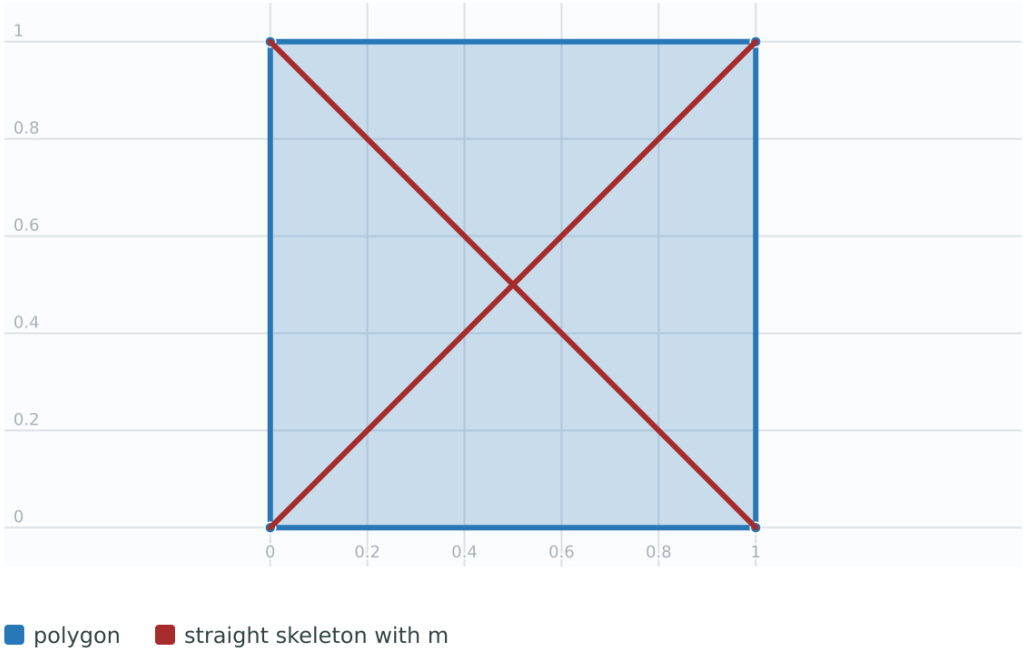
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,1 0,1 1,0 1,0 0))'::geometry AS polygon
)
SELECT CG_StraightSkeleton(polygon, true) AS straight_skeleton_with_m
FROM data;
```

Output

```
MULTILINESTRING M ((0 0 0,0.5 0.5 0.5),(1 0 0,0.5 0.5 0.5),(1 1 0,0.5 0.5 0.5),(0 1 0,0.5 0.5 0.5))
```

Figure



Note that valid inputs with rings that touch at a single point will raise an error.

Code

```
SELECT CG_StraightSkeleton('POLYGON((0 0,3 0,3 3,0 3,0 0),(0 0,1 2,2 1,0 0))'));
```

Output

```
NOTICE:  During straight_skeleton(A) :
NOTICE:    with A: POLYGON((0/1 0/1,3/1 0/1,3/1 3/1,0/1 3/1,0/1 0/1),(0/1 0/1,1/1 2/1,2/1 1/1,0/1 0/1))
ERROR:  straight skeleton of Polygon with point touching rings is not implemented.
```

See Also

CG_StraightSkeletonPartition, CG_ExtrudeStraightSkeleton

8.3.29 CG_Tesselate

CG_Tesselate — Perform surface Tessellation of a polygon or polyhedralsurface and returns as a TIN or collection of TINS

Synopsis

geometry **CG_Tesselate**(geometry geom);

Description

Takes as input a surface such a MULTI(POLYGON) or POLYHEDRALSURFACE and returns a TIN representation via the process of tessellation using triangles.



Note

ST_TriangulatePolygon does similar to this function except that it returns a geometry collection of polygons instead of a TIN and also only works with 2D geometries.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Examples

Tessellation divides each face of a cube into triangles.

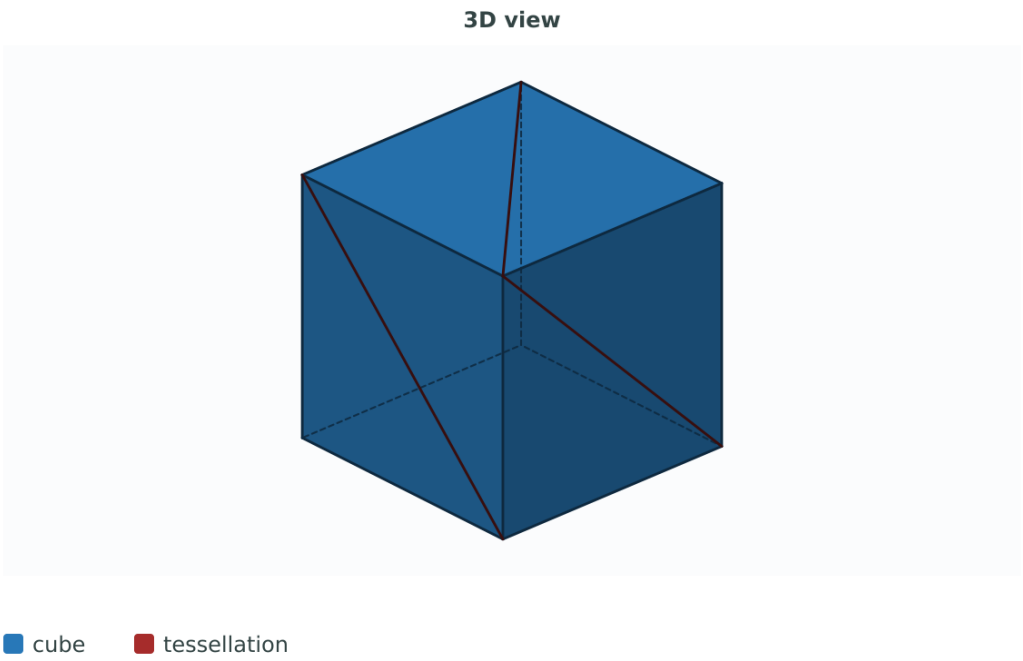
Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry AS cube
)
SELECT CG_Tesselate(cube) AS tessellation
FROM data;
```

Output

```
TIN Z (((0 0 0,0 0 1,0 1 1,0 0 0)),((0 1 0,0 0 0,0 1 1,0 1 0)),((0 0 0,0 1 0,1 1 0,0 0 0)),((1 0 0,0 0 0,1 1 0,1 0 0)) ↩
    ,((0 0 1,1 0 0,1 0 1,0 0 1)),((0 0 1,0 0 0,1 0 0,0 0 1)),((1 1 0,1 1 1,1 0 1,1 1 0)),((1 0 0,1 1 0,1 0 1,1 0 0)) ↩
    ,((0 1 0,0 1 1,1 1 1,0 1 0)),((1 1 0,0 1 0,1 1 1,1 1 0)),((0 1 1,1 0 1,1 1 1,0 1 1)),((0 1 1,0 0 1,1 0 1,0 1 1)))
```

Figure



Tessellation of a Polygon.

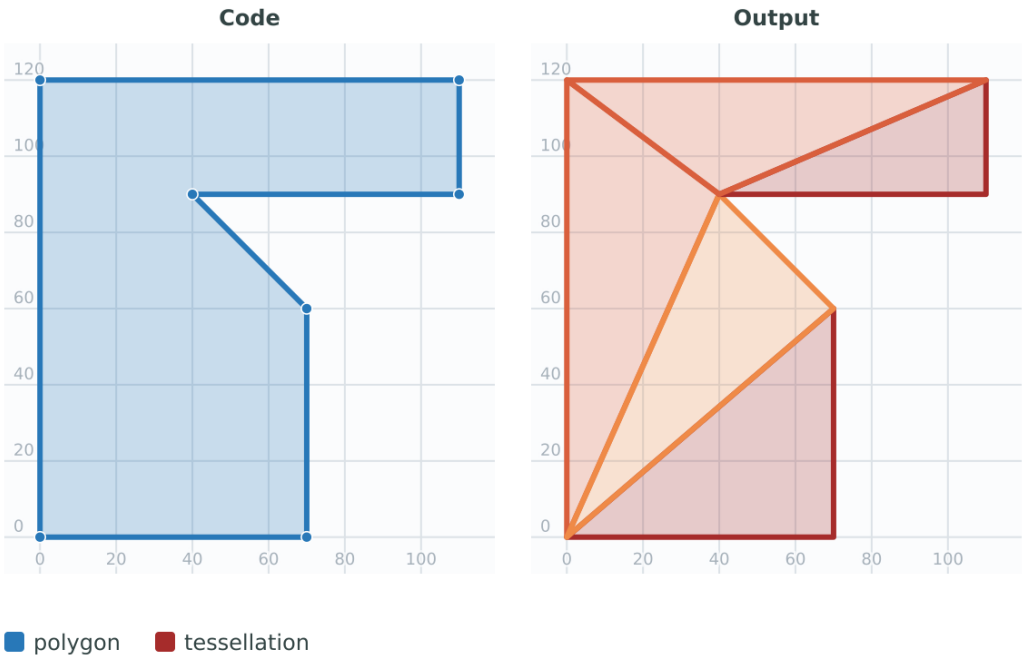
Code

```
WITH data AS (
  SELECT 'POLYGON((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190))'::geometry AS ↩
    polygon
)
SELECT polygon AS input_polygon,
       CG_Tesselate(polygon) AS tessellation
FROM data;
```

Output

```
POLYGON((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190)) | TIN(((80 130,10 70,80 70,80 130)),((50 160,10 ↩
    190,10 70,50 160)),((80 130,50 160,10 70,80 130)),((120 160,120 190,50 160,120 160)),((120 190,10 190,50 160,120 ↩
    190)))
```

Figure



See Also

CG_ConstrainedDelaunayTriangles, ST_DelaunayTriangles, ST_TriangulatePolygon

8.3.30 CG_Triangulate

CG_Triangulate — Triangulates a polygonal geometry

Synopsis

geometry **CG_Triangulate**(geometry geom);

Description

Triangulates a polygonal geometry.

Performed by the SFCGAL module



Note
NOTE: this function returns a geometry representing the triangulated result.

Availability: 3.5.0


This method needs SFCGAL backend.

Geometry Examples

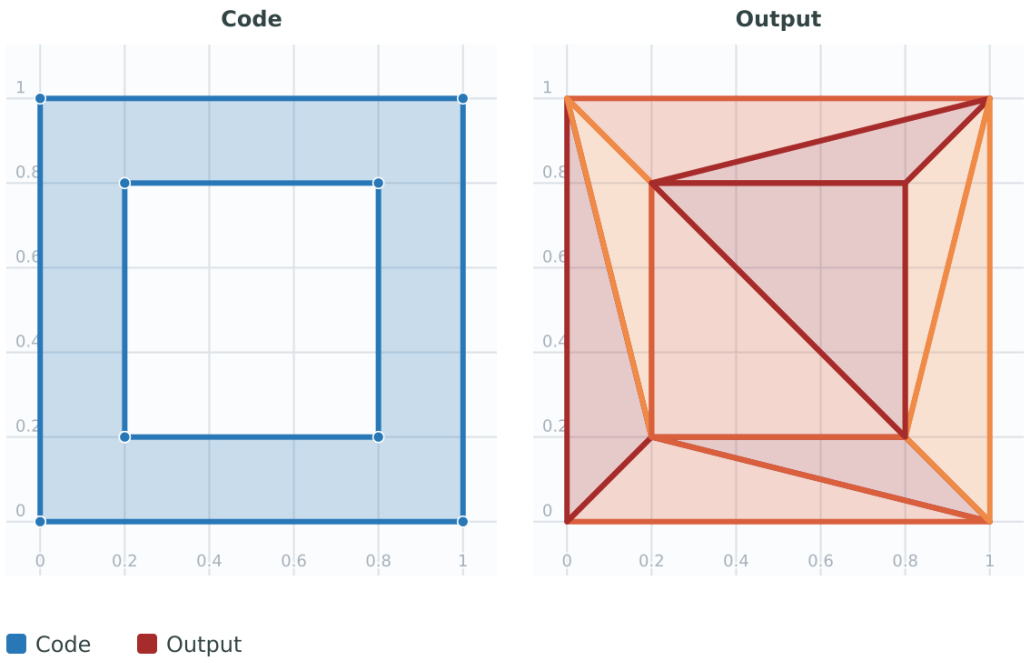
Code

```
SELECT CG_Triangulate(
  'POLYGON((0 0,1 0,1 1,0 1,0 0),
           (0.2 0.2,0.2 0.8,0.8 0.8,0.8 0.2,0.2 0.2))'
);
```

Output

```
TIN(((0.8 0.2,0.2 0.2,1 0,0.8 0.2)),((0.2 0.2,0 0,1 0,0.2 0.2)),((1 1,0.8 0.8,0.8 0.2,1 1)),((0 1,0 0,0.2 0.2,0 1)),((0 ↩
     1,0.2 0.8,1 1,0 1)),((0 1,0.2 0.2,0.2 0.8,0 1)),((0.2 0.8,0.8 0.8,1 1,0.2 0.8)),((0.2 0.8,0.2 0.2,0.8 0.2,0.2  ↩
    0.8)),((1 1,0.8 0.2,1 0,1 1)),((0.8 0.8,0.2 0.8,0.8 0.2,0.8 0.8)))
```

Figure



See Also

CG_ConstrainedDelaunayTriangles, ST_DelaunayTriangles, ST_TriangulatePolygon

8.3.31 CG_Visibility

CG_Visibility — Compute a visibility polygon from a point or a segment in a polygon geometry

Synopsis

geometry **CG_Visibility**(geometry polygon, geometry point);
geometry **CG_Visibility**(geometry polygon, geometry pointA, geometry pointB);

Description

Returns the portion of a polygon visible from an observer point, or from the observer segment defined by *pointA* and *pointB*.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

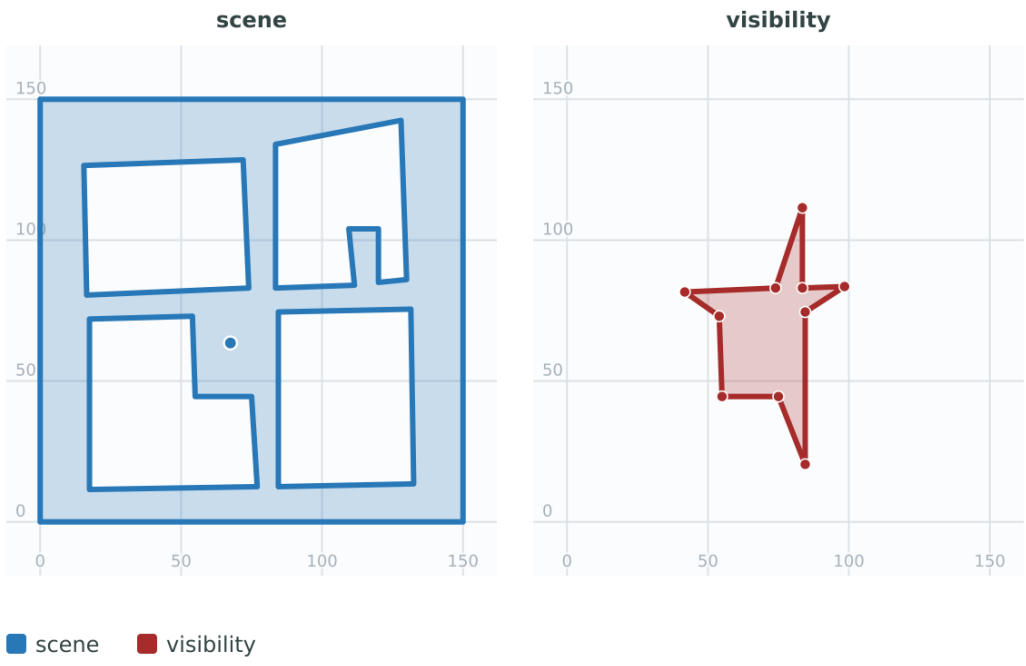
Examples

Visibility from a point inside a polygon with holes.

Code

```
WITH data AS (
  SELECT 'POLYGON((23.5 23.5,23.5 173.5,173.5 173.5,173.5 23.5,23.5 23.5),(108 98,108 36,156 ↩
    37,155 99,108 98),(107 157.5,107 106.5,135 107.5,133 127.5,143.5 127.5,143.5 ↩
   108.5,153.5 109.5,151.5 166,107 157.5),(41 95.5,41 35,100.5 36,98.5 68,78.5 68,77.5 ↩
   96.5,41 95.5),(39 150,40 104,97.5 106.5,95.5 152,39 150))'::geometry AS polygon,
        'POINT(91 87)'::geometry AS observer
)
SELECT ST_Collect(polygon, observer) AS input_scene,
       CG_Visibility(polygon, observer) AS visibility
FROM data;
```

Figure



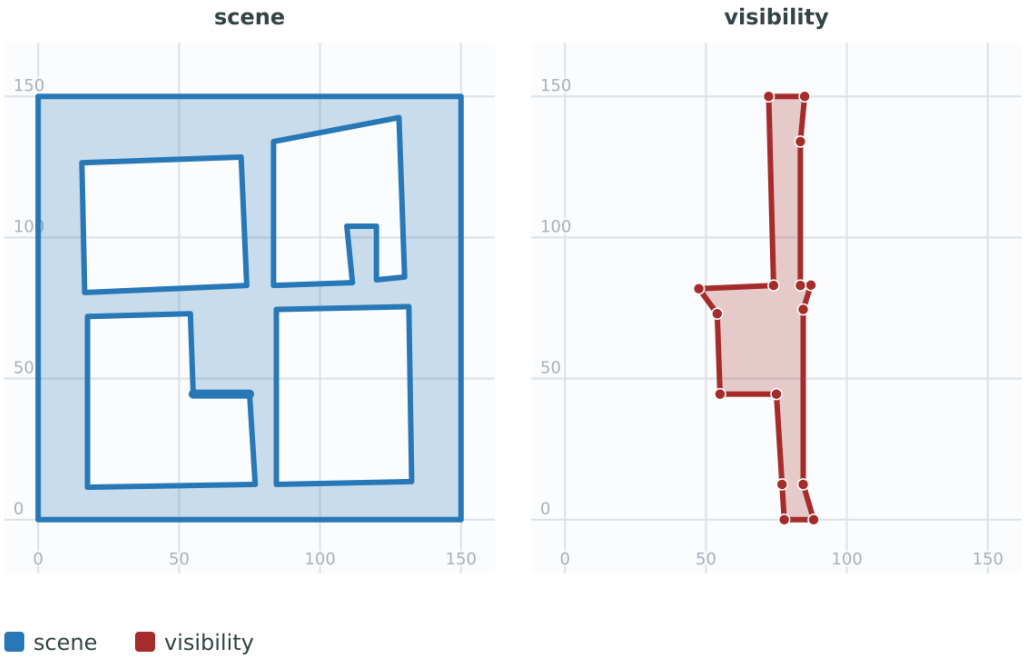
Visibility from a segment inside the same polygon.

Code

```
WITH data AS (
  SELECT 'POLYGON((23.5 23.5,23.5 173.5,173.5 173.5,173.5 23.5,23.5 23.5),(108 98,108 36,156 ↩
    37,155 99,108 98),(107 157.5,107 106.5,135 107.5,133 127.5,143.5 127.5,143.5 ↩
   108.5,153.5 109.5,151.5 166,107 157.5),(41 95.5,41 35,100.5 36,98.5 68,78.5 68,77.5 ↩
   96.5,41 95.5),(39 150,40 104,97.5 106.5,95.5 152,39 150))'::geometry AS polygon,
```

```
        'LINESTRING(78.5 68,98.5 68)'::geometry AS observer
)
SELECT ST_Collect(polygon, observer) AS input_scene,
       CG_Visibility(polygon,
                     ST_StartPoint(observer),
                     ST_EndPoint(observer)) AS visibility
FROM data;
```

Figure



8.3.32 CG_YMonotonePartition

CG_YMonotonePartition — Computes y-monotone partition of the polygon geometry

Synopsis

geometry **CG_YMonotonePartition**(geometry geom);

Description

Computes y-monotone partition of the polygon geometry.

> **Note**
> A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. A y-monotone polygon is a polygon whose vertices v1,. . . ,vn can be divided into two chains v1,. . . ,vk and vk,. . . ,vn,v1, such that any horizontal line intersects either chain at most once. This algorithm does not guarantee a bound on the number of polygons produced with respect to the optimal number.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

Examples

The same polygon can also be partitioned into y-monotone pieces for downstream processing.

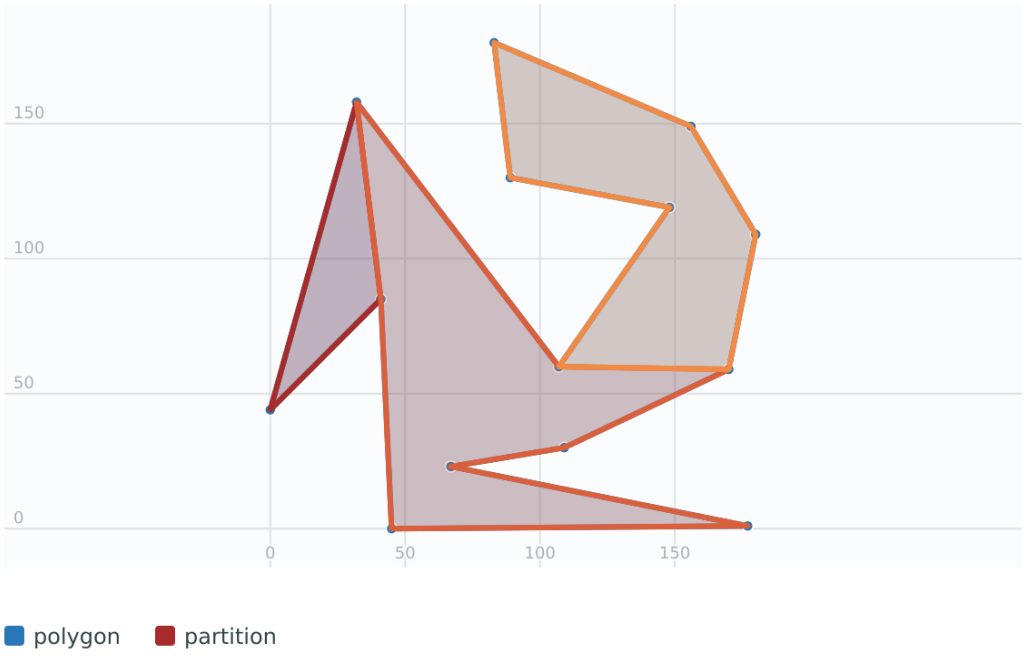
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_YMonotonePartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((32 159,0 45,41 86,32 159)), POLYGON((107 61,32 159,41 86,45 1,177 2,67 24,109 31,170 ↩
    60,107 61)), POLYGON((156 150,83 181,89 131,148 120,107 61,170 60,180 110,156 150)))
```

Figure



See Also

CG_ApproxConvexPartition, CG_GreeneApproxConvexPartition, CG_OptimalConvexPartition

8.3.33 CG_StraightSkeletonPartition

CG_StraightSkeletonPartition — Computes the straight skeleton partition of a polygon.

Synopsis

geometry **CG_StraightSkeletonPartition**(geometry geom, boolean auto_orientation);

Description

Computes the straight skeleton partition of the input polygon geometry *geom*. The straight skeleton is a partitioning of the polygon into faces formed by tracing the collapse of its edges. If *auto_orientation* is set to true, the function will automatically adjust the orientation of the input polygon to ensure correct results.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0.


This method needs SFCGAL backend.

Examples

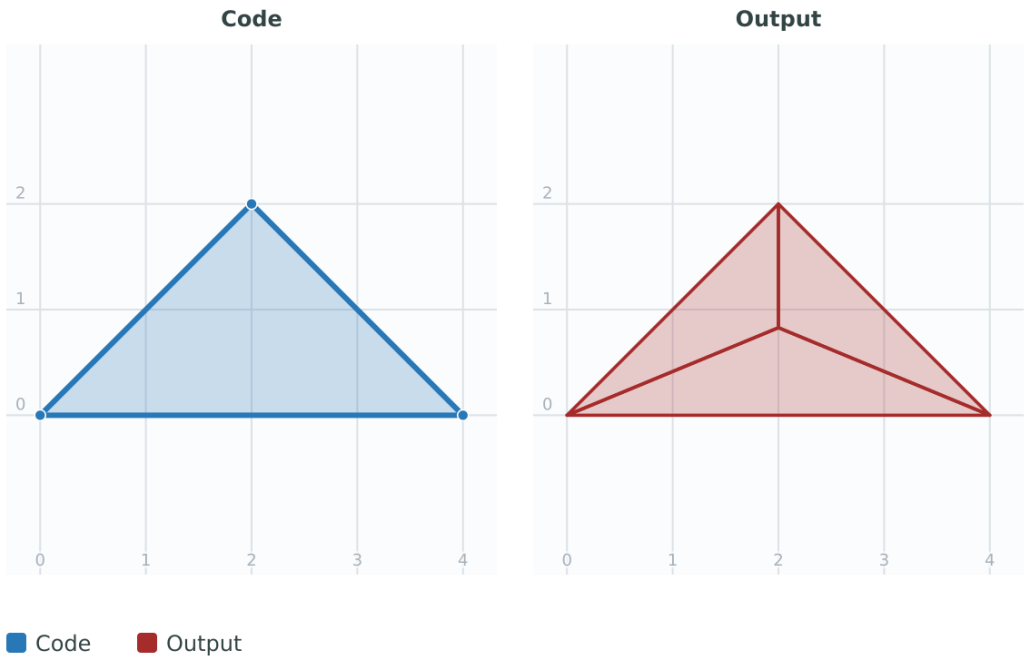
Code

```
SELECT CG_StraightSkeletonPartition('POLYGON((0 0,4 0,2 2,0 0))', true);
```

Output

```
POLYHEDRALSURFACE(((0 0,2 0.82842712474619,2 2,0 0)),((4 0,2 0.82842712474619,0 0,4 0)),((2 2,2 0.82842712474619,4 0,2
    2)))
```

Figure

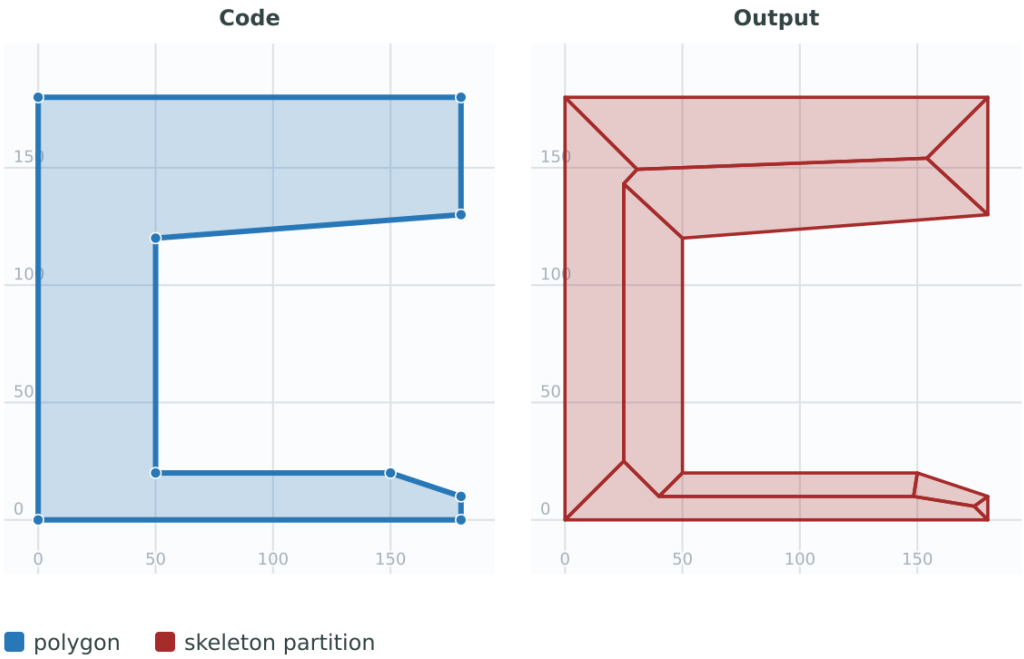


Straight skeleton partition of a polygon.

Code

```
WITH data AS (
  SELECT 'POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))' ↩
    ::geometry AS polygon
)
SELECT polygon AS input_polygon,
       CG_StraightSkeletonPartition(polygon, true) AS skeleton_partition
FROM data;
```

Figure



See Also

CG_StraightSkeleton, ST_Polygonize

8.3.34 CG_3DBuffer

CG_3DBuffer — Computes a 3D buffer around a geometry.

Synopsis

geometry **CG_3DBuffer**(geometry geom, float8 radius, integer segments, integer buffer_type);

Description

Generates a 3D buffer around the input geometry *geom* with a specified *radius*. The buffer is constructed in 3D space, creating a volumetric representation of the geometry's surroundings. The *segments* parameter defines the number of segments used to approximate the curved sections of the buffer, with a minimum value of 4 segments required. The *buffer_type* specifies the type of buffer to create: 0: Rounded buffer (default) 1: Flat buffer 2: Square buffer

Input geometry must be a Point or LineString.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

Examples

Code

```
SELECT CG_3DBuffer('POINT(0 0 0)', 1, 8, 0);
```

Output

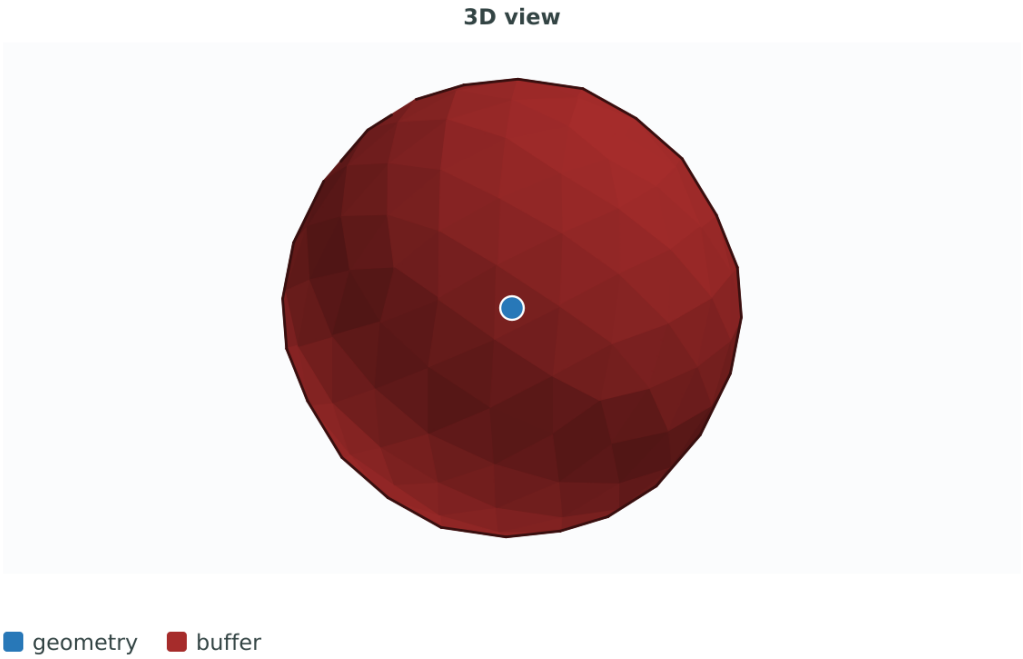
```
POLYHEDRALSURFACE Z (((0 0 1, 0.5 -0.5 0.71, 0 -0.71 0.71, 0 0 1)), ... )
```

A rounded point buffer using 32 segments.

Code

```
SELECT
  'POINT(100 90)'::geometry AS input_geometry,
  CG_3DBuffer('POINT(100 90)'::geometry, 50, 32, 0) AS buffer;
```

Figure

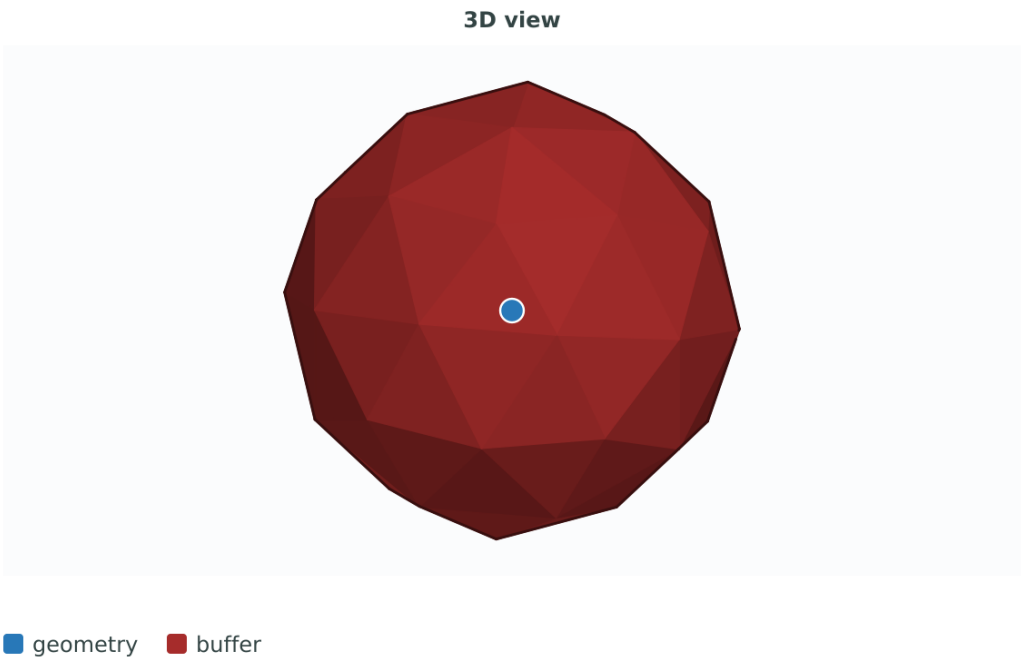


A coarser rounded point buffer using 5 segments.

Code

```
SELECT
  'POINT(100 90)'::geometry AS input_geometry,
  CG_3DBuffer('POINT(100 90)'::geometry, 50, 5, 0) AS buffer;
```

Figure

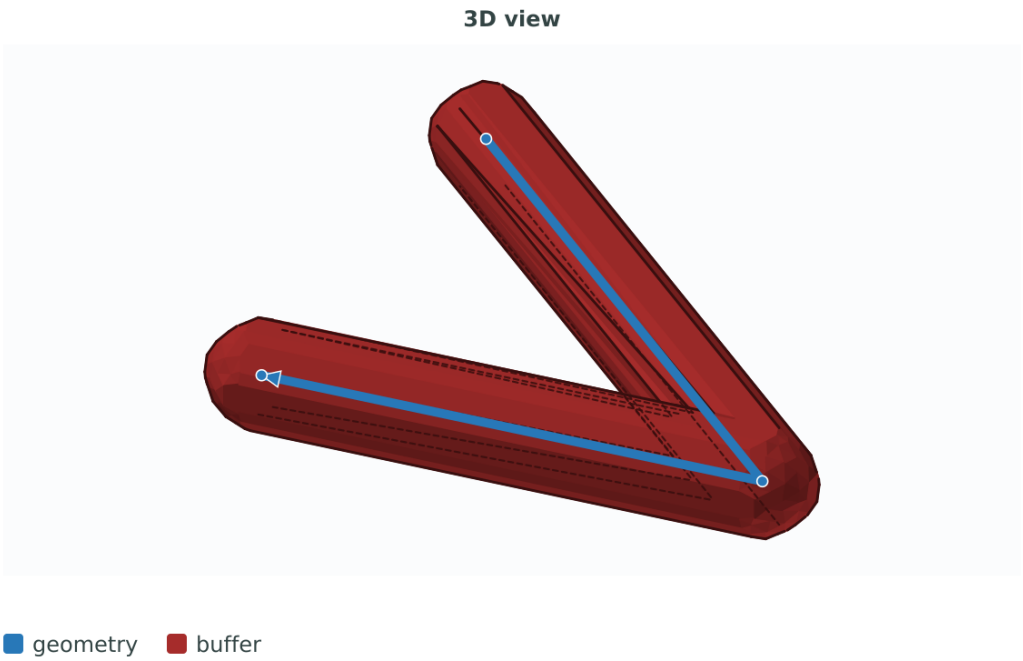


A rounded buffer around a line using 32 segments.

Code

```
SELECT
  'LINESTRING(50 50,150 150,150 50)'::geometry AS input_geometry,
  CG_3DBuffer('LINESTRING(50 50,150 150,150 50)'::geometry, 10, 32, 0) AS buffer;
```

Figure

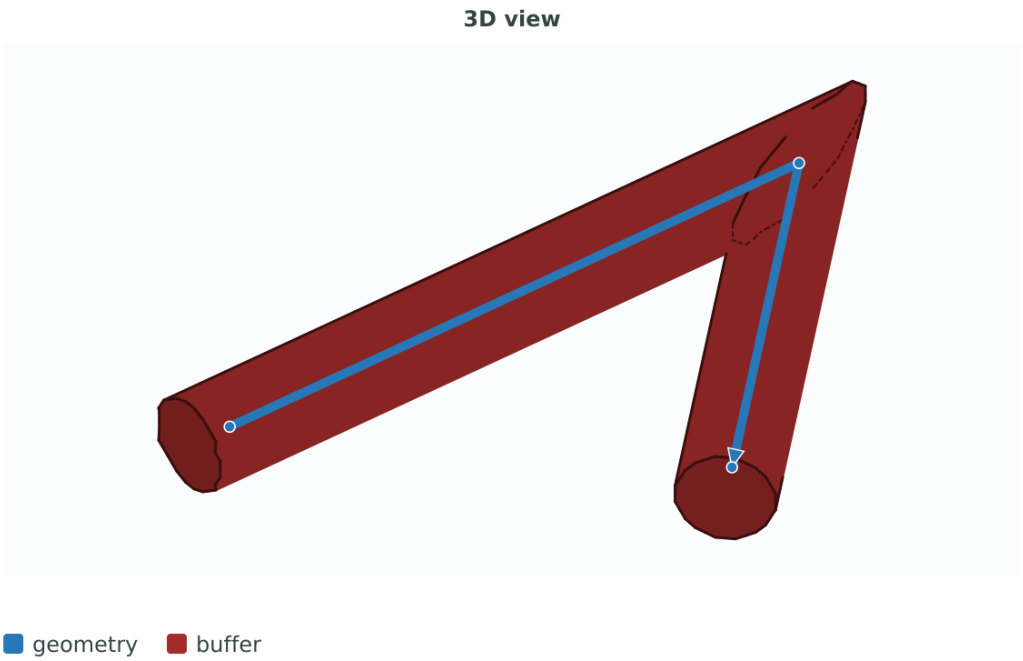


A square buffer around the same line using 32 segments.

Code

```
SELECT
  'LINESTRING(50 50,150 150,150 50)'::geometry AS input_geometry,
  CG_3DBuffer('LINESTRING(50 50,150 150,150 50)'::geometry, 10, 32, 2) AS buffer;
```

Figure



See Also

ST_Buffer, CG_3DConvexHull, ST_AsX3D

8.3.35 CG_Rotate

CG_Rotate — Rotates a geometry by a given angle around the origin (0,0).

Synopsis

geometry **CG_Rotate**(geometry geom, float8 angle);

Description

Rotates the input geometry *geom* by *angle* radians around the origin point (0,0). The rotation is performed in 2D space; Z coordinates are not modified. Positive angles rotate the geometry counter-clockwise.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

 This method needs SFCGAL backend.

Examples

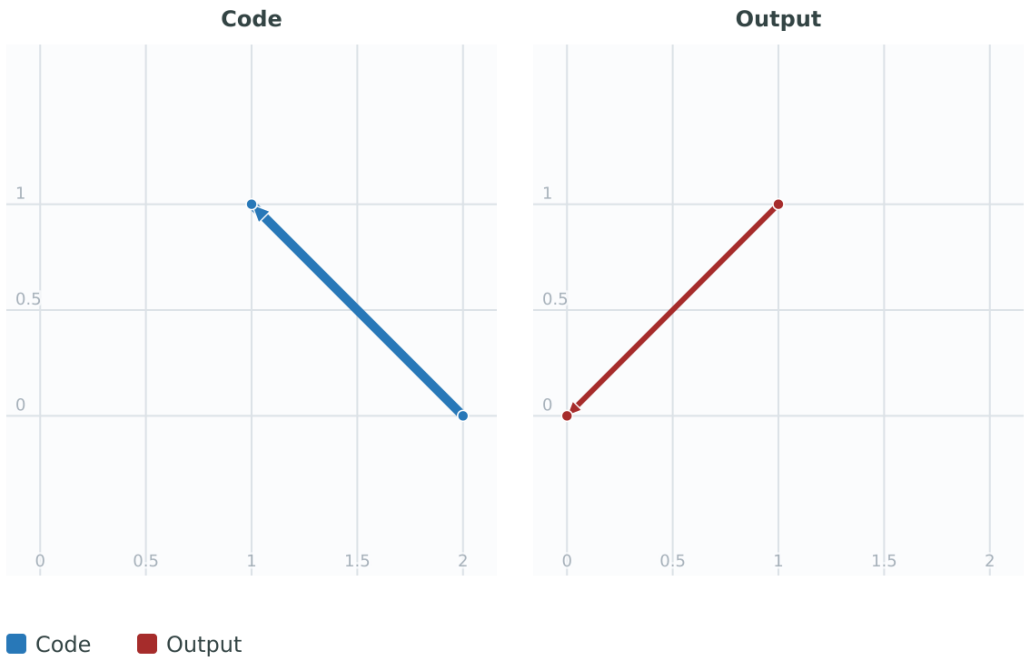
Code

```
SELECT CG_Rotate('LINESTRING(1 0,0 1)', pi()/2);
```

Output

```
LINESTRING(6.123233995736766e-17 1,-1 6.123233995736766e-17)
```

Figure



See Also

CG_2DRotate, ST_Rotate

8.3.36 CG_2DRotate

CG_2DRotate — Rotates a geometry by a given angle around a specified point in 2D.

Synopsis

geometry **CG_2DRotate**(geometry geom, float8 angle, float8 cx, float8 cy);

Description

Rotates the input geometry *geom* by *angle* radians around the point (*cx*, *cy*). The rotation is performed in 2D space; Z coordinates are dropped. Positive angles rotate the geometry counter-clockwise.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

Examples

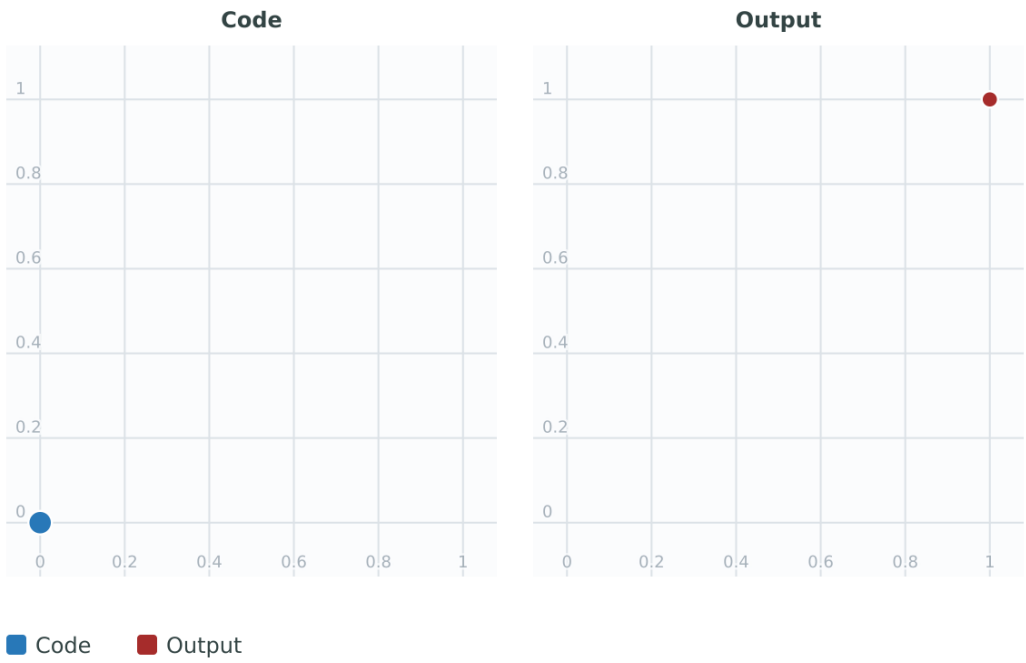
Code

```
SELECT CG_2DRotate('POINT(1 0)', pi()/2, 1, 1);
```

Output

```
POINT(2 1)
```

Figure



See Also

CG_Rotate, CG_3DRotate

8.3.37 CG_3DRotate

CG_3DRotate — Rotates a geometry in 3D space around an axis vector.

Synopsis

geometry **CG_3DRotate**(geometry geom, float8 angle, float8 ax, float8 ay, float8 az);

Description

Rotates the input geometry *geom* by *angle* radians around an axis defined by the vector (*ax*, *ay*, *az*) passing through the origin (0,0,0).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

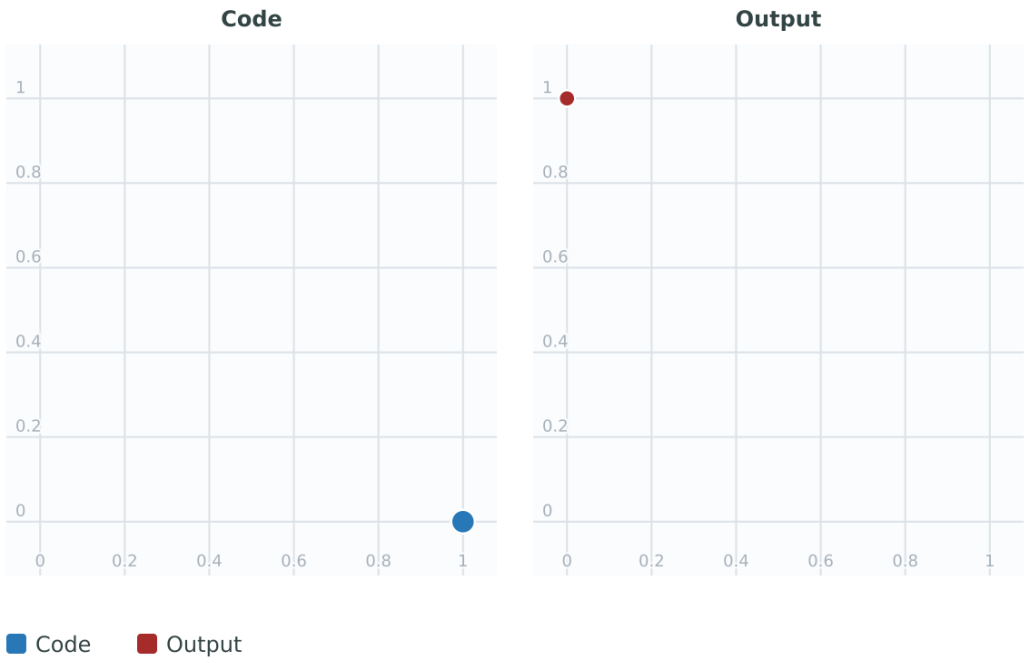
Code

```
SELECT CG_3DRotate('POINT(1 0 0)', pi()/2, 0, 0, 1);
```

Output

```
POINT Z (6.123233995736766e-17 1 0)
```

Figure



See Also

CG_RotateX, CG_RotateY, CG_RotateZ

8.3.38 CG_RotateX

CG_RotateX — Rotates a geometry around the X-axis by a given angle.

Synopsis

geometry **CG_RotateX**(geometry geom, float8 angle);

Description

Rotates the input geometry *geom* by *angle* radians around the X-axis.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

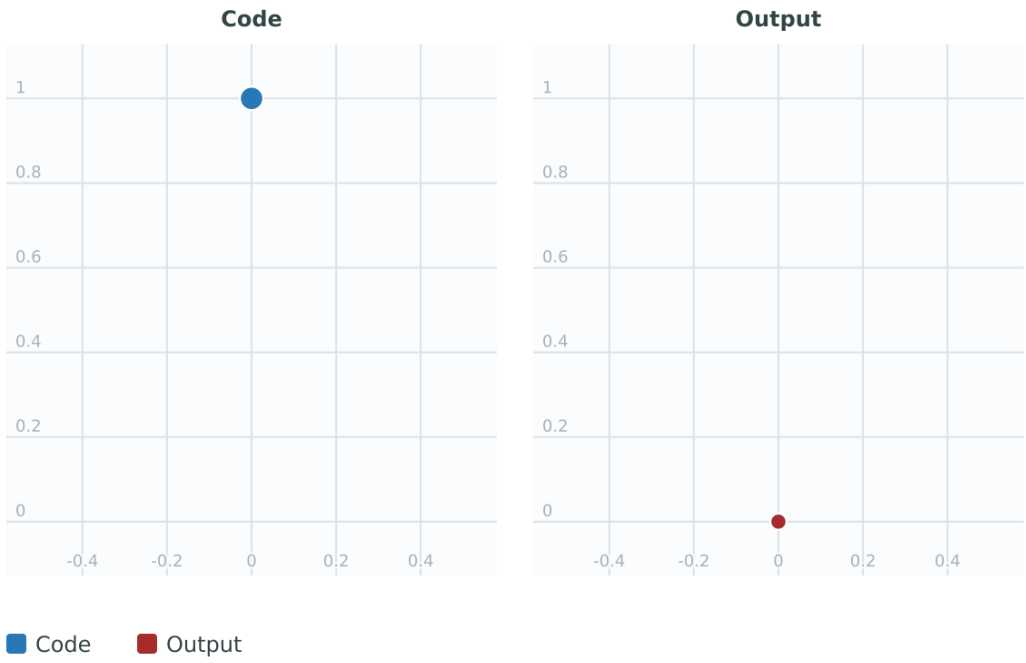
Code

```
SELECT CG_RotateX('POINT(0 1 0)', pi()/2);
```

Output

```
POINT Z (0 6.123233995736766e-17 1)
```

Figure



See Also

CG_RotateY, CG_RotateZ, CG_3DRotate

8.3.39 CG_RotateY

CG_RotateY — Rotates a geometry around the Y-axis by a given angle.

Synopsis

geometry **CG_RotateY**(geometry geom, float8 angle);

Description

Rotates the input geometry *geom* by *angle* radians around the Y-axis passing.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

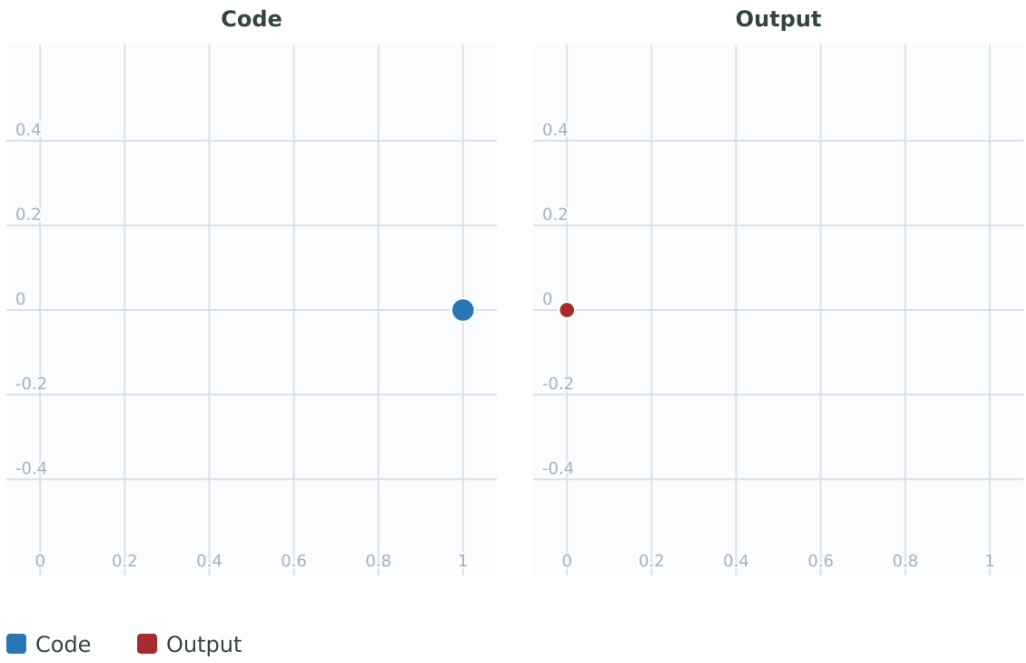
Code

```
SELECT CG_RotateY('POINT(1 0 0)', pi()/2);
```

Output

```
POINT Z (6.123233995736766e-17 0 -1)
```

Figure



See Also

CG_RotateX, CG_RotateZ, CG_3DRotate

8.3.40 CG_RotateZ

CG_RotateZ — Rotates a geometry around the Z-axis by a given angle.

Synopsis

geometry **CG_RotateZ**(geometry geom, float8 angle);

Description

Rotates the input geometry *geom* by *angle* radians around the Z-axis.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

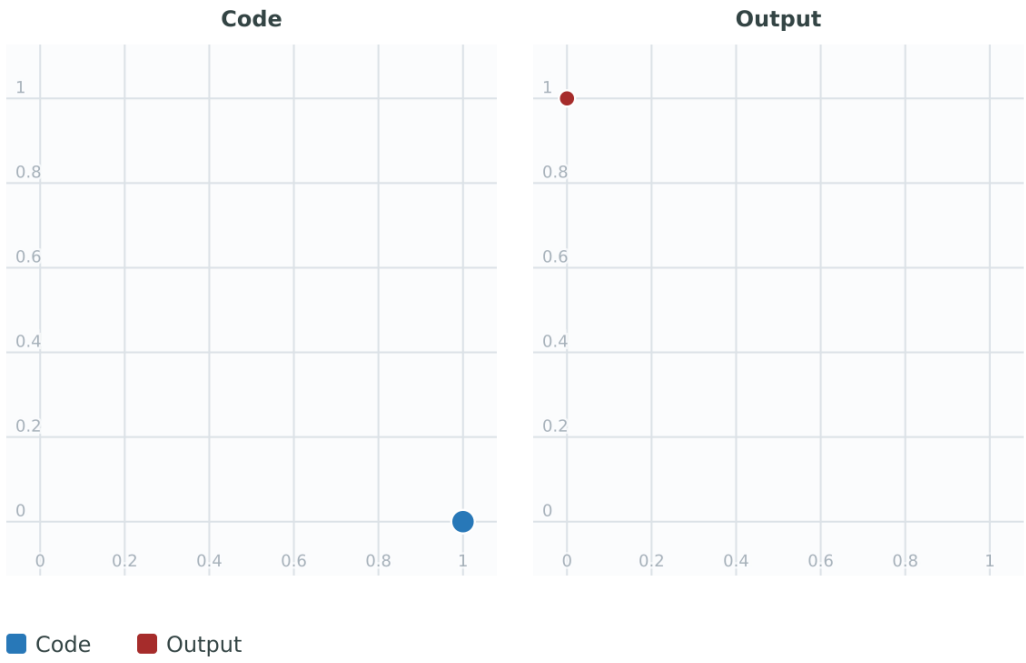
Code

```
SELECT CG_RotateZ('POINT(1 0 0)', pi()/2);
```

Output

```
POINT Z (6.123233995736766e-17 1 0)
```

Figure



See Also

CG_RotateX, CG_RotateY, CG_3DRotate

8.3.41 CG_Scale

CG_Scale — Scales a geometry uniformly in all dimensions by a given factor.

Synopsis

geometry **CG_Scale**(geometry geom, float8 factor);

Description

Scales the input geometry *geom* by a uniform scale *factor* in all dimensions (X, Y, and Z). The scaling is performed relative to the origin point (0,0,0).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

Examples

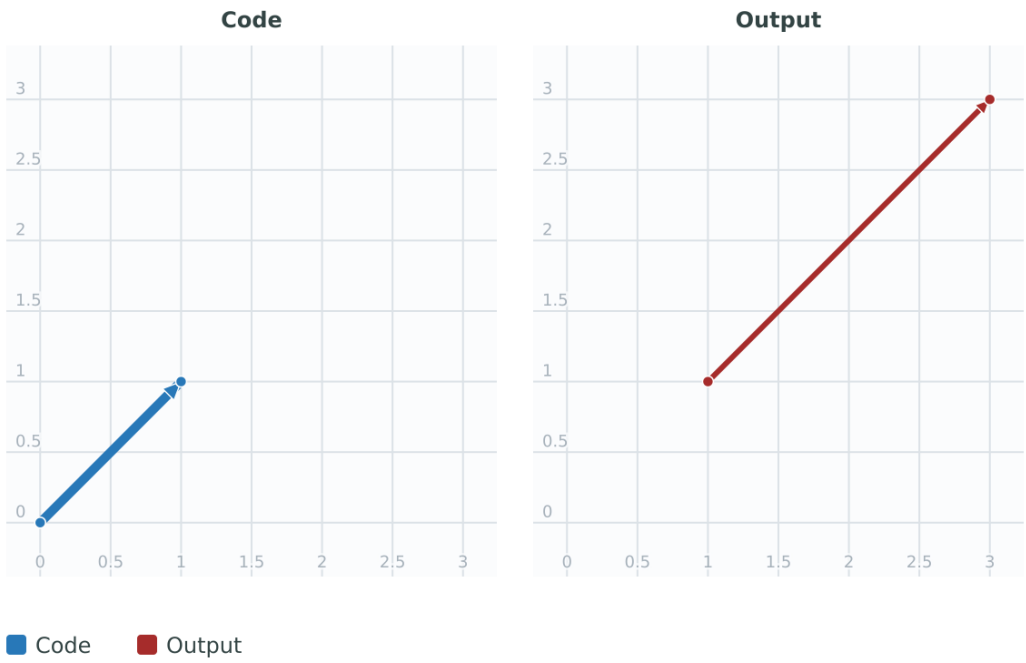
Code

```
SELECT CG_Scale('LINESTRING(1 1,2 2)', 2);
```

Output

```
LINESTRING(2 2,4 4)
```

Figure



See Also

CG_3DScale, CG_3DScaleAroundCenter, ST_Scale

8.3.42 CG_3DScale

CG_3DScale — Scales a geometry by separate factors along X, Y, and Z axes.

Synopsis

geometry **CG_3DScale**(geometry geom, float8 factorX, float8 factorY, float8 factorZ);

Description

Scales the input geometry *geom* by different factors along the X, Y, and Z axes. The scaling is performed relative to the origin point (0,0,0).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT CG_3DScale('POINT(1 1 1)', 2, 3, 4);
```

Output

```
POINT Z (2 3 4)
```

Figure



See Also

CG_Scale, CG_3DScaleAroundCenter

8.3.43 CG_3DScaleAroundCenter

CG_3DScaleAroundCenter — Scales a geometry in 3D space around a specified center point.

Synopsis

geometry **CG_3DScaleAroundCenter**(geometry geom, float8 factorX, float8 factorY, float8 factorZ, float8 centerX, float8 centerY, float8 centerZ);

Description

Scales the input geometry *geom* by different factors along the X, Y, and Z axes, relative to a specified center point (*centerX*, *centerY*, *centerZ*).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

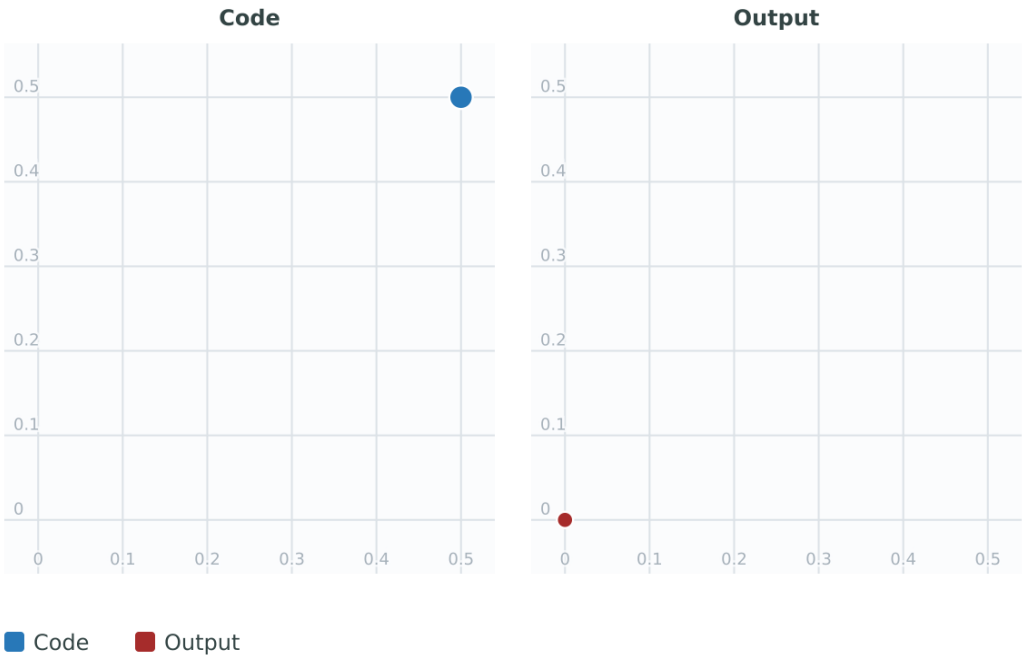
Code

```
SELECT CG_3DScaleAroundCenter('POINT(2 2 2)', 0.5, 0.5, 0.5, 1, 1, 1);
```

Output

```
POINT Z (1.5 1.5 1.5)
```

Figure



See Also

CG_Scale, CG_3DScale

8.3.44 CG_Translate

CG_Translate — Translates (moves) a geometry by given offsets in 2D space.

Synopsis

geometry **CG_Translate**(geometry geom, float8 deltaX, float8 deltaY);

Description

Translates the input geometry *geom* by adding *deltaX* to the X coordinates and *deltaY* to the Y coordinates. Z coordinates are dropped.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

Examples

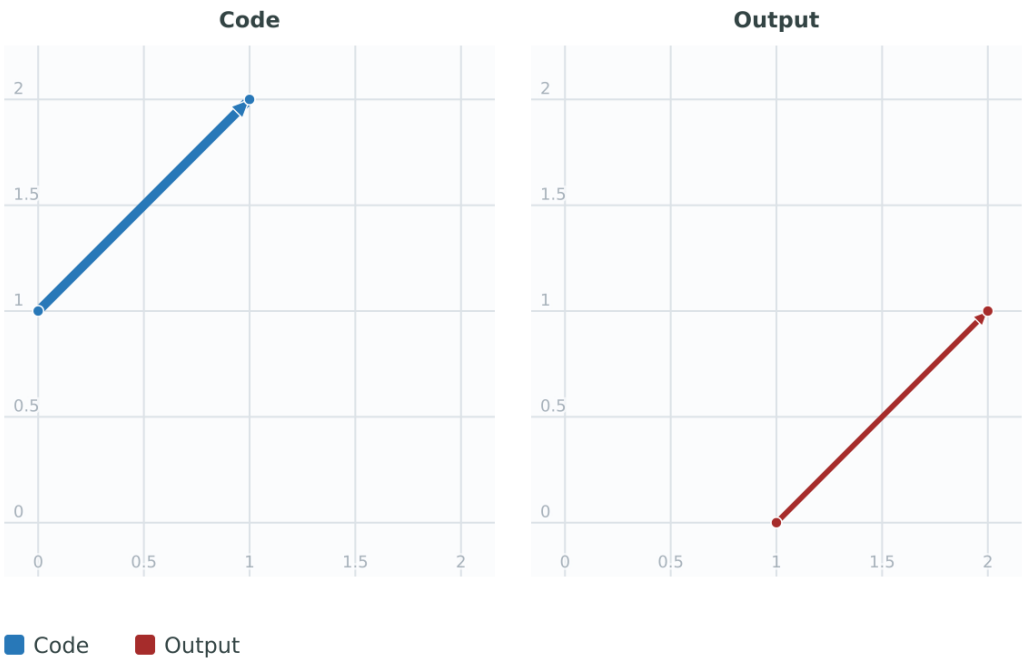
Code

```
SELECT CG_Translate('LINESTRING(1 1,2 2)', 1, -1);
```

Output

```
LINESTRING(2 0,3 1)
```

Figure



See Also

CG_3DTranslate, ST_Translate

8.3.45 CG_3DTranslate

CG_3DTranslate — Translates (moves) a geometry by given offsets in 3D space.

Synopsis

geometry **CG_3DTranslate**(geometry geom, float8 deltaX, float8 deltaY, float8 deltaZ);

Description

Translates the input geometry *geom* by adding *deltaX* to the X coordinates, *deltaY* to the Y coordinates, and *deltaZ* to the Z coordinates.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

Code

```
SELECT CG_3DTranslate('POINT(1 1 1)', 1, -1, 2);
```

Output

```
POINT Z (2 0 3)
```

Figure



See Also

CG_Translate, ST_Translate

8.3.46 CG_Simplify

CG_Simplify — Reduces the complexity of a geometry while preserving essential features and Z/M values.

Synopsis

geometry **CG_Simplify**(geometry geom, double precision threshold, boolean preserveTopology = false);

Description

Simplifies a geometry using SFCGAL's simplification algorithm, which reduces the number of points or vertices while preserving the essential features of the geometry. This function preserves Z and M values during simplification.

The algorithm is based on constrained triangulation and uses the CGAL Polyline Simplification 2 library with additional handling to preserve Z and M coordinates. When topology is preserved and geometries intersect, Z and M values are interpolated at intersection points.

This function works well with 3D terrain-like geometries (2.5D) but is not designed for vertical surfaces like walls.

Availability: 3.6.0 - requires SFCGAL >= 2.1.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Parameters

geom Input geometry

threshold Maximum distance threshold (in geometry unit) for simplification. The higher this value, the more simplified the resulting geometry will be.

preserveTopology If set to true, the function ensures that the topology of the geometry is preserved. When geometries intersect in this mode, Z and M values at intersection points are interpolated. The default value is false.

Return Value

Returns a simplified geometry with preserved Z and M values.

Examples

This example simplifies a polygon with a threshold of 0.5.

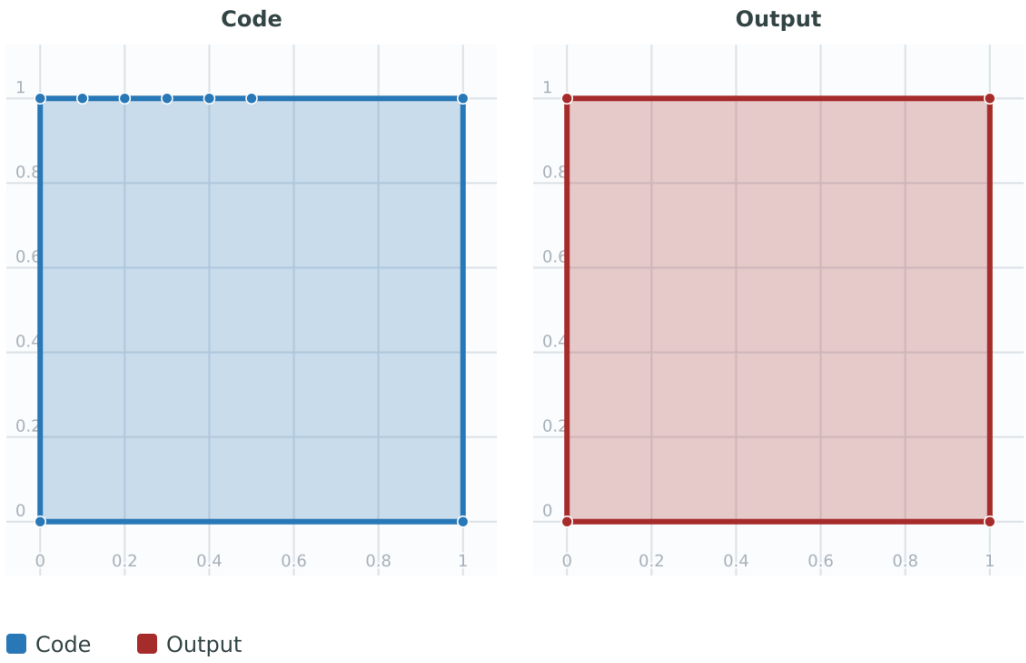
Code

```
SELECT CG_Simplify(
    'POLYGON((0 0,0 1,0.1 1,0.2 1,0.3 1,0.4 1,0.5 1,1 1,1 0,0 0))',
    0.5
  );
```

Output

```
POLYGON((0 0,0 1,1 1,1 0,0 0))
```

Figure



Preserving topology can keep additional support points in a mixed ZM collection.

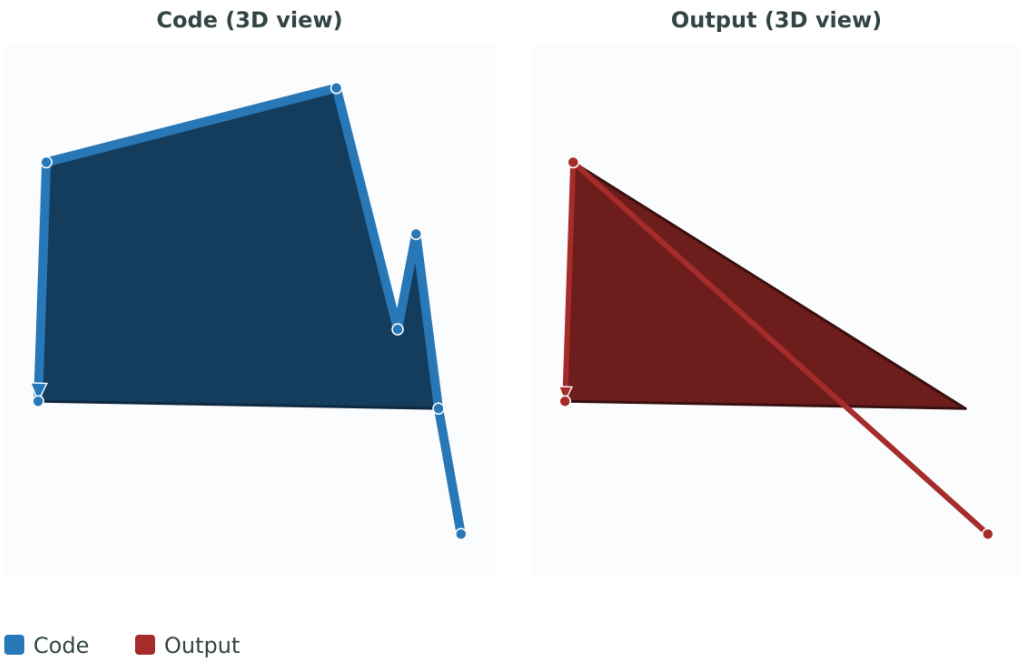
Code

```
SELECT CG_Simplify(
    'GEOMETRYCOLLECTION ZM(
      LINESTRING ZM(-1 -1 3 4,0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 ↩
    15 150),
      POLYGON ZM((0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 15 150,0 0 ↩
    10 100))
    )',
    2,
    false
  );
```

Output

```
GEOMETRYCOLLECTION ZM (LINESTRING ZM (-1 -1 3 4,2 19 25 250,-4 20 15 150),POLYGON ZM ((0 0 10 100,2 19 25 250,-4 20 15 ↩
    150,0 0 10 100)))
```

Figure



The topology-preserving variant keeps the shared anchor point at the collection intersection.

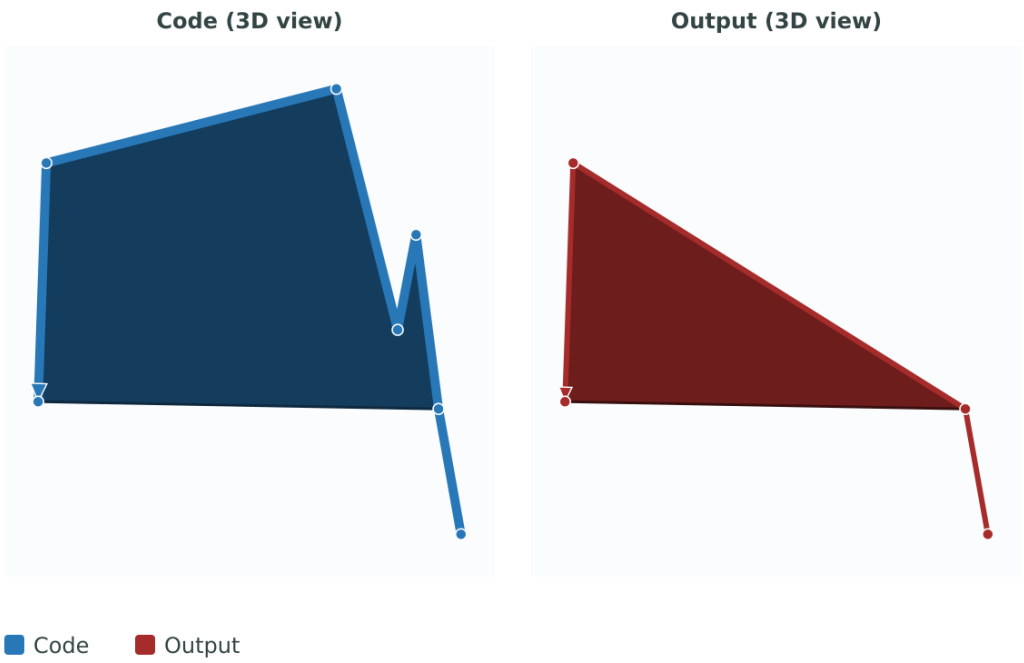
Code

```
SELECT CG_Simplify(
    'GEOMETRYCOLLECTION ZM(
      LINESTRING ZM(-1 -1 3 4,0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 ↩
    15 150),
      POLYGON ZM((0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 15 150,0 0 ↩
    10 100))
    )',
    2,
    true
  );
```

Output

```
GEOMETRYCOLLECTION ZM (LINESTRING ZM (-1 -1 3 4,0 0 10 100,2 19 25 250,-4 20 15 150),POLYGON ZM ((0 0 10 100,2 19 25 ↩
    250,-4 20 15 150,0 0 10 100)))
```

Figure

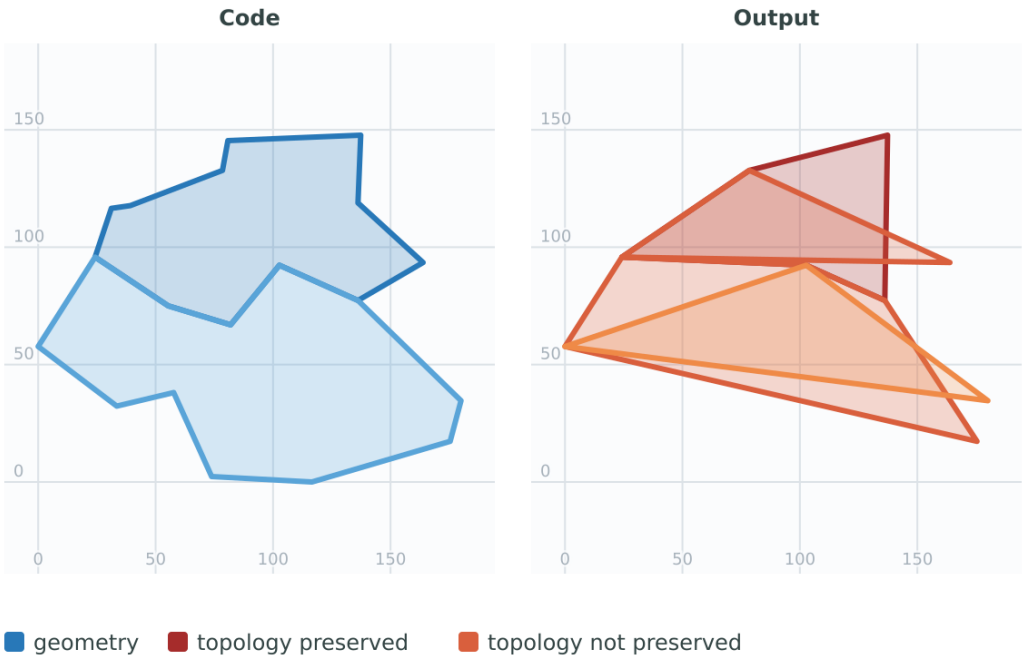


Compare simplification with and without topology preservation, using a threshold of 50.

Code

```
WITH data AS (
  SELECT ST_GeomFromText('GEOMETRYCOLLECTION(
    POLYGON((88.46 158.85,90.77 171.54,147.31 173.85,146.15 145,173.85 119.62,146.15 ↩
    103.46,112.69 118.46,91.92 93.08,65.38 101.15,34.23 121.92,41.15 142.69,49.23 ↩
    143.85,88.46 158.85)),
    POLYGON((112.69 118.46,146.15 103.46,190 60.77,185.38 43.46,126.54 26.15,83.85 ↩
    28.46,67.69 64.23,43.46 58.46,10 83.85,34.23 121.92,65.38 101.15,91.92 93.08,112.69 ↩
    118.46)))
  ') AS geom
)
SELECT geom AS input_geometry,
       CG_Simplify(geom, 50, true) AS topology_preserved,
       CG_Simplify(geom, 50, false) AS topology_not_preserved
FROM data;
```

Figure



See Also

ST_Simplify, ST_SimplifyPreserveTopology

8.3.47 CG_3DAlphaWrapping

CG_3DAlphaWrapping — Computes a 3D Alpha-wrapping strictly enclosing a geometry.

Synopsis

geometry **CG_3DAlphaWrapping**(geometry geom, integer relative_alpha, integer relative_offset);

Description

Computes the 3D Alpha Wrapping of the points in a geometry. An alpha wrapping is a watertight and orientable surface mesh that strictly encloses the input. It can be seen as an extension or refinement of an alpha-shape.

The `relative_alpha` parameter controls which features will appear in the output. It can have values from 0 to infinity. Higher relative_alpha values produce coarser wraps; lower values retain more detail.

The `relative_offset` parameter controls the tightness of the result. It can have values from 0 to infinity. If this parameter is set to 0, its value is automatically determined based on the relative_alpha parameter.

Availability: 3.6.0 - requires SFCGAL >= 2.1.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

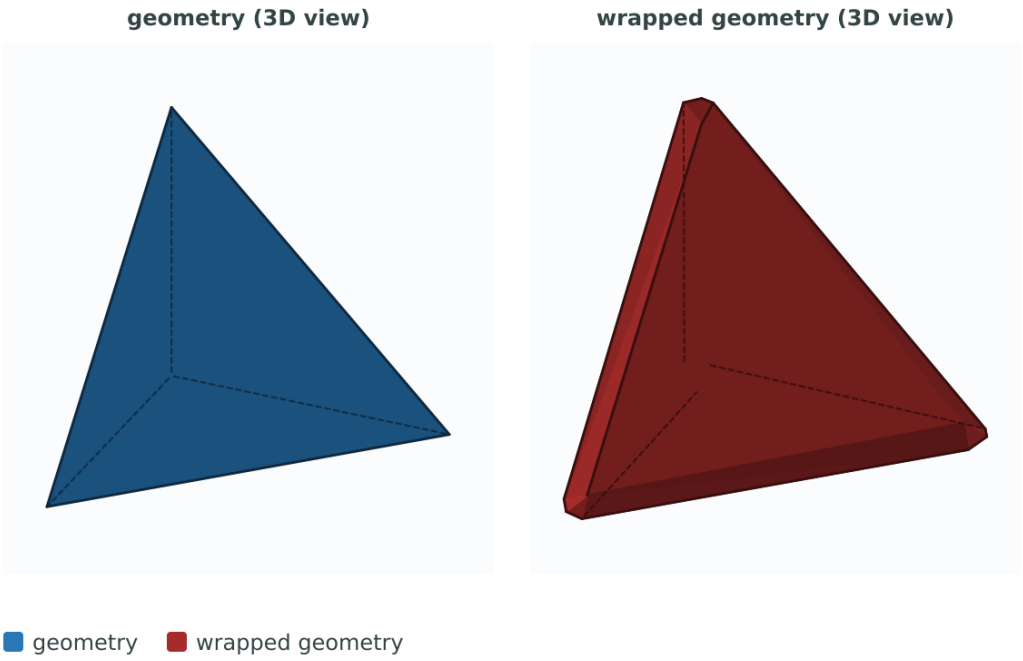
Examples

Wrap a tetrahedral surface and show the native 3D input beside the actual enclosing mesh.

Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z (
    ((0 0 0,4 0 0,0 4 0,0 0 0)),
    ((0 0 0,0 0 4,4 0 0,0 0 0)),
    ((0 0 0,0 4 0,0 0 4,0 0 0)),
    ((4 0 0,0 0 4,0 4 0,4 0 0))
  )'::geometry AS input_geometry
)
SELECT input_geometry AS input_geometry,
       CG_3DAlphaWrapping(input_geometry, 1)
         AS wrapped_geometry
FROM data;
```

Figure



See Also

CG_AlphaShape

8.3.48 CG_NurbsCurveInterpolate

CG_NurbsCurveInterpolate — Creates an interpolating NURBS curve passing through all given data points

Synopsis

geometry **CG_NurbsCurveInterpolate**(geometry data_points, integer degree);

Description

Creates a NURBS curve that interpolates through all provided data points. The resulting curve passes exactly through each point in the input geometry.

Parameters:

- `data_points` - A LINESTRING geometry containing the points to interpolate through.
- `degree` - Polynomial degree of the NURBS curve (typically 2 or 3).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

Interpolate a degree-3 NURBS curve through four 2D points.

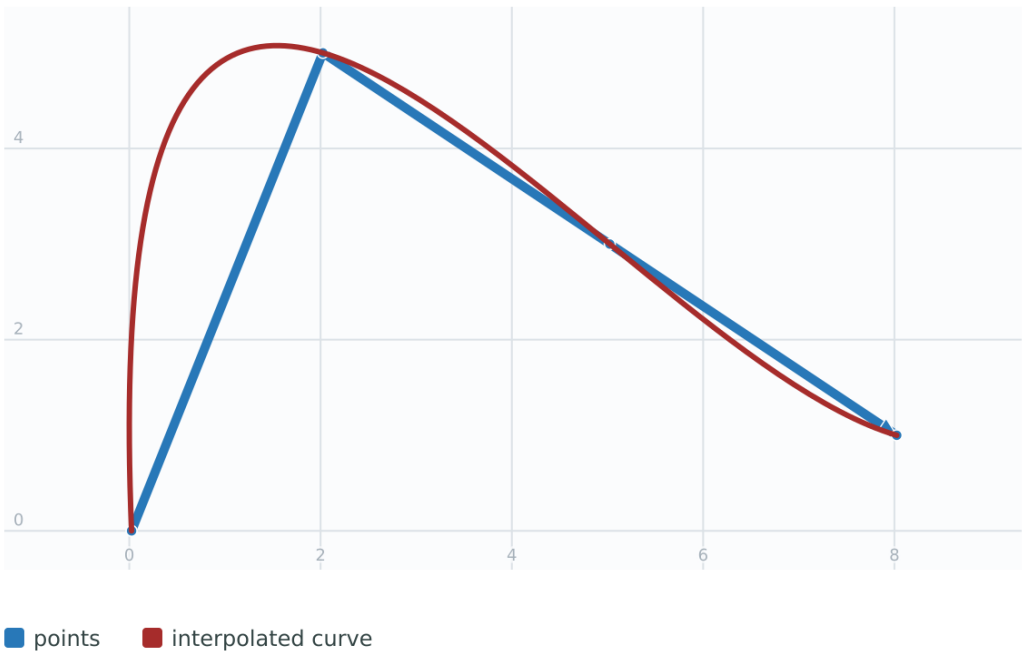
Code

```
WITH data AS (
  SELECT 'LINESTRING(0 0,2 5,5 3,8 1)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveInterpolate(input_points, 3) AS interpolated_curve
FROM data;
```

Output

```
LINESTRING(0 0,2 5,5 3,8 1) | NURBSCURVE(DEGREE 3, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT( ↩
    WEIGHTEDPOINT(-0.43 10.29),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(4.95 1.9),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(8 1), ↩
    WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(1,4)))
```

Figure



Interpolate a 3D degree-3 NURBS curve through the documented control points.

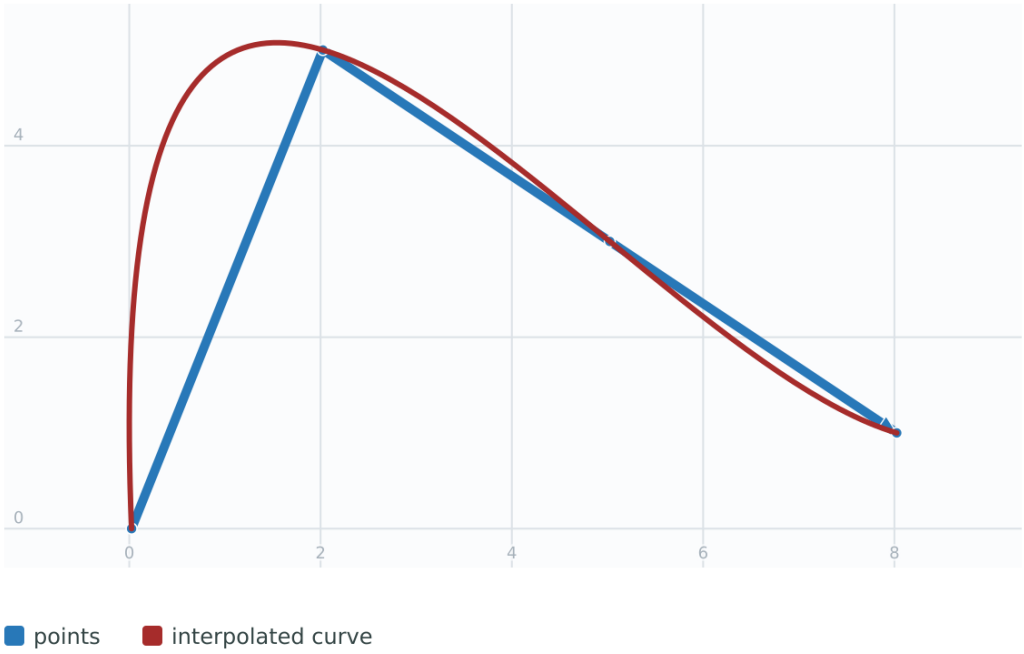
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (0 0 0,2 5 2,5 3 1,8 1 0)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveInterpolate(input_points, 3) AS interpolated_curve
FROM data;
```

Output

```
LINESTRING Z (0 0 0,2 5 2,5 3 1,8 1 0) | NURBSCURVE Z (DEGREE 3,CONTROLPOINTS Z (NURBSPOINT(WEIGHTEDPOINT Z (0 0 0), ↩
    WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (-0.43 10.29 4.36),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (4.95 1.9 0.54), ↩
    WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (8 1 0),WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(1,4)))
```

Figure



See Also

CG_NurbsCurveApproximate, ST_MakeNurbsCurve

8.3.49 CG_NurbsCurveApproximate

CG_NurbsCurveApproximate — Creates an approximating NURBS curve fitting data points within a tolerance

Synopsis

geometry **CG_NurbsCurveApproximate**(geometry data_points, integer degree, float8 tolerance, integer max_control_points=100);

Description

Creates a NURBS curve that approximates the given data points within the specified tolerance. Unlike interpolation, the curve does not necessarily pass through all points but provides a smooth fit with fewer control points.

Parameters:

- `data_points` - A LINESTRING geometry containing the points to approximate.
- `degree` - Polynomial degree of the NURBS curve.
- `tolerance` - Maximum allowed distance between the curve and data points.
- `max_control_points` - Optional maximum number of control points (default: 100). Limits curve complexity.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

Examples

Approximate noisy 2D data with a degree-2 curve and a tolerance of 0.5.

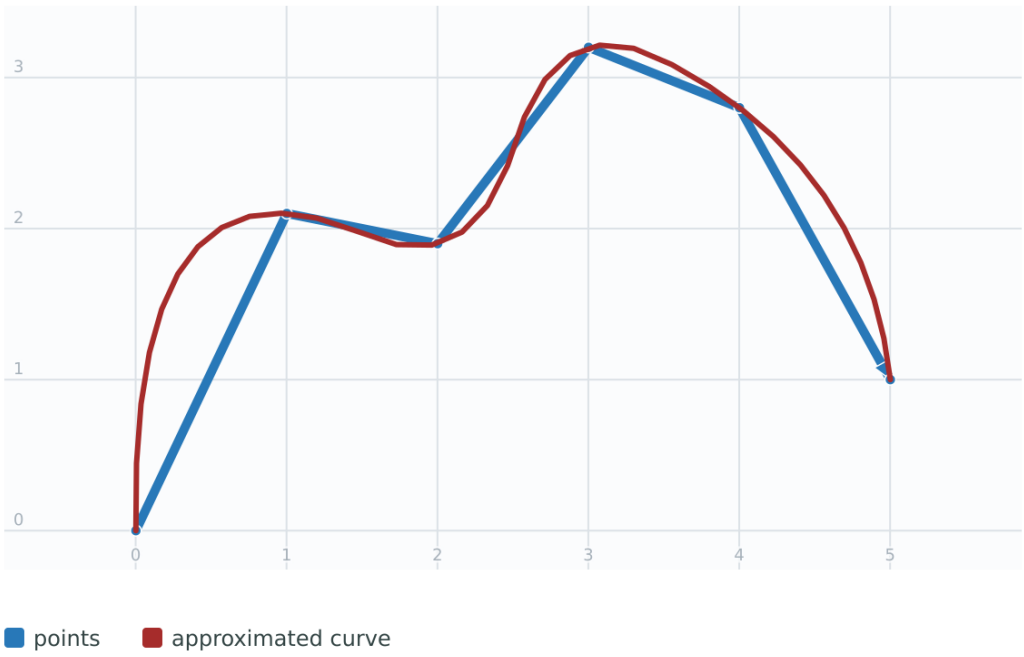
Code

```
WITH data AS (
  SELECT 'LINESTRING(0 0,1 2.1,2 1.9,3 3.2,4 2.8,5 1)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveApproximate(input_points, 2, 0.5) AS approximated_curve
FROM data;
```

Output

```
LINESTRING(0 0,1 2.1,2 1.9,3 3.2,4 2.8,5 1) | NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT
    1), NURBSPOINT(WEIGHTEDPOINT(-0.05 2.64),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(2.24 1.65),WEIGHT 1), NURBSPOINT(
    WEIGHTEDPOINT(2.79 3.48),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(4.84 2.43),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 1),
    WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(0.35,1), KNOT(0.51,1), KNOT(0.68,1), KNOT(1,3)))
```

Figure



Approximate a 3D point set while preserving the Z dimension in the fitted curve.

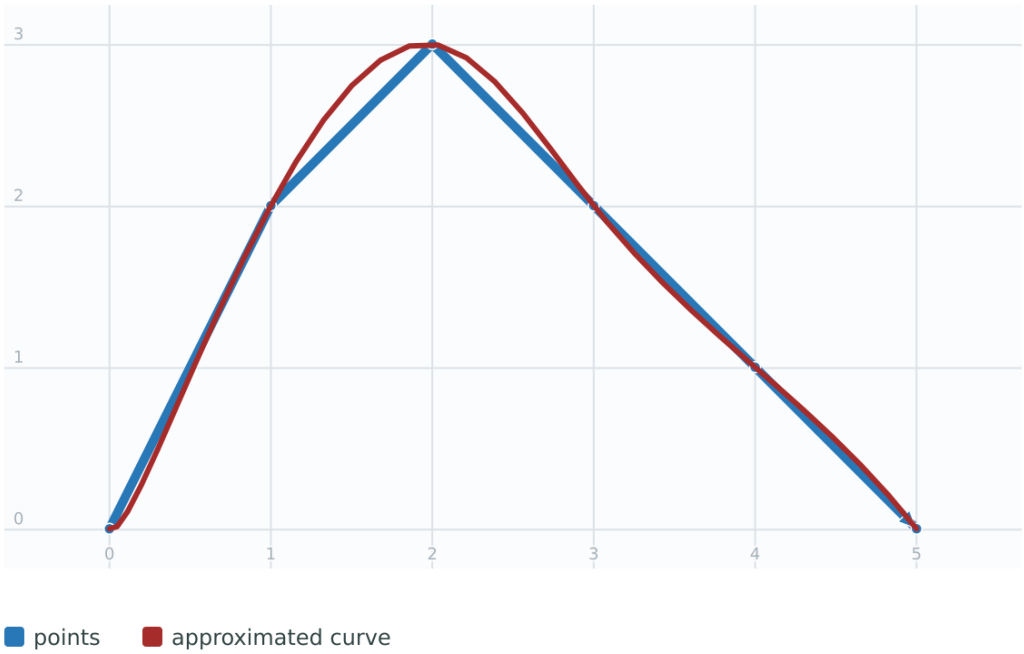
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (0 0 0,1 2 1,2 3 2,3 2 1,4 1 0,5 0 0)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveApproximate(input_points, 3, 0.1) AS approximated_curve
FROM data;
```

Output

```
LINESTRING Z (0 0 0,1 2 1,2 3 2,3 2 1,4 1 0,5 0 0) | NURBSCURVE Z (DEGREE 3,CONTROLPOINTS Z (NURBSPOINT(WEIGHTEDPOINT Z ↩
      (0 0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (0.18 -0.15 -1.29),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (1.49 4.36 ↩
      3.35),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (3.33 1.33 0.38),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (4.34 0.83  ↩
     -0.44),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (5 0 0),WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(0.46,1), KNOT(0.64,1), KNOT ↩
     (1,4)))
```

Figure



See Also

CG_NurbsCurveInterpolate, ST_MakeNurbsCurve

8.3.50 CG_NurbsCurveDerivative

CG_NurbsCurveDerivative — Computes the derivative (tangent, curvature) of a NURBS curve at a given parameter

Synopsis

geometry **CG_NurbsCurveDerivative**(geometry nurbscurve, float8 parameter, integer derivative_order);

Description

Computes the derivative of a NURBS curve at a specific parameter value. The derivative represents the rate of change at that point:

- Order 1: Tangent vector (direction and speed)
- Order 2: Curvature vector (how the tangent changes)
- Order 3: Third-order derivative (supported maximum)

Parameters:

- `nurbscurve` - A NURBS curve geometry.
- `parameter` - Parameter value for evaluation (typically 0.0 to 1.0).
- `derivative_order` - Order of derivative to compute (1, 2, or 3).

Returns a POINT geometry representing the derivative vector at the specified parameter.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

Examples

The first derivative gives the tangent vector at the evaluation parameter.

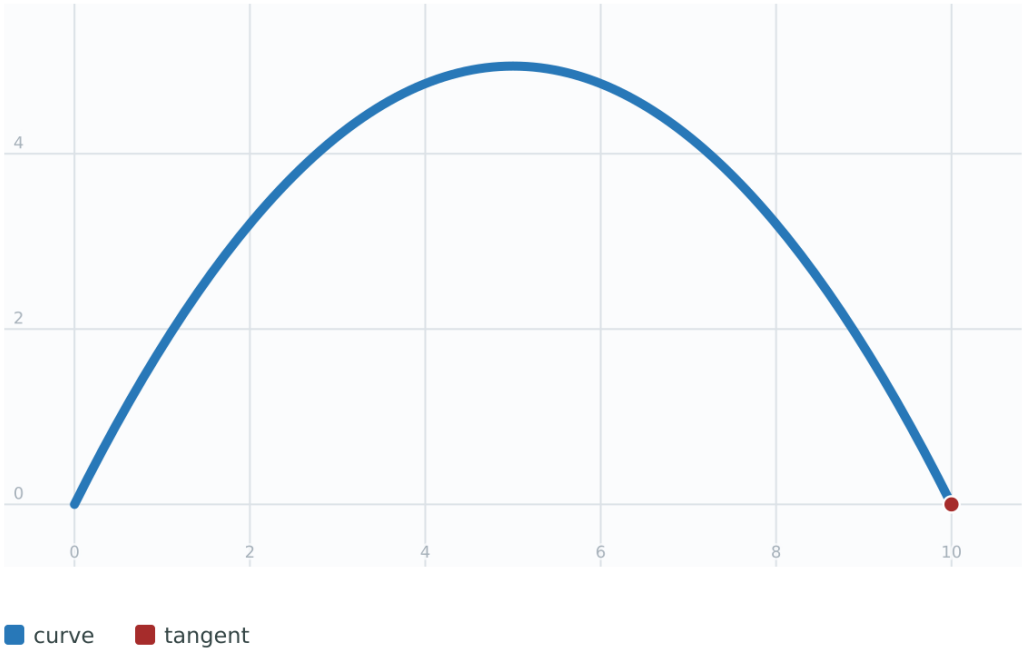
Code

```
WITH data AS (
  SELECT 'NURBSCURVE (2, (0.00 0.00, 5.00 10.00, 10.00 0.00))'::geometry AS input_curve
)
SELECT input_curve AS input_curve,
       CG_NurbsCurveDerivative(input_curve, 0.5, 1) AS tangent
FROM data;
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 10),WEIGHT 1),  ↩
    NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3))) | POINT Z (10 0 0)
```

Figure



The second derivative exposes curvature change for a higher-degree curve.

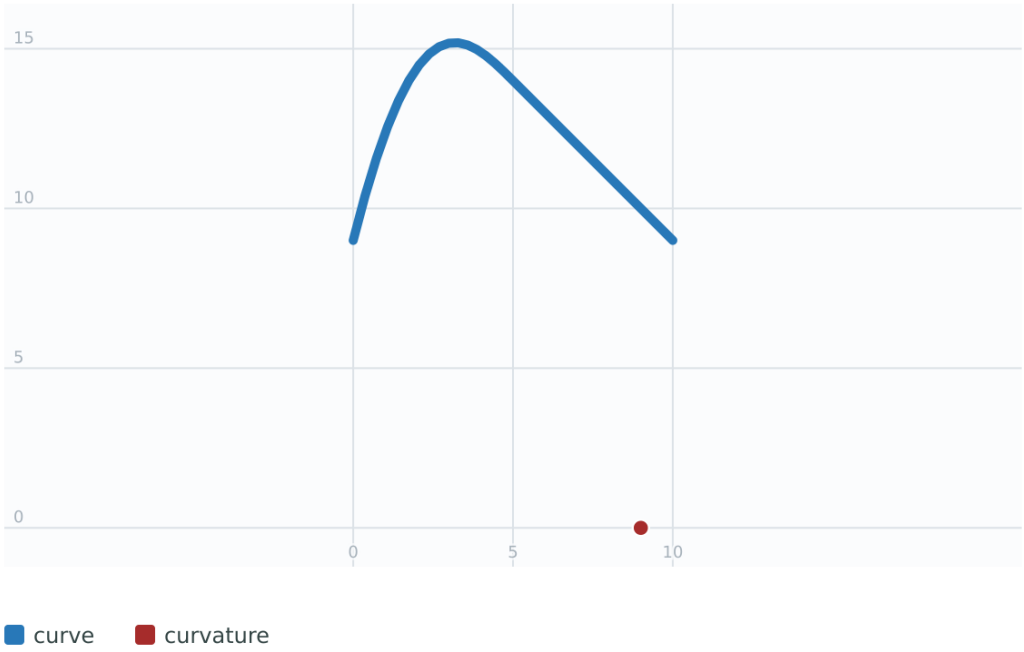
Code

```
WITH data AS (
  SELECT 'NURBSCURVE (3, (0.00 0.00, 2.00 8.00, 5.00 5.00, 8.00 2.00, 10.00 0.00))':: ↩
    geometry AS input_curve
)
SELECT input_curve AS input_curve,
       CG_NurbsCurveDerivative(input_curve, 0.5, 2) AS curvature
FROM data;
```

Output

```
NURBSCURVE(DEGREE 3, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(2 8),WEIGHT 1),  ↩
    NURBSPOINT(WEIGHTEDPOINT(5 5),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(8 2),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(10 0), ↩
    WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(0.5,1), KNOT(1,4))) | POINT Z (9 -9 0)
```

Figure



The same derivative call also works for 3D NURBS curves.

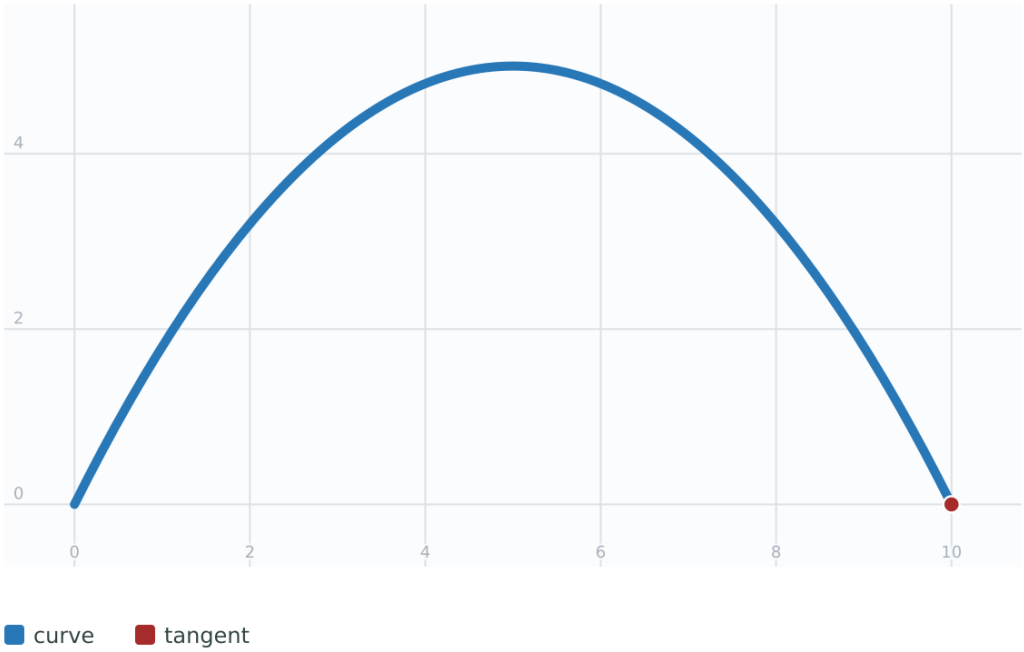
Code

```
WITH data AS (
  SELECT 'NURBSCURVE Z (2, (0.00 0.00 0.00, 5.00 10.00 5.00, 10.00 0.00 0.00))'::geometry AS ↩
     input_curve
)
SELECT input_curve AS input_curve,
       CG_NurbsCurveDerivative(input_curve, 0.5, 1) AS tangent
FROM data;
```

Output

```
NURBSCURVE Z (DEGREE 2,CONTROLPOINTS Z (NURBSPOINT(WEIGHTEDPOINT Z (0 0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (5 10 ↩
    5),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (10 0 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3))) | POINT Z (10 0 0)
```

Figure



See Also

ST_Evaluate, ST_MakeNurbsCurve

8.4 Deprecated SFCGAL Function Aliases

These `ST_` names remain available for compatibility and emit deprecation notices. Their entries contain only the legacy signature and migration link. Use the corresponding `CG_` pages for behavior, examples, and geometry support details.

8.4.1 ST_3DIntersection

ST_3DIntersection — Deprecated alias for `CG_3DIntersection`.

Synopsis

geometry **ST_3DIntersection**(geometry geom1, geometry geom2);

Description

Warning
`ST_3DIntersection` is deprecated as of 3.5.0. Use CG_3DIntersection instead.

Availability: 2.1.0

This method needs SFCGAL backend.

8.4.2 ST_3DDifference

ST_3DDifference — Deprecated alias for `CG_3DDifference`.

Synopsis

geometry **ST_3DDifference**(geometry geom1, geometry geom2);

Description



Warning
`ST_3DDifference` is deprecated as of 3.5.0. Use CG_3DDifference instead.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.3 ST_3DUnion

ST_3DUnion — Deprecated alias for `CG_3DUnion`.

Synopsis

geometry **ST_3DUnion**(geometry geom1, geometry geom2);
geometry **ST_3DUnion**(geometry set g1field);

Description



Warning
`ST_3DUnion` is deprecated as of 3.5.0. Use CG_3DUnion instead.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.4 ST_Tesselate

ST_Tesselate — Deprecated alias for `CG_Tesselate`.

Synopsis

geometry **ST_Tesselate**(geometry geom);

Description



Warning
`ST_Tesselate` is deprecated as of 3.5.0. Use CG_Tesselate instead.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.5 ST_3DArea

ST_3DArea — Deprecated alias for `CG_3DArea`.

Synopsis

float**ST_3DArea**(geometry geom1);

Description



Warning
`ST_3DArea` is deprecated as of 3.5.0. Use CG_3DArea instead.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.6 ST_Extrude

ST_Extrude — Deprecated alias for `CG_Extrude`.

Synopsis

geometry **ST_Extrude**(geometry geom, float x, float y, float z);

Description



Warning
`ST_Extrude` is deprecated as of 3.5.0. Use CG_Extrude instead.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.7 ST_ForceLHR

ST_ForceLHR — Deprecated alias for `CG_ForceLHR`.

Synopsis

geometry **ST_ForceLHR**(geometry geom);

Description



Warning
`ST_ForceLHR` is deprecated as of 3.5.0. Use CG_ForceLHR instead.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.8 ST_Orientation

ST_Orientation — Deprecated alias for `CG_Orientation`.

Synopsis

integer **ST_Orientation**(geometry geom);

Description



Warning
`ST_Orientation` is deprecated as of 3.5.0. Use CG_Orientation instead.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.9 ST_MinkowskiSum

ST_MinkowskiSum — Deprecated alias for `CG_MinkowskiSum`.

Synopsis

geometry **ST_MinkowskiSum**(geometry geom1, geometry geom2);

Description



Warning
`ST_MinkowskiSum` is deprecated as of 3.5.0. Use CG_MinkowskiSum instead.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.10 ST_StraightSkeleton

ST_StraightSkeleton — Deprecated alias for `CG_StraightSkeleton`.

Synopsis

geometry **ST_StraightSkeleton**(geometry geom);

Description



Warning
`ST_StraightSkeleton` is deprecated as of 3.1.0. Use CG_StraightSkeleton instead.



Note
Like `CG_StraightSkeleton`, this function ignores the Z dimension and always gives a 2D result, even when used on a 3D geometry.

Availability: 2.1.0


This method needs SFCGAL backend.

8.4.11 ST_ApproximateMedialAxis

ST_ApproximateMedialAxis — Deprecated alias for `CG_ApproximateMedialAxis`.

Synopsis

geometry **ST_ApproximateMedialAxis**(geometry geom);

Description



Warning
`ST_ApproximateMedialAxis` is deprecated as of 3.5.0. Use CG_ApproximateMedialAxis instead.



Note
Like `CG_ApproximateMedialAxis`, this function ignores the Z dimension and always gives a 2D result, even when used on a 3D geometry.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.12 ST_IsPlanar

ST_IsPlanar — Deprecated alias for `CG_IsPlanar`.

Synopsis

boolean **ST_IsPlanar**(geometry geom);

Description



Warning
`ST_IsPlanar` is deprecated as of 3.5.0. Use CG_IsPlanar instead.

Availability: 2.2.0: This was documented in 2.1.0 but got accidentally left out in 2.1 release.


This method needs SFCGAL backend.

8.4.13 ST_Volume

ST_Volume — Deprecated alias for `CG_Volume`.

Synopsis

float **ST_Volume**(geometry geom1);

Description



Warning
`ST_Volume` is deprecated as of 3.5.0. Use CG_Volume instead.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.14 ST_MakeSolid

ST_MakeSolid — Deprecated alias for `CG_MakeSolid`.

Synopsis

geometry **ST_MakeSolid**(geometry geom1);

Description



Warning
`ST_MakeSolid` is deprecated as of 3.5.0. Use CG_MakeSolid instead.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.15 ST_IsSolid

ST_IsSolid — Deprecated alias for `CG_IsSolid`.

Synopsis

boolean **ST_IsSolid**(geometry geom1);

Description



Warning
`ST_IsSolid` is deprecated as of 3.5.0. Use CG_IsSolid instead.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.16 ST_ConstrainedDelaunayTriangles

ST_ConstrainedDelaunayTriangles — Deprecated alias for `CG_ConstrainedDelaunayTriangles`.

Synopsis

geometry **ST_ConstrainedDelaunayTriangles**(geometry g1);

Description



Warning
`ST_ConstrainedDelaunayTriangles` is deprecated as of 3.5.0. Use CG_ConstrainedDelaunayTriangles instead.

Availability: 3.0.0


This method needs SFCGAL backend.

8.4.17 ST_3DConvexHull

ST_3DConvexHull — Deprecated alias for `CG_3DConvexHull`.

Synopsis

geometry **ST_3DConvexHull**(geometry geom1);

Description



Warning
`ST_3DConvexHull` is deprecated as of 3.5.0. Use CG_3DConvexHull instead.

Availability: 3.3.0


This method needs SFCGAL backend.

8.4.18 ST_AlphaShape

ST_AlphaShape — Deprecated alias for `CG_AlphaShape`.

Synopsis

geometry **ST_AlphaShape**(geometry geom, float alpha , boolean allow_holes = false);

Description



Warning
`ST_AlphaShape` is deprecated as of 3.5.0. Use CG_AlphaShape instead.


This method needs SFCGAL backend.

8.4.19 ST_OptimalAlphaShape

ST_OptimalAlphaShape — Deprecated alias for `CG_OptimalAlphaShape`.

Synopsis

geometry **ST_OptimalAlphaShape**(geometry geom, boolean allow_holes = false , integer nb_components = 1);

Description



Warning
`ST_OptimalAlphaShape` is deprecated as of 3.5.0. Use CG_OptimalAlphaShape instead.

Availability: 3.3.0 - requires SFCGAL >= 1.4.1.


This method needs SFCGAL backend.

Chapter 9

Topology

The PostGIS Topology types and functions are used to manage topological objects such as faces, edges and nodes.

Sandro Santilli's presentation at PostGIS Day Paris 2011 conference gives a good synopsis of PostGIS Topology and where it is headed Topology with PostGIS 2.0 slide deck.

Vincent Picavet provides a good synopsis and overview of what is Topology, how is it used, and various FOSS4G tools that support it in PostGIS Topology PGConf EU 2012.

An example of a topologically based GIS database is the US Census Topologically Integrated Geographic Encoding and Referencing System (TIGER) database. If you want to experiment with PostGIS topology and need some data.

The PostGIS topology module has existed for a long time but was not always part of the official documentation. Extensive cleanup removed deprecated functions, fixed known usability issues, documented the features and functions, added new functionality, and improved SQL-MM compliance.

This chapter is the maintained reference for the topology module, including its primitive tables, TopoGeometry type, topology management functions, editing functions, validation functions, and conversion helpers.

All functions and tables associated with this module are installed in a schema called `topology`.

Functions that are defined in SQL/MM standard are prefixed with ST_ and functions specific to PostGIS are not prefixed.

Topology support is built by default and can be disabled by specifying the --without-topology configure option at build time as described in Chapter 2

9.1 Building Faces from Linework

This example builds topology faces from noded linework and associates point attributes with the resulting faces. It uses ST_CreateTopoGeo to populate an empty topology, ValidateTopology to check the result, GetFaceByPoint to locate each attribute point, and ST_GetFaceGeometry to materialize ordinary polygon geometries.

Create an empty topology and populate it with linework that encloses two faces:

Code

```
CREATE EXTENSION IF NOT EXISTS postgis_topology;

SELECT topology.CreateTopology('topo_example', 3857);

WITH lines(geom) AS (VALUES
    ('SRID=3857;LINESTRING(0 0, 2 0)'::geometry),
    ('SRID=3857;LINESTRING(0 1, 2 1)'::geometry),
    ('SRID=3857;LINESTRING(0 0, 0 1)'::geometry),
    ('SRID=3857;LINESTRING(1 0, 1 1)'::geometry),
    ('SRID=3857;LINESTRING(2 0, 2 1)'::geometry)
```

```
)
SELECT topology.ST_CreateTopoGeo(
    'topo_example',
    ST_Collect(geom)
)
FROM lines;

SELECT * FROM topology.ValidateTopology('topo_example');
```

An empty result from `ValidateTopology` means that no topology errors were found. Attribute points can now be matched to faces. A face identifier of `0` denotes the universal face, so the point is outside every bounded face.

Code

```
WITH labels(name, geom) AS (VALUES
    ('west', 'SRID=3857;POINT(0.5 0.5)'::geometry),
    ('east', 'SRID=3857;POINT(1.5 0.5)'::geometry),
    ('outside', 'SRID=3857;POINT(3 0.5)'::geometry)
),
located AS (
    SELECT name,
           topology.GetFaceByPoint('topo_example', geom, 0) AS face_id
    FROM labels
)
SELECT name,
       face_id,
       CASE WHEN face_id = 0 THEN NULL
            ELSE topology.ST_GetFaceGeometry('topo_example', face_id)
       END AS geom
FROM located
ORDER BY name;
```

Before copying attributes into a face table, check both for points outside bounded faces and for multiple source points assigned to the same face:

Code

```
WITH labels(name, geom) AS (VALUES
    ('west', 'SRID=3857;POINT(0.5 0.5)'::geometry),
    ('east', 'SRID=3857;POINT(1.5 0.5)'::geometry),
    ('outside', 'SRID=3857;POINT(3 0.5)'::geometry)
),
located AS (
    SELECT name,
           topology.GetFaceByPoint('topo_example', geom, 0) AS face_id
    FROM labels
)
SELECT 'outside face' AS problem,
       array_agg(name ORDER BY name) AS labels
FROM located
WHERE face_id = 0
HAVING count(*) > 0
UNION ALL
SELECT 'duplicate face ' || face_id,
       array_agg(name ORDER BY name)
FROM located
WHERE face_id != 0
GROUP BY face_id
HAVING count(*) > 1;
```

Use DropTopology to remove the example topology when it is no longer needed:

Code

```
SELECT topology.DropTopology('topo_example');
```

9.2 Topology Primitive Tables

The core primitives of any topology are stored in the `edge_data`, `node`, and `face` tables that live in the schema created by CreateTopology. Each row of `edge_data` represents an oriented edge: it records a directed curve from `start_node` to `end_node` together with the identifier of the face encountered on the left of that direction (`left_face`) and the face encountered on the right (`right_face`). The same geometric segment may therefore appear twice—once for each orientation—when it belongs to two faces.

The `next_left_edge` and `next_right_edge` columns complete this orientation information by encoding how to keep walking around a face. They store signed integers whose absolute value is the identifier of the next oriented edge and whose sign determines whether the stored orientation has to be followed as-is or reversed when traversing. Formally, the following rules hold for every edge `e`:

- `abs(next_left_edge)` is the identifier of the edge reached by continuing around the face that lies to the left of `e`. If the value is positive the walk continues from the end node of `e` along the stored orientation of the referenced edge; if it is negative the referenced edge must be followed backwards so that the shared face remains on the walker's left.
- `abs(next_right_edge)` analogously follows the boundary of the face located on the right of `e`. A positive value means that the next edge is taken with its recorded orientation starting from the end node of `e`, whereas a negative value instructs to traverse the referenced edge in reverse, starting from its end node, so that the right-hand face is preserved.
- A zero value indicates that the edge is dangling on the corresponding side (for example an isolated edge whose incident face is the universal face `0`). The `abs_next_left_edge` and `abs_next_right_edge` columns exposed by the `edge` view are convenience projections of these absolute values.

This representation is a variant of a doubly connected edge list and is exploited by many topology routines. Functions such as GetRingEdges and ValidateTopology rely on it to reconstruct face boundaries and to diagnose inconsistencies—hence the "invalid next_left_edge" and "invalid next_right_edge" diagnostics reported during validation. Constructors like AddEdge initialise the `next_*` attributes with trivial self references, while editing routines including ST_AddEdgeModFace and ST_RemEdgeModFace update the links as edges are inserted or removed. Other bulk operations (for example Polygonize) may intentionally leave the fields unset, which is why the documentation flags their behaviour explicitly.

9.3 Topology Types

9.3.1 getfaceedges_returntype

getfaceedges_returntype — A composite type that consists of a sequence number and an edge number.

Description

A composite type that consists of a sequence number and an edge number. This is the return type for `ST_GetFaceEdges` and `GetNodeEdges` functions.

1. `sequence` is an integer: Refers to a topology defined in the topology.topology table which defines the topology schema and srid.
2. `edge` is an integer: The identifier of an edge.

9.3.2 TopoGeometry

TopoGeometry — A composite type representing a topologically defined geometry.

Description

A composite type that refers to a topology geometry in a specific topology layer, having a specific type and a specific id. The elements of a TopoGeometry are the properties: topology_id, layer_id, id integer, type integer.

1. `topology_id` is an integer: Refers to a topology defined in the topology.topology table which defines the topology schema and srid.
2. `layer_id` is an integer: The layer_id in the layers table that the TopoGeometry belongs to. The combination of topology_id, layer_id provides a unique reference in the topology.layers table.
3. `id` is an integer: The id is the autogenerated sequence number that uniquely defines the topogeometry in the respective topology layer.
4. `type` integer between 1 - 4 that defines the geometry type: 1:[multi]point, 2:[multi]line, 3:[multi]poly, 4:collection

Casting Behavior

This section lists the automatic as well as explicit casts allowed for this data type

Cast To	Behavior
geometry	automatic

See Also

CreateTopoGeom

9.3.3 validatetopology_returntype

validatetopology_returntype — A composite type that consists of an error message and id1 and id2 to denote location of error. This is the return type for `ValidateTopology`.

Description

A composite type that consists of an error message and two integers. The ValidateTopology function returns a set of these to denote validation errors and the id1 and id2 to denote the ids of the topology objects involved in the error.

Field	Type	Description
`error`	varchar	Denotes the type of error. Current error descriptors are: coincident nodes, edge crosses node, edge not simple, edge end node geometry mismatch, edge start node geometry mismatch, face overlaps face, face within face.
`id1`	integer	Denotes the identifier of the edge, face, or node in error.
`id2`	integer	For errors that involve two objects, denotes the secondary edge or node.

See Also

ValidateTopology

9.4 Topology Domains

9.4.1 TopoElement

TopoElement — An array of 2 integers generally used to identify a TopoGeometry component.

Description

An array of 2 integers used to represent one component of a simple or hierarchical TopoGeometry.

In the case of a simple TopoGeometry the first element of the array represents the identifier of a topological primitive and the second element represents its type (1:node, 2:edge, 3:face). In the case of a hierarchical TopoGeometry the first element of the array represents the identifier of a child TopoGeometry and the second element represents its layer identifier.



Note
For any given hierarchical TopoGeometry all child TopoGeometry elements will come from the same child layer, as specified in the topology.layer record for the layer of the TopoGeometry being defined.

Examples

Code

```
SELECT te[1] AS id, te[2] AS type FROM
( SELECT ARRAY[1, 2]::topology.topoelement AS te ) f;
```

Output

```
 id | type
----+------
  1 |    2
```

Code

```
SELECT ARRAY[1, 2]::topology.topoelement;
```

Output

```
  te
-------
 {1,2}
```

This example shows what happens when casting a three-element array to topology.topoelement. The domain requires a two-element array, so the dimension check fails.

Code

```
SELECT ARRAY[1, 2, 3]::topology.topoelement;
```

Output

```
ERROR:  value for domain topology.topoelement violates check constraint "dimensions"
```

See Also

GetTopoGeomElements, TopoElementArray, TopoGeometry, TopoGeom_addElement, TopoGeom_remElement

9.4.2 TopoElementArray

TopoElementArray — An array of TopoElement objects.

Description

An array of 1 or more TopoElement objects, generally used to pass around components of TopoGeometry objects.

Examples

Code

```
SELECT '{{1,2},{4,3}}'::topology.topoelementarray AS tea;
```

Output

```
{{1,2},{4,3}}
```

This is the more verbose equivalent.

Code

```
SELECT ARRAY[ARRAY[1, 2], ARRAY[4, 3]]::topology.topoelementarray AS tea;
```

Output

```
{{1,2},{4,3}}
```

This example uses the array aggregate function packaged with topology.

Code

```
SELECT topology.TopoElementArray_Agg(ARRAY[e, t] ORDER BY e, t) As tea
FROM generate_series(1, 4) As e CROSS JOIN generate_series(1, 3) As t;
```

Output

```
{{1,1},{1,2},{1,3},{2,1},{2,2},{2,3},{3,1},{3,2},{3,3},{4,1},{4,2},{4,3}}
```

Code

```
SELECT '{{1,2,4},{3,4,5}}'::topology.topoelementarray AS tea;
```

Output

```
ERROR:  value for domain topology.topoelementarray violates check constraint "dimensions"
```

See Also

TopoElement, GetTopoGeomElementArray, TopoElementArray_Agg

9.5 Topology and TopoGeometry Management

9.5.1 AddTopoGeometryColumn

AddTopoGeometryColumn — Adds a topogeometry column to an existing table, registers this new column as a layer in topology.layer and returns the new layer_id.

Synopsis

integer **AddTopoGeometryColumn**(name topology_name, name schema_name, name table_name, name column_name, varchar feature_type, integer child_layer);
integer **AddTopoGeometryColumn**(name topology_name, regclass tab, name column_name, integer layer_id, varchar feature_type, integer child_layer);

Description

Each TopoGeometry object belongs to a specific Layer of a specific Topology. Before creating a TopoGeometry object you need to create its TopologyLayer. A Topology Layer is an association of a feature-table with the topology. It also contain type and hierarchy information. We create a layer using the AddTopoGeometryColumn() function:

This function will both add the requested column to the table and add a record to the topology.layer table with all the given info.

If you don't specify [child_layer] (or set it to NULL) this layer would contain Basic TopoGeometries (composed by primitive topology elements). Otherwise this layer will contain hierarchical TopoGeometries (composed by TopoGeometries from the child_layer).

Once the layer is created (its id is returned by the AddTopoGeometryColumn function) you're ready to construct TopoGeometry objects in it

Valid `feature_types` are: POINT, MULTIPOINT, LINE, MULTILINE, POLYGON, MULTIPOLYGON, COLLECTION

Availability: 1.1

Examples

This example creates the new table in the `ma_topo` schema. The table could be created in a different schema, in which case `topology_name` and `schema_name` would be different.

Code

```
CREATE SCHEMA ma;
CREATE TABLE ma.parcels(gid serial, parcel_id varchar(20) PRIMARY KEY, address text);
SELECT topology.AddTopoGeometryColumn('ma_topo', 'ma', 'parcels', 'topo', 'POLYGON');
```

Code

```
CREATE SCHEMA ri;
CREATE TABLE ri.roads(gid serial PRIMARY KEY, road_name text);
SELECT topology.AddTopoGeometryColumn('ri_topo', 'ri', 'roads', 'topo', 'LINE');
```

See Also

DropTopoGeometryColumn, toTopoGeom, CreateTopology, CreateTopoGeom

9.5.2 RenameTopoGeometryColumn

RenameTopoGeometryColumn — Renames a topogeometry column

Synopsis

topology.layer **RenameTopoGeometryColumn**(regclass layer_table, name feature_column, name new_name);

Description

This function changes the name of an existing TopoGeometry column ensuring metadata information about it is updated accordingly.

Availability: 3.4.0

Examples

Code

```
SELECT topology.RenameTopoGeometryColumn('public.parcels', 'topogeom', 'tgeom');
```

See Also

AddTopoGeometryColumn, RenameTopology

9.5.3 DropTopology

DropTopology — Use with caution: Drops a topology schema and deletes its reference from topology.topology table and references to tables in that schema from the geometry_columns table.

Synopsis

integer **DropTopology**(varchar topology_schema_name);

Description

Drops a topology schema and deletes its reference from topology.topology table and references to tables in that schema from the geometry_columns table. This function should be USED WITH CAUTION, as it could destroy data you care about. If the schema does not exist, it just removes reference entries the named schema.

Availability: 1.1

Examples

Cascade drops the ma_topo schema and removes all references to it in topology.topology and geometry_columns.

Code

```
SELECT topology.DropTopology('ma_topo');
```

See Also

DropTopoGeometryColumn

9.5.4 RenameTopology

RenameTopology — Renames a topology

Synopsis

varchar **RenameTopology**(varchar old_name, varchar new_name);

Description

Renames a topology schema, updating its metadata record in the `topology.topology` table.

Availability: 3.4.0

Examples

Rename a topology from `topo_stage` to `topo_prod`.

Code

```
SELECT topology.RenameTopology('topo_stage', 'topo_prod');
```

See Also

CopyTopology, RenameTopoGeometryColumn

9.5.5 DropTopoGeometryColumn

DropTopoGeometryColumn — Drops the topogeometry column from the table named `table_name` in schema `schema_name` and unregisters the columns from topology.layer table.

Synopsis

text **DropTopoGeometryColumn**(varchar schema_name, varchar table_name, varchar column_name);

Description

Drops the topogeometry column from the table named `table_name` in schema `schema_name` and unregisters the columns from topology.layer table. Returns summary of drop status. NOTE: it first sets all values to NULL before dropping to bypass referential integrity checks.

Availability: 1.1

Examples

Code

```
SELECT topology.DropTopoGeometryColumn('ma_topo', 'parcel_topo', 'topo');
```

See Also

AddTopoGeometryColumn

9.5.6 FixCorruptTopoGeometryColumn

FixCorruptTopoGeometryColumn — Fixes topogeometry corruption caused by upgrade to postgis_topology 3.6.0 and higher

Synopsis

text **FixCorruptTopoGeometryColumn**(name layerSchema, name layerTable, name layerColumn);

Description

When upgrading from PostGIS topology <3.6.0 to version >3.6.0+, the topogeometry column definition was changed. This caused corruption in topogeometries created before the upgrade. This function fixes this corruption in affected tables.

The function refuses to process tables with expression or partial indexes; drop or rebuild those indexes manually around the repair.

Availability: 3.6.1

Examples

Fix all topology columns

Code

```
SELECT topology.FixCorruptTopoGeometryColumn(schema_name, table_name, feature_column)
    FROM topology.layer;
```

See Also

UpgradeTopology

9.5.7 Populate_Topology_Layer

Populate_Topology_Layer — Adds missing entries to topology.layer table by reading metadata from topo tables.

Synopsis

setof record **Populate_Topology_Layer**();

Description

Adds missing entries to the `topology.layer` table by inspecting topology constraints on tables. This function is useful for fixing up entries in topology catalog after restores of schemas with topo data.

It returns the list of entries created. Returned columns are `schema_name`, `table_name`, `feature_column`.

Availability: 2.3.0

Examples

Code

```
SELECT CreateTopology('strk_topo');
CREATE SCHEMA strk;
CREATE TABLE strk.parcels(gid serial, parcel_id varchar(20) PRIMARY KEY, address text);
SELECT topology.AddTopoGeometryColumn('strk_topo', 'strk', 'parcels', 'topo', 'POLYGON');
```

Output

```
 createtopology
----------------
              1
(1 row)

CREATE SCHEMA
CREATE TABLE
 addtopogeometrycolumn
-----------------------
                     1
(1 row)
```

This query returns no records because the feature is already registered.

Code

```
SELECT *
FROM topology.Populate_Topology_Layer();
```

Output

```
(0 rows)
```

Rebuild the topology layer catalog.

Code

```
TRUNCATE TABLE topology.layer;
```

Output

```
TRUNCATE TABLE
```

Code

```
SELECT *
FROM topology.Populate_Topology_Layer();
```

Output

```
 schema_name | table_name | feature_column
-------------+------------+----------------
 strk        | parcels    | topo
(1 row)
```

Code

```
SELECT topology_id, layer_id,
       schema_name AS sn, table_name AS tn, feature_column AS fc
FROM topology.layer;
```

Output

```
 topology_id | layer_id |  sn  |   tn    |  fc
-------------+----------+------+---------+------
           1 |        1 | strk | parcels | topo
(1 row)
```

See Also

AddTopoGeometryColumn

9.5.8 TopologySummary

TopologySummary — Takes a topology name and provides summary totals of types of objects in topology.

Synopsis

text **TopologySummary**(varchar topology_schema_name);

Description

Takes a topology name and provides summary totals of types of objects in topology.

Availability: 2.0.0

Examples

Code

```
SELECT topology.TopologySummary('city_data');
```

Output

```
Topology city_data (329), SRID 4326, precision: 0
22 nodes, 24 edges, 10 faces, 29 topogeoms in 5 layers
Layer 1, type Polygonal (3), 9 topogeoms
 Deploy: features.land_parcels.feature
Layer 2, type Puntal (1), 8 topogeoms
 Deploy: features.traffic_signs.feature
Layer 3, type Lineal (2), 8 topogeoms
 Deploy: features.city_streets.feature
Layer 4, type Polygonal (3), 3 topogeoms
 Hierarchy level 1, child layer 1
```

```
 Deploy: features.big_parcels.feature
Layer 5, type Puntal (1), 1 topogeoms
 Hierarchy level 1, child layer 2
 Deploy: features.big_signs.feature
```

9.5.9 ValidateTopology

ValidateTopology — Returns a set of validatetopology_returntype objects detailing issues with topology.

Synopsis

setof validatetopology_returntype **ValidateTopology**(varchar toponame, geometry bbox);

Description

Returns a set of validatetopology_returntype objects detailing issues with topology, optionally limiting the check to the area specified by the `bbox` parameter.

List of possible errors, what they mean and what the returned ids represent are displayed below:

Error	id1	id2	Meaning
coincident nodes	Identifier of first node.	Identifier of second node.	Two nodes have the same geometry.
edge crosses node	Identifier of the edge.	Identifier of the node.	An edge has a node in its interior. See ST_Relate.
invalid edge	Identifier of the edge.		An edge geometry is invalid. See ST_IsValid.
edge not simple	Identifier of the edge.		An edge geometry has self-intersections. See ST_IsSimple.
edge crosses edge	Identifier of first edge.	Identifier of second edge.	Two edges have an interior intersection. See ST_Relate.
edge start node geometry mismatch	Identifier of the edge.	Identifier of the indicated start node.	The geometry of the node indicated as the starting node for an edge does not match the first point of the edge geometry. See ST_StartPoint.
edge end node geometry mismatch	Identifier of the edge.	Identifier of the indicated end node.	The geometry of the node indicated as the ending node for an edge does not match the last point of the edge geometry. See ST_EndPoint.
face without edges	Identifier of the orphaned face.		No edge reports an existing face on either of its sides (left_face, right_face).
face has no rings	Identifier of the partially-defined face.		Edges reporting a face on their sides do not form a ring.

Error	id1	id2	Meaning
face has wrong mbr	Identifier of the face with wrong mbr cache.		Minimum bounding rectangle of a face does not match minimum bounding box of the collection of edges reporting the face on their sides.
hole not in advertised face	Signed identifier of an edge, identifying the ring. See GetRingEdges.		A ring of edges reporting a face on its exterior is contained in different face.
not-isolated node has not-containing_face	Identifier of the ill-defined node.		A node which is reported as being on the boundary of one or more edges is indicating a containing face.
isolated node has containing_face	Identifier of the ill-defined node.		A node which is not reported as being on the boundary of any edges is lacking the indication of a containing face.
isolated node has wrong containing_face	Identifier of the misrepresented node.		A node which is not reported as being on the boundary of any edges indicates a containing face which is not the actual face containing it. See GetFaceContainingPoint.
invalid next_right_edge	Identifier of the misrepresented edge.	Signed id of the edge which should be indicated as the next right edge.	The edge indicated as the next edge encountered walking on the right side of an edge is wrong.
invalid next_left_edge	Identifier of the misrepresented edge.	Signed id of the edge which should be indicated as the next left edge.	The edge indicated as the next edge encountered walking on the left side of an edge is wrong.
mixed face labeling in ring	Signed identifier of an edge, identifying the ring. See GetRingEdges.		Edges in a ring indicate conflicting faces on the walking side. This is also known as a "Side Location Conflict".
non-closed ring	Signed identifier of an edge, identifying the ring. See GetRingEdges.		A ring of edges formed by following next_left_edge/next_right_edge attributes starts and ends on different nodes.
face has multiple shells	Identifier of the contended face.	Signed identifier of an edge, identifying the ring. See GetRingEdges.	More than a one ring of edges indicate the same face on its interior.

Availability: 1.0.0

Enhanced: 2.0.0 more efficient edge crossing detection and fixes for false positives that were existent in prior versions.

Changed: 2.2.0 values for id1 and id2 were swapped for 'edge crosses node' to be consistent with error description.

Changed: 3.2.0 added optional bbox parameter, perform face labeling and edge linking checks.

Examples

Code

```
SELECT * FROM topology.ValidateTopology('ma_topo');
```

Output

```
      error         | id1 | id2
--------------------+-----+-----
face without edges |   1 |
```

See Also

validatetopology_returntype

9.5.10 ValidateTopologyRelation

ValidateTopologyRelation — Returns info about invalid topology relation records

Synopsis

setof record **ValidateTopologyRelation**(varchar toponame);

Description

Returns a set records giving information about invalidities in the relation table of the topology.

Availability: 3.2.0

See Also

ValidateTopology

9.5.11 ValidateTopologyPrecision

ValidateTopologyPrecision — Returns non-precise vertices in the topology.

Synopsis

geometry **ValidateTopologyPrecision**(name toponame, geometry bbox, float8 gridSize);

Description

Returns all vertices that are not rounded to the topology or given `gridSize` as a puntal geometry, optionally limiting the check to the area specified by the `bbox` parameter.

Availability: 3.6.0

Examples

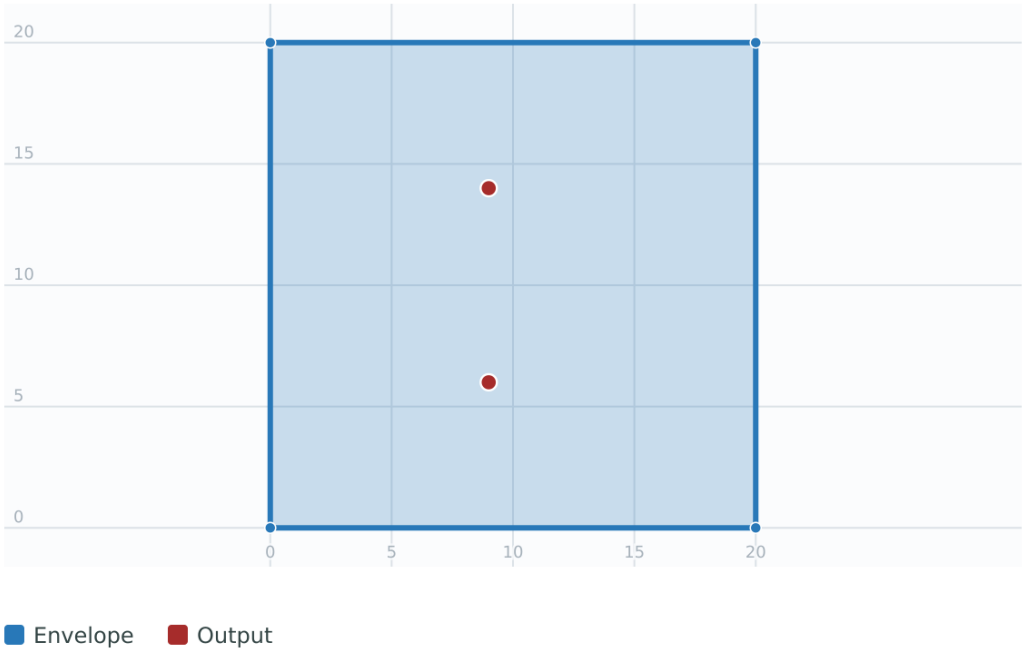
Code

```
SELECT g
FROM topology.ValidateTopologyPrecision(
    'city_data',
    gridSize => 2,
    bbox => ST_MakeEnvelope(0, 0, 20, 20)
) g;
```

Output

```
MULTIPOINT(9 6,9 14)
```

Figure



See Also

MakeTopologyPrecise

9.5.12 MakeTopologyPrecise

MakeTopologyPrecise — Snap topology vertices to precision grid.

Synopsis

void **MakeTopologyPrecise**(name toponame, geometry bbox, float8 gridSize);

Description

Snaps all vertices of a topology to the topology precision grid or to the grid whose size is specified with the `gridSize` parameter, optionally limiting the operation to the objects intersecting the area specified by the `bbox` parameter.

Using a `gridSize` larger than the smallest edge extent raises an exception rather than collapsing the edge into an empty geometry. This safeguards adjacent topological relationships by preventing precision snapping from removing primitives altogether.



Note
Snapping could make the topology invalid, so it is recommended to check the outcome of operation with ValidateTopology.

Availability: 3.6.0

Examples

Code

```
SELECT topology.MakeTopologyPrecise(
    'city_data',
    gridSize => 2
);
```

See Also

ValidateTopologyPrecision, ValidateTopology

9.5.13 FindTopology

FindTopology — Returns a topology record by different means.

Synopsis

topology **FindTopology**(topogeometry topogeom);
topology **FindTopology**(regclass layerTable, name layerColumn);
topology **FindTopology**(name layerSchema, name layerTable, name layerColumn);
topology **FindTopology**(text topoName);
topology **FindTopology**(int id);

Description

Takes a topology identifier or the identifier of a topology-related object and returns a topology.topology record.

Availability: 3.2.0

Examples

Code

```
SELECT name(FindTopology('features.land_parcels', 'feature'));
```

Output

```
city_data
```

See Also

FindLayer

9.5.14 FindLayer

FindLayer — Returns a topology.layer record by different means.

Synopsis

topology.layer **FindLayer**(topogeometry tg);
topology.layer **FindLayer**(regclass layer_table, name feature_column);
topology.layer **FindLayer**(name schema_name, name table_name, name feature_column);
topology.layer **FindLayer**(integer topology_id, integer layer_id);

Description

Takes a layer identifier or the identifier of a topology-related object and returns a topology.layer record.

Availability: 3.2.0

Examples

Code

```
SELECT layer_id(FindLayer('features.land_parcels', 'feature'));
```

Output

```
1
```

See Also

FindTopology

9.5.15 TotalTopologySize

TotalTopologySize — Total disk space used by the specified topology, including all indexes and TOAST data.

Synopsis

int8 **TotalTopologySize**(name toponame);

Description

Takes a topology name and provides the total disk space used by all its tables, including indexes and TOAST data.

Availability: 3.6.0

Examples

Code

```
SELECT topology.TopologySummary('city_data');
```

Output

```
Topology city_data (329), SRID 4326, precision: 0
22 nodes, 24 edges, 10 faces, 29 topogeoms in 5 layers
Layer 1, type Polygonal (3), 9 topogeoms
 Deploy: features.land_parcels.feature
Layer 2, type Puntal (1), 8 topogeoms
 Deploy: features.traffic_signs.feature
Layer 3, type Lineal (2), 8 topogeoms
 Deploy: features.city_streets.feature
Layer 4, type Polygonal (3), 3 topogeoms
 Hierarchy level 1, child layer 1
 Deploy: features.big_parcels.feature
Layer 5, type Puntal (1), 1 topogeoms
 Hierarchy level 1, child layer 2
 Deploy: features.big_signs.feature
```

9.5.16 UpgradeTopology

UpgradeTopology — Upgrades the specified topology to support large ids (int8) for topology and primitive ids.

Synopsis

void **UpgradeTopology**(name toponame);

Description

Takes a topology name and upgrades it to support large ids (int8) for topology and primitive ids. The function upgrades the following: - face (face_id column from int4 to int8, face_id_seq from int4 to int8) - node (node_id column from int4 to int8, containing_face column from int4 to int8, node_id_seq from int4 to int8) - edge_data (edge_id column from int4 to int8, edge_data_edge_id_seq from int4 to int8, left_face and right_face columns from int4 to int8, start_node and end_node columns from int4 to int8, next_left_edge and next_right_edge columns from int4 to int8) - relation (topogeo_id column from int4 to int8, element_id from int4 to int8) - topology (useslargeids column set to true)

Availability: 3.6.0

Examples

Code

```
SELECT topology.UpgradeTopology('city_data');
```

9.6 Topology Statistics Management

Adding elements to a topology triggers many database queries for finding existing edges that will be split, adding nodes and updating edges that will node with the new linework. For this reason it is useful that statistics about the data in the topology tables are up-to-date.

PostGIS Topology population and editing functions do not automatically update the statistics because a updating stats after each and every change in a topology would be overkill, so it is the caller's duty to take care of that.



Note
That the statistics updated by autovacuum will NOT be visible to transactions which started before autovacuum process completed, so long-running transactions will need to run ANALYZE themselves, to use updated statistics.

9.7 Topology Constructors

9.7.1 CreateTopology

CreateTopology — Creates a new topology schema and registers it in the topology.topology table.

Synopsis

integer **CreateTopology**(name topology_schema_name, integer srid, double precision prec=0, boolean hasz=false, integer topoid=0, boolean useslargeids=false);

Description

Creates a new topology schema with name `topology_name` and registers it in the `topology.topology` table. Topologies must be uniquely named. The topology tables (`edge_data`, `face`, `node`,and `relation` are created in the schema. It returns the id of the topology.

The `srid` is the spatial reference system SRID for the topology. The SRID defaults to -1 (unknown) if not specified.

The tolerance `prec` is measured in the units of the spatial reference system. The tolerance defaults to 0.

`hasz` defaults to false if not specified.

`topoid` optional explicit identifier (allows deterministic topology id assignment, needs to be unique)

`useslargeids` optional, defaults to false. If true, the topology will be created to support large ids (int8) for topology and primitive ids.

This is similar to the SQL/MM ST_InitTopoGeo but has more functionality.

Availability: 1.1

Enhanced: 2.0 added support for `hasZ`.

Enhanced: 3.6.0 added `topoid` and `usesLargeIDs` parameters.

Examples

Create a topology schema called `ma_topo` that stores edges and nodes in Massachusetts State Plane-meters (SRID = 26986). The tolerance represents 0.5 meters since the spatial reference system is meter-based.

Code

```
SELECT topology.CreateTopology('ma_topo', 26986, 0.5);
```

Create a topology for Rhode Island called `ri_topo` in spatial reference system State Plane-feet (SRID = 3438)

Code

```
SELECT topology.CreateTopology('ri_topo', 3438) AS topoid;
```

Output

```
2
```

See Also

Section 4.5, ST_InitTopoGeo

9.7.2 CopyTopology

CopyTopology — Makes a copy of a topology (nodes, edges, faces, layers and TopoGeometries) into a new schema

Synopsis

integer **CopyTopology**(varchar existing_topology_name, varchar new_name);

Description

Creates a new topology with name `new_name`, with SRID and precision copied from `existing_topology_name` The nodes, edges and faces in `existing_topology_name` are copied into the new topology, as well as Layers and their associated TopoGeometries.



Note
The new rows in the `topology.layer` table contain synthetic values for schema_name, table_name and feature_column. This is because the TopoGeometry objects exist only as a definition and are not yet available in a user-defined table.

Availability: 2.0.0

Examples

Make a backup of a topology called `ma_topo`.

Code

```
SELECT topology.CopyTopology('ma_topo', 'ma_topo_backup');
```

See Also

Section 4.5, CreateTopology, RenameTopology

9.7.3 ST_InitTopoGeo

ST_InitTopoGeo — Creates a new topology schema and registers it in the topology.topology table.

Synopsis

text **ST_InitTopoGeo**(varchar topology_schema_name);

Description

This is the SQL-MM equivalent of CreateTopology. It lacks options for spatial reference system and tolerance. it returns a text description of the topology creation, instead of the topology id.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM 3 Topo-Geo and Topo-Net 3: Routine Details: X.3.17

Examples

Code

```
SELECT topology.ST_InitTopoGeo('topo_schema_to_create') AS topocreation;
```

Output

```
Topology-Geometry 'topo_schema_to_create' (id:7) created.
```

See Also

CreateTopology

9.7.4 ST_CreateTopoGeo

ST_CreateTopoGeo — Adds a collection of geometries to a given empty topology and returns a message detailing success.

Synopsis

text **ST_CreateTopoGeo**(varchar atopology, geometry acollection);

Description

Adds a collection of geometries to a given empty topology and returns a message detailing success.

Useful for populating an empty topology.

Availability: 2.0

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details -- X.3.18

Examples

Populate the topology.

Code

```
SELECT topology.ST_CreateTopoGeo(
'ri_topo',
ST_GeomFromText(
 'MULTILINESTRING((384744 236928,384750 236923,384769 236911,384799 236895,384811 ↩
    236890,384833 236884,
 384844 236882,384866 236881,384879 236883,384954 236898,385087 236932,385117 236938,
 385167 236938,385203 236941,385224 236946,385233 236950,385241 236956,385254 236971,
 385260 236979,385268 236999,385273 237018,385273 237037,385271 237047,385267 237057,
```

```
 385225 237125,385210 237144,385192 237161,385167 237192,385162 237202,385159 237214,
 385159 237227,385162 237241,385166 237256,385196 237324,385209 237345,385234 237375,
 385237 237383,385238 237399,385236 237407,385227 237419,385213 237430,385193 237439,
 385174 237451,385170 237455,385169 237460,385171 237475,385181 237503,385190 237521,
 385200 237533,385206 237538,385213 237541,385221 237542,385235 237540,385242 237541,
 385249 237544,385260 237555,385270 237570,385289 237584,385292 237589,385291 237596,385284 ↩
    237630))',
 3438)
 );
```

Output

```
Topology ri_topo populated
```

Create the tables and topogeometries.

Code

```
CREATE TABLE ri.roads(gid serial PRIMARY KEY, road_name text);

SELECT topology.AddTopoGeometryColumn('ri_topo', 'ri', 'roads', 'topo', 'LINE');
```

See Also

TopoGeo_LoadGeometry, AddTopoGeometryColumn, CreateTopology, DropTopology

9.7.5 TopoGeo_AddPoint

TopoGeo_AddPoint — Adds a point to an existing topology using a tolerance and possibly splitting an existing edge.

Synopsis

bigint **TopoGeo_AddPoint**(varchar atopology, geometry apoint, float8 tolerance);

Description

Adds a point to an existing topology and returns its identifier. The given point will snap to existing nodes or edges within given tolerance. An existing edge may be split by the snapped point.

Availability: 2.0.0

See Also

TopoGeo_AddLineString, TopoGeo_AddPolygon, TopoGeo_LoadGeometry, AddNode, CreateTopology

9.7.6 TopoGeo_AddLineString

TopoGeo_AddLineString — Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/-faces.

Synopsis

SETOF bigint **TopoGeo_AddLineString**(varchar atopology, geometry aline, float8 tolerance, int max_edges);

Description

Adds a linestring to an existing topology and returns a set of signed edge identifiers forming it up (negative identifies mean the edge goes in the opposite direction of the input linestring).

The given line will snap to existing nodes or edges within given tolerance. Existing edges and faces may be split by the line. New nodes and faces may be added, in addition to new edges.

The returned edge identifiers may be either existing edges or newly created edges as needed to fully represent the input line as closely as possible.

The number of newly created edges (either covering space previously uncovered or resulting from split of existing edges) may be limited by the `max_edges` parameter.



Note
When adding many lines, the load order can have a large impact on performance because each call checks the line against topology elements already loaded. In bulk workflows, adding large closed or enclosing lines before smaller contained lines can avoid expensive later splits. For loading a collection in one call, see TopoGeo_LoadGeometry.



Note
Updating statistics about topologies being loaded via this function is up to caller, see maintaining statistics during topology editing and population.

Availability: 2.0.0

Enhanced: 3.2.0 added support for returning signed identifier.

Enhanced: 3.7.0 added support for limiting the number of new edges created in the topology.

See Also

TopoGeo_AddPoint, TopoGeo_AddPolygon, TopoGeo_LoadGeometry, AddEdge, CreateTopology

9.7.7 TopoGeo_AddPolygon

TopoGeo_AddPolygon — Adds a polygon to an existing topology using a tolerance and possibly splitting existing edges/faces. Returns face identifiers.

Synopsis

SETOF bigint **TopoGeo_AddPolygon**(varchar atopology, geometry apoly, float8 tolerance);

Description

Adds a polygon to an existing topology and returns a set of face identifiers forming it up. The boundary of the given polygon will snap to existing nodes or edges within given tolerance. Existing edges and faces may be split by the boundary of the new polygon.



Note
Updating statistics about topologies being loaded via this function is up to caller, see maintaining statistics during topology editing and population.

Availability: 2.0.0

See Also

TopoGeo_AddPoint, TopoGeo_AddLineString, TopoGeo_LoadGeometry, AddFace, CreateTopology

9.7.8 TopoGeo_LoadGeometry

TopoGeo_LoadGeometry — Load a geometry into an existing topology, snapping and splitting as needed.

Synopsis

void **TopoGeo_LoadGeometry**(varchar atopology, geometry ageom, float8 tolerance);

Description

Loads a geometry into an existing topology. The given geometry will snap to existing nodes or edges within given tolerance. Existing edges and faces may be split as a consequence of the load.



Note
Updating statistics about topologies being loaded via this function is up to caller, see maintaining statistics during topology editing and population.

Availability: 3.5.0

See Also

TopoGeo_AddPoint, TopoGeo_AddLineString, TopoGeo_AddPolygon, CreateTopology

9.8 Topology Editors

9.8.1 ST_AddIsoNode

ST_AddIsoNode — Adds an isolated node to a face in a topology and returns the nodeid of the new node. If face is null, the node is still created.

Synopsis

bigint **ST_AddIsoNode**(varchar atopology, bigint aface, geometry apoint);

Description

Adds an isolated node with point location `apoint` to an existing face with faceid `aface` to a topology `atopology` and returns the nodeid of the new node.

If the spatial reference system (srid) of the point geometry is not the same as the topology, the `apoint` is not a point geometry, the point is null, or the point intersects an existing edge (even at the boundaries) then an exception is thrown. If the point already exists as a node, an exception is thrown.

If `aface` is not null and the `apoint` is not within the face, then an exception is thrown.

Availability: 1.1

 This method implements the SQL/MM specification. SQL-MM: Topo-Net Routines: X+1.3.1

Examples

See Also

AddNode, CreateTopology, DropTopology, ST_Intersects

9.8.2 ST_AddIsoEdge

ST_AddIsoEdge — Adds an isolated edge defined by geometry `alinestring` to a topology connecting two existing isolated nodes `anode` and `anothernode` and returns the edge id of the new edge.

Synopsis

bigint **ST_AddIsoEdge**(varchar atopology, bigint anode, bigint anothernode, geometry alinestring);

Description

Adds an isolated edge defined by geometry `alinestring` to a topology connecting two existing isolated nodes `anode` and `anothernode` and returns the edge id of the new edge.

If the spatial reference system (srid) of the `alinestring` geometry is not the same as the topology, any of the input arguments are null, or the nodes are contained in more than one face, or the nodes are start or end nodes of an existing edge, then an exception is thrown.

If the `alinestring` is not within the face of the face the `anode` and `anothernode` belong to, then an exception is thrown.

If the `anode` and `anothernode` are not the start and end points of the `alinestring` then an exception is thrown.

Availability: 1.1

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.4

Examples

See Also

ST_AddIsoNode, ST_IsSimple, ST_Within

9.8.3 ST_AddEdgeNewFaces

ST_AddEdgeNewFaces — Add a new edge and, if in doing so it splits a face, delete the original face and replace it with two new faces.

Synopsis

bigint **ST_AddEdgeNewFaces**(varchar atopology, bigint anode, bigint anothernode, geometry acurve);

Description

Add a new edge and, if in doing so it splits a face, delete the original face and replace it with two new faces. Returns the id of the newly added edge.

Updates all existing joined edges and relationships accordingly.

If any arguments are null, the given nodes are unknown (must already exist in the `node` table of the topology schema) , the `acurve` is not a `LINESTRING`, the `anode` and `anothernode` are not the start and endpoints of `acurve` then an error is thrown.

If the spatial reference system (srid) of the `acurve` geometry is not the same as the topology an exception is thrown.

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.12

Examples

See Also

ST_RemEdgeNewFace

ST_AddEdgeModFace

9.8.4 ST_AddEdgeModFace

ST_AddEdgeModFace — Add a new edge and, if in doing so it splits a face, modify the original face and add a new face.

Synopsis

bigint **ST_AddEdgeModFace**(varchar atopology, bigint anode, bigint anothernode, geometry acurve);

Description

Add a new edge and, if doing so splits a face, modify the original face and add a new one.



Note
If possible, the new face will be created on left side of the new edge. This will not be possible if the face on the left side will need to be the Universe face (unbounded).

Returns the id of the newly added edge.

Updates all existing joined edges and relationships accordingly.

If any arguments are null, the given nodes are unknown (must already exist in the `node` table of the topology schema) , the `acurve` is not a `LINESTRING`, the `anode` and `anothernode` are not the start and endpoints of `acurve` then an error is thrown.

If the spatial reference system (srid) of the `acurve` geometry is not the same as the topology an exception is thrown.

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.13

Examples

See Also

ST_RemEdgeModFace

ST_AddEdgeNewFaces

9.8.5 ST_RemEdgeNewFace

ST_RemEdgeNewFace — Removes an edge and, if the removed edge separated two faces, delete the original faces and replace them with a new face.

Synopsis

bigint **ST_RemEdgeNewFace**(varchar atopology, bigint anedge);

Description

Removes an edge and, if the removed edge separated two faces, delete the original faces and replace them with a new face.

Returns the id of a newly created face or NULL, if no new face is created. No new face is created when the removed edge is dangling or isolated or confined with the universe face (possibly making the universe flood into the face on the other side).

Updates all existing joined edges and relationships accordingly.

Refuses to remove an edge participating in the definition of an existing TopoGeometry. Refuses to heal two faces if any TopoGeometry is defined by only one of them (and not the other).

If any arguments are null, the given edge is unknown (must already exist in the `edge` table of the topology schema), the topology name is invalid then an error is thrown.

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.14

Examples

See Also

ST_RemEdgeModFace

ST_AddEdgeNewFaces

9.8.6 ST_RemEdgeModFace

ST_RemEdgeModFace — Removes an edge, and if the edge separates two faces deletes one face and modifies the other face to cover the space of both.

Synopsis

bigint **ST_RemEdgeModFace**(varchar atopology, bigint anedge);

Description

Removes an edge, and if the removed edge separates two faces deletes one face and modifies the other face to cover the space of both. Preferentially keeps the face on the right, to be consistent with ST_AddEdgeModFace. Returns the id of the face which is preserved.

Updates all existing joined edges and relationships accordingly.

Refuses to remove an edge participating in the definition of an existing TopoGeometry. Refuses to heal two faces if any TopoGeometry is defined by only one of them (and not the other).

If any arguments are null, the given edge is unknown (must already exist in the `edge` table of the topology schema), the topology name is invalid then an error is thrown.

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.15

Examples

See Also

ST_AddEdgeModFace

ST_RemEdgeNewFace

9.8.7 ST_ChangeEdgeGeom

ST_ChangeEdgeGeom — Changes the shape of an edge without affecting the topology structure.

Synopsis

text **ST_ChangeEdgeGeom**(varchar atopology, bigint anedge, geometry acurve);

Description

Changes the shape of an edge without affecting the topology structure.

If any arguments are null, the given edge does not exist in the `edge` table of the topology schema, the `acurve` is not a `LINESTRING`, or the modification would change the underlying topology then an error is thrown.

If the spatial reference system (srid) of the `acurve` geometry is not the same as the topology an exception is thrown.

If the new `acurve` is not simple, then an error is thrown.

If moving the edge from old to new position would hit an obstacle then an error is thrown.

Availability: 1.1.0

Enhanced: 2.0.0 adds topological consistency enforcement

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details X.3.6

Examples

Code

```
SELECT topology.ST_ChangeEdgeGeom(
    'ma_topo',
    1,
        ST_GeomFromText(
            'LINESTRING(227591.9 893900.4,227622.6 893844.3,227641.6 893816.6,227704.5 ↩
    893778.5)',
            26986)
);
```

Output

```
Edge 1 changed
```

See Also

ST_AddEdgeModFace

ST_RemEdgeModFace

ST_ModEdgeSplit

9.8.8 ST_ModEdgeSplit

ST_ModEdgeSplit — Split an edge by creating a new node along an existing edge, modifying the original edge and adding a new edge.

Synopsis

bigint **ST_ModEdgeSplit**(varchar atopology, bigint anedge, geometry apoint);

Description

Split an edge by creating a new node along an existing edge, modifying the original edge and adding a new edge. Updates all existing joined edges and relationships accordingly. Returns the identifier of the newly added node.

Availability: 1.1

Changed: 2.0 - In prior versions, this was misnamed ST_ModEdgesSplit

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.9

Examples

Add an edge to split.

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227592 893910,227600 893910) ↩
    ', 26986)) AS edgeid;
```

Output

```
3
```

Split the edge with a new node.

Code

```
SELECT topology.ST_ModEdgeSplit('ma_topo', 3, ST_SetSRID(ST_Point(227594, 893910), 26986)) ↩
    AS node_id;
```

Output

```
7
```

See Also

ST_NewEdgesSplit, ST_ModEdgeHeal, ST_NewEdgeHeal, AddEdge

9.8.9 ST_ModEdgeHeal

ST_ModEdgeHeal — Heals two edges by deleting the node connecting them, modifying the first edge and deleting the second edge. Returns the id of the deleted node.

Synopsis

bigint **ST_ModEdgeHeal**(varchar atopology, bigint anedge, bigint anotheredge);

Description

Heals two edges by deleting the node connecting them, modifying the first edge and deleting the second edge. Returns the id of the deleted node. Updates all existing joined edges and relationships accordingly.

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.9

See Also

ST_ModEdgeSplit ST_NewEdgesSplit

9.8.10 ST_NewEdgeHeal

ST_NewEdgeHeal — Heals two edges by deleting the node connecting them, deleting both edges, and replacing them with an edge whose direction is the same as the first edge provided.

Synopsis

bigint **ST_NewEdgeHeal**(varchar atopology, bigint anedge, bigint anotheredge);

Description

Heals two edges by deleting the node connecting them, deleting both edges, and replacing them with an edge whose direction is the same as the first edge provided. Returns the id of the new edge replacing the healed ones. Updates all existing joined edges and relationships accordingly.

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.9

See Also

ST_ModEdgeHeal ST_ModEdgeSplit ST_NewEdgesSplit

9.8.11 ST_MoveIsoNode

ST_MoveIsoNode — Moves an isolated node in a topology from one point to another. If new `apoint` geometry exists as a node an error is thrown. Returns description of move.

Synopsis

text **ST_MoveIsoNode**(varchar atopology, bigint anode, geometry apoint);

Description

Moves an isolated node in a topology from one point to another. If new `apoint` geometry exists as a node an error is thrown.

If any arguments are null, the `apoint` is not a point, the existing node is not isolated (is a start or end point of an existing edge), new node location intersects an existing edge (even at the end points) or the new location is in a different face (since 3.2.0) then an exception is thrown.

If the spatial reference system (srid) of the point geometry is not the same as the topology an exception is thrown.

Availability: 2.0.0

Enhanced: 3.2.0 ensures the nod cannot be moved in a different face

 This method implements the SQL/MM specification. SQL-MM: Topo-Net Routines: X.3.2

Examples

Add an isolated node with no face.

Code

```
SELECT topology.ST_AddIsoNode('ma_topo', NULL, ST_GeomFromText('POINT(227579 893916)', ↩
    26986)) AS nodeid;
```

Output

```
7
```

Move the new node.

Code

```
SELECT topology.ST_MoveIsoNode('ma_topo', 7, ST_GeomFromText('POINT(227579.5 893916.5)', ↩
    26986)) AS descrip;
```

Output

```
Isolated Node 7 moved to location 227579.5,893916.5
```

See Also

ST_AddIsoNode

9.8.12 ST_NewEdgesSplit

ST_NewEdgesSplit — Split an edge by creating a new node along an existing edge, deleting the original edge and replacing it with two new edges. Returns the id of the new node created that joins the new edges.

Synopsis

bigint **ST_NewEdgesSplit**(varchar atopology, bigint anedge, geometry apoint);

Description

Split an edge with edge id `anedge` by creating a new node with point location `apoint` along current edge, deleting the original edge and replacing it with two new edges. Returns the id of the new node created that joins the new edges. Updates all existing joined edges and relationships accordingly.

If the spatial reference system (srid) of the point geometry is not the same as the topology, the `apoint` is not a point geometry, the point is null, the point already exists as a node, the edge does not correspond to an existing edge or the point is not within the edge then an exception is thrown.

Availability: 1.1

 This method implements the SQL/MM specification. SQL-MM: Topo-Net Routines: X.3.8

Examples

Add an edge to split.

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227575 893917,227592 893900) ↩
    ', 26986)) AS edgeid;
```

Output

```
2
```

Split the new edge.

Code

```
SELECT topology.ST_NewEdgesSplit('ma_topo', 2, ST_GeomFromText('POINT(227578.5 893913.5)', ↩
    26986)) AS newnodeid;
```

Output

```
6
```

See Also

ST_ModEdgeSplit ST_ModEdgeHeal ST_NewEdgeHeal AddEdge

9.8.13 ST_RemoveIsoNode

ST_RemoveIsoNode — Removes an isolated node and returns description of action. If the node is not isolated (is start or end of an edge), then an exception is thrown.

Synopsis

text **ST_RemoveIsoNode**(varchar atopology, bigint anode);

Description

Removes an isolated node and returns description of action. If the node is not isolated (is start or end of an edge), then an exception is thrown.

Availability: 1.1


This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X+1.3.3

Examples

Remove an isolated node with no face.

Code

```
SELECT topology.ST_RemoveIsoNode('ma_topo', 7) AS result;
```

Output

```
Isolated node 7 removed
```

See Also

ST_AddIsoNode

9.8.14 ST_RemoveIsoEdge

ST_RemoveIsoEdge — Removes an isolated edge and returns description of action. If the edge is not isolated, then an exception is thrown.

Synopsis

text **ST_RemoveIsoEdge**(varchar atopology, bigint anedge);

Description

Removes an isolated edge and returns description of action. If the edge is not isolated, then an exception is thrown.

Availability: 1.1

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X+1.3.3

Examples

Remove an isolated edge with no face.

Code

```
SELECT topology.ST_RemoveIsoEdge('ma_topo', 7) AS result;
```

Output

```
Isolated edge 7 removed
```

See Also

ST_AddIsoNode

9.9 Topology Accessors

9.9.1 GetEdgeByPoint

GetEdgeByPoint — Finds the edge-id of an edge that intersects a given point.

Synopsis

bigint **GetEdgeByPoint**(varchar atopology, geometry apoint, float8 tol1);

Description

Retrieves the id of an edge that intersects a Point.

The function returns an integer (id-edge) given a topology, a POINT and a tolerance. If tolerance = 0 then the point has to intersect the edge.

If `apoint` doesn't intersect an edge, returns 0 (zero).

If use tolerance > 0 and there is more than one edge near the point then an exception is thrown.



Note
If tolerance = 0, the function uses ST_Intersects otherwise uses ST_DWithin.

Performed by the GEOS module.

Availability: 2.0.0

Examples

These examples use edges we created in AddEdge

Code

```
SELECT topology.GetEdgeByPoint('ma_topo', geom, 1) As with1mtol, topology.GetEdgeByPoint(' ↩
    ma_topo', geom, 0) As withnotol
FROM ST_GeomFromEWKT('SRID=26986;POINT(227622.6 893843)') As geom;
```

Output

```
 with1mtol | withnotol
-----------+-----------
         2 |         0
```

This query errors because two or more edges are found.

Code

```
SELECT topology.GetEdgeByPoint('ma_topo', geom, 1) As nearnode
FROM ST_GeomFromEWKT('SRID=26986;POINT(227591.9 893900.4)') As geom;
```

Output

```
ERROR:  Two or more edges found
```

See Also

AddEdge, GetNodeByPoint, GetFaceByPoint

9.9.2 GetFaceByPoint

GetFaceByPoint — Finds face intersecting a given point.

Synopsis

bigint **GetFaceByPoint**(varchar atopology, geometry apoint, float8 tol1);

Description

Finds a face referenced by a Point, with given tolerance.

The function will effectively look for a face intersecting a circle having the point as center and the tolerance as radius.

If no face intersects the given query location, 0 is returned (universal face).

If more than one face intersect the query location an exception is thrown.

Availability: 2.0.0

Enhanced: 3.2.0 more efficient implementation and clearer contract, stops working with invalid topologies.

Examples

Code

```
SELECT topology.GetFaceByPoint('ma_topo', geom, 10) As with1mtol, topology.GetFaceByPoint(' ↩
    ma_topo', geom, 0) As withnotol
FROM ST_GeomFromEWKT('POINT(234604.6 899382.0)') As geom;
```

Output

```
 with1mtol | withnotol
-----------+-----------
         1 |         0
```

This query errors because two or more faces are found.

Code

```
SELECT topology.GetFaceByPoint('ma_topo', geom, 1) As nearnode
FROM ST_GeomFromEWKT('POINT(227591.9 893900.4)') As geom;
```

Output

```
ERROR:  Two or more faces found
```

See Also

GetFaceContainingPoint, AddFace, GetNodeByPoint, GetEdgeByPoint

9.9.3 GetFaceContainingPoint

GetFaceContainingPoint — Finds the face containing a point.

Synopsis

bigint **GetFaceContainingPoint**(text atopology, geometry apoint);

Description

Returns the id of the face containing a point.

An exception is thrown if the point falls on a face boundary.



Note
The function relies on a valid topology, using edge linking and face labeling.

Availability: 3.2.0

See Also

ST_GetFaceGeometry

9.9.4 GetNodeByPoint

GetNodeByPoint — Finds the node-id of a node at a point location.

Synopsis

bigint **GetNodeByPoint**(varchar atopology, geometry apoint, float8 tol1);

Description

Retrieves the id of a node at a point location.

The function returns an integer (id-node) given a topology, a POINT and a tolerance. If tolerance = 0 means exact intersection, otherwise retrieves the node from an interval.

If `apoint` doesn't intersect a node, returns 0 (zero).

If use tolerance > 0 and there is more than one node near the point then an exception is thrown.



Note
If tolerance = 0, the function uses ST_Intersects otherwise uses ST_DWithin.

Performed by the GEOS module.

Availability: 2.0.0

Examples

These examples use edges we created in AddEdge

Code

```
SELECT topology.GetNodeByPoint('ma_topo', geom, 1) As nearnode
FROM ST_GeomFromEWKT('SRID=26986;POINT(227591.9 893900.4)') As geom;
```

Output

```
2
```

This query errors because the tolerance matches two or more nodes.

Code

```
SELECT topology.GetNodeByPoint('ma_topo', geom, 1000) As too_much_tolerance
FROM ST_GeomFromEWKT('SRID=26986;POINT(227591.9 893900.4)') As geom;
```

Output

```
ERROR:  Two or more nodes found
```

See Also

AddEdge, GetEdgeByPoint, GetFaceByPoint

9.9.5 GetTopologyID

GetTopologyID — Returns the id of a topology in the topology.topology table given the name of the topology.

Synopsis

integer **GetTopologyID**(varchar toponame);

Description

Returns the id of a topology in the topology.topology table given the name of the topology.

Availability: 1.1

Examples

Code

```
SELECT topology.GetTopologyID('ma_topo') AS topo_id;
```

Output

```
1
```

See Also

CreateTopology, DropTopology, GetTopologyName, GetTopologySRID

9.9.6 GetTopologySRID

GetTopologySRID — Returns the SRID of a topology in the topology.topology table given the name of the topology.

Synopsis

integer **GetTopologyID**(varchar toponame);

Description

Returns the spatial reference id of a topology in the topology.topology table given the name of the topology.

Availability: 2.0.0

Examples

Code

```
SELECT topology.GetTopologySRID('ma_topo') AS srid;
```

Output

```
4326
```

See Also

CreateTopology, DropTopology, GetTopologyName, GetTopologyID

9.9.7 GetTopologyName

GetTopologyName — Returns the name of a topology (schema) given the id of the topology.

Synopsis

varchar **GetTopologyName**(integer topology_id);

Description

Returns the topology name (schema) of a topology from the topology.topology table given the topology id of the topology.

Availability: 1.1

Examples

Code

```
SELECT topology.GetTopologyName(1) AS topo_name;
```

Output

```
ma_topo
```

See Also

CreateTopology, DropTopology, GetTopologyID, GetTopologySRID

9.9.8 ST_GetFaceEdges

ST_GetFaceEdges — Returns a set of ordered edges that bound `aface`.

Synopsis

getfaceedges_returntype **ST_GetFaceEdges**(varchar atopology, bigint aface);

Description

Returns a set of ordered edges that bound `aface`. Each output consists of a sequence and edgeid. Sequence numbers start with value 1.

Enumeration of each ring edges start from the edge with smallest identifier. Order of edges follows a left-hand-rule (bound face is on the left of each directed edge).

Availability: 2.0

 This method implements the SQL/MM specification. SQL-MM 3 Topo-Geo and Topo-Net 3: Routine Details: X.3.5

Examples

This query returns the edges that bound face 1.

Code

```
SELECT (topology.ST_GetFaceEdges('tt', 1)).*;
```

Output

```
 sequence | edge
----------+------
        1 |   -4
        2 |    5
        3 |    7
        4 |   -6
        5 |    1
        6 |    2
        7 |    3
```

This query returns the sequence, edge identifier, and geometry of the edges that bound face 1. Use ST_GetFaceGeometry if only the geometry and sequence are needed.

Code

```
SELECT t.seq, t.edge, geom
FROM topology.ST_GetFaceEdges('tt', 1) As t(seq, edge)
    INNER JOIN tt.edge AS e ON abs(t.edge) = e.edge_id;
```

See Also

GetRingEdges, AddFace, ST_GetFaceGeometry

9.9.9 ST_GetFaceGeometry

ST_GetFaceGeometry — Returns the polygon in the given topology with the specified face id.

Synopsis

geometry **ST_GetFaceGeometry**(varchar atopology, bigint aface);

Description

Returns the polygon in the given topology with the specified face id. Builds the polygon from the edges making up the face.

Availability: 1.1


This method implements the SQL/MM specification. SQL-MM 3 Topo-Geo and Topo-Net 3: Routine Details: X.3.16

Examples

This query returns the WKT of the polygon added with AddFace.

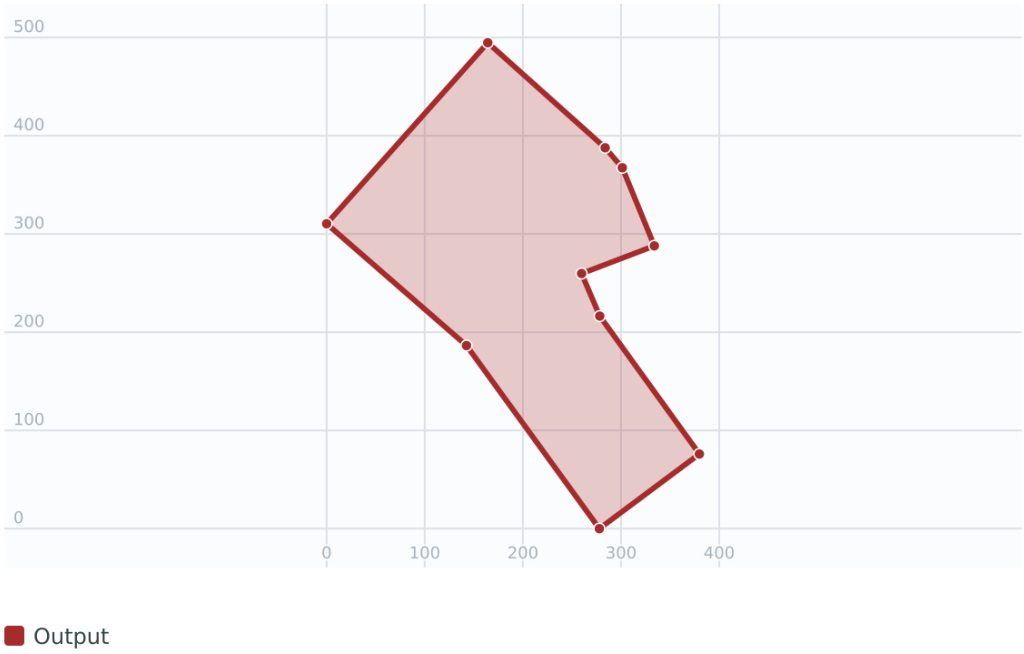
Code

```
SELECT topology.ST_GetFaceGeometry('ma_topo', 1) AS facegeomwkt;
```

Output

```
POLYGON((234776.9 899563.7,234896.5 899456.7,234914 899436.4,234946.6 899356.9,
234872.5 899328.7,234891 899285.4,234992.5 899145,234890.6 899069,
234755.2 899255.4,234612.7 899379.4,234776.9 899563.7))
```

Figure



See Also

AddFace

9.9.10 GetRingEdges

GetRingEdges — Returns the ordered set of signed edge identifiers met by walking on an a given edge side.

Synopsis

getfaceedges_returntype **GetRingEdges**(varchar atopology, bigint aring, integer max_edges=null);

Description

Returns the ordered set of signed edge identifiers met by walking on an a given edge side. Each output consists of a sequence and a signed edge id. Sequence numbers start with value 1.

If you pass a positive edge id, the walk starts on the left side of the corresponding edge and follows the edge direction. If you pass a negative edge id, the walk starts on the right side of it and goes backward.

If `max_edges` is not null no more than those records are returned by that function. This is meant to be a safety parameter when dealing with possibly invalid topologies.



Note
This function uses edge ring linking metadata.

Availability: 2.0.0

See Also

ST_GetFaceEdges, GetNodeEdges

9.9.11 GetNodeEdges

GetNodeEdges — Returns an ordered set of edges incident to the given node.

Synopsis

getfaceedges_returntype **GetNodeEdges**(varchar atopology, bigint anode);

Description

Returns an ordered set of edges incident to the given node. Each output consists of a sequence and a signed edge id. Sequence numbers start with value 1. A positive edge starts at the given node. A negative edge ends into the given node. Closed edges will appear twice (with both signs). Order is clockwise starting from northbound.



Note
This function computes ordering rather than deriving from metadata and is thus usable to build edge ring linking.

Availability: 2.0

See Also

getfaceedges_returntype, GetRingEdges, ST_Azimuth

9.9.12 FindVertexSegmentPairsBelowDistance

FindVertexSegmentPairsBelowDistance — Find pairs of topology vertex/segment that are closer than tolerated distance

Synopsis

setof record **FindVertexSegmentPairsBelowDistance**(varchar atopology, float8 tolerance);

Description

Returns a relation in which each tuple is a pair of vertex/segment where the vertex is NOT an endpoint of the segment but the distance between them is below the given tolerance.

Availability: 3.7

See Also

ValidateTopologyPrecision

9.10 Topology Processing

9.10.1 Polygonize

Polygonize — Finds and registers all faces defined by topology edges.

Synopsis

text **Polygonize**(varchar toponame);

Description

Registers all faces that can be built out a topology edge primitives.

The target topology is assumed to contain no self-intersecting edges.



Note
Already known faces are recognized, so it is safe to call Polygonize multiple times on the same topology.



Note
This function does not use nor set the next_left_edge and next_right_edge fields of the edge table.

Availability: 2.0.0

See Also

AddFace, ST_Polygonize

9.10.2 AddNode

AddNode — Adds a point node to the node table in the specified topology schema and returns the nodeid of new node. If point already exists as node, the existing nodeid is returned.

Synopsis

bigint **AddNode**(varchar toponame, geometry apoint, boolean allowEdgeSplitting=false, boolean computeContainingFace=false);

Description

Adds a point node to the node table in the specified topology schema. The AddEdge function automatically adds start and end points of an edge when called so not necessary to explicitly add nodes of an edge.

If any edge crossing the node is found either an exception is raised or the edge is split, depending on the `allowEdgeSplitting` parameter value.

If `computeContainingFace` is true a newly added node would get the correct containing face computed.



Note
If the `apoint` geometry already exists as a node, the node is not added but the existing nodeid is returned.

Availability: 2.0.0

Examples

Code

```
SELECT topology.AddNode('ma_topo', ST_GeomFromText('POINT(227641.6 893816.5)', 26986)) AS ↩
    nodeid;
```

Output

```
4
```

See Also

AddEdge, CreateTopology

9.10.3 AddEdge

AddEdge — Adds a linestring edge to the edge table and associated start and end points to the point nodes table of the specified topology schema using the specified linestring geometry and returns the edgeid of the new (or existing) edge.

Synopsis

bigint **AddEdge**(varchar toponame, geometry aline);

Description

Adds an edge to the edge table and associated nodes to the nodes table of the specified `toponame` schema using the specified linestring geometry and returns the edgeid of the new or existing record. The newly added edge has "universe" face on both sides and links to itself.



Note
If the `aline` geometry crosses, overlaps, contains or is contained by an existing linestring edge, then an error is thrown and the edge is not added.



Note
The geometry of `aline` must have the same `srid` as defined for the topology otherwise an invalid spatial reference sys error will be thrown.

Performed by the GEOS module.



Warning
AddEdge is deprecated as of 3.5.0. Use TopoGeo_AddLineString instead.

Availability: 2.0.0

Examples

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227575.8 893917.2,227591.9 ↩
    893900.4)', 26986)) AS edgeid;
```

Output

```
1
```

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227591.9 893900.4,227622.6 ↩
    893844.2,227641.6 893816.5,227704.5 893778.5)', 26986)) AS edgeid;
```

Output

```
2
```

This query errors because the edge intersects an existing edge away from the endpoints.

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227591.2 893900,227591.9 ↩
    893900.4,227704.5 893778.5)', 26986)) AS edgeid;
```

Output

```
ERROR:  Edge intersects (not on endpoints) with existing edge 1
```

See Also

TopoGeo_AddLineString, CreateTopology, Section 4.5

9.10.4 AddFace

AddFace — Registers a face primitive to a topology and gets its identifier.

Synopsis

bigint **AddFace**(varchar toponame, geometry apolygon, boolean force_new=false);

Description

Registers a face primitive to a topology and gets its identifier.

For a newly added face, the edges forming its boundaries and the ones contained in the face will be updated to have correct values in the left_face and right_face fields. Isolated nodes contained in the face will also be updated to have a correct containing_face field value.



Note
This function does not use nor set the next_left_edge and next_right_edge fields of the edge table.

The target topology is assumed to be valid (containing no self-intersecting edges). An exception is raised if: The polygon boundary is not fully defined by existing edges or the polygon overlaps an existing face.

If the `apolygon` geometry already exists as a face, then: if `force_new` is false (the default) the face id of the existing face is returned; if `force_new` is true a new id will be assigned to the newly registered face.



Note
When a new registration of an existing face is performed (force_new=true), no action will be taken to resolve dangling references to the existing face in the edge, node an relation tables, nor will the MBR field of the existing face record be updated. It is up to the caller to deal with that.



Note
The `apolygon` geometry must have the same `srid` as defined for the topology otherwise an invalid spatial reference sys error will be thrown.

Availability: 2.0.0

Examples

First add the edges using `generate_series` as an iterator. This query only works for polygons with fewer than 10000 points because of the maximum value in `generate_series`.

Code

```
SELECT topology.AddEdge('ma_topo', ST_MakeLine(ST_PointN(geom,i), ST_PointN(geom, i + 1) ))  ↩
    As edgeid
FROM (
  SELECT ST_NPoints(geom) AS npt, geom
  FROM (
    SELECT ST_Boundary(ST_GeomFromText('POLYGON((234896.5 899456.7,234914 899436.4,234946.6  ↩
    899356.9,234872.5 899328.7,
                234891 899285.4,234992.5 899145, 234890.6 899069,234755.2 899255.4,
                234612.7 899379.4,234776.9 899563.7,234896.5 899456.7))', 26986) )  As geom
  ) AS geoms
) AS facen
CROSS JOIN generate_series(1,10000) AS i
WHERE i < npt;
```

Output

```
3
4
5
6
7
8
9
10
11
12
```

Then add the face.

Code

```
SELECT topology.AddFace(
    'ma_topo',
    ST_GeomFromText(
    'POLYGON((234896.5 899456.7,234914 899436.4,234946.6 899356.9,234872.5 899328.7,
    234891 899285.4,234992.5 899145, 234890.6 899069,234755.2 899255.4,
    234612.7 899379.4,234776.9 899563.7,234896.5 899456.7))',
    26986) ) As faceid;
```

Output

```
1
```

See Also

AddEdge, CreateTopology, Section 4.5

9.10.5 ST_Simplify

ST_Simplify — Returns a "simplified" geometry version of the given TopoGeometry using the Douglas-Peucker algorithm.

Synopsis

geometry **ST_Simplify**(topogeometry tg, float8 tolerance);

Description

Returns a "simplified" geometry version of the given TopoGeometry using the Douglas-Peucker algorithm on each component edge.



Note
The returned geometry may be non-simple or non-valid.
Splitting component edges may help retaining simplicity/validity.

Performed by the GEOS module.

Availability: 2.1.0

See Also

Geometry ST_Simplify, ST_IsSimple, ST_IsValid, ST_ModEdgeSplit

9.10.6 RemoveUnusedPrimitives

RemoveUnusedPrimitives — Removes topology primitives which not needed to define existing TopoGeometry objects.

Synopsis

bigint **RemoveUnusedPrimitives**(text topology_name, geometry bbox);

Description

Finds all primitives (nodes, edges, faces) that are not strictly needed to represent existing TopoGeometry objects and removes them, maintaining topology validity (edge linking, face labeling) and TopoGeometry space occupation.

No new primitive identifiers are created, but rather existing primitives are expanded to include merged faces (upon removing edges) or healed edges (upon removing nodes).

Availability: 3.3.0

See Also

ST_ModEdgeHeal, ST_RemEdgeModFace

9.11 TopoGeometry Constructors

9.11.1 CreateTopoGeom

CreateTopoGeom — Creates a new topo geometry object from topo element array - tg_type: 1:[multi]point, 2:[multi]line, 3:[multi]poly, 4:collection

Synopsis

topogeometry **CreateTopoGeom**(varchar toponame, integer tg_type, integer layer_id, topoelementarray tg_objs, bigint tg_id);
topogeometry **CreateTopoGeom**(varchar toponame, integer tg_type, integer layer_id);

Description

Creates a topogeometry object for layer denoted by `layer_id` and registers it in the relations table in the `toponame` schema.

`tg_type` is an integer: 1:[multi]point (punctal), 2:[multi]line (lineal), 3:[multi]poly (areal), 4:collection. `layer_id` is the layer id in the topology.layer table.

punctal layers are formed from set of nodes, lineal layers are formed from a set of edges, areal layers are formed from a set of faces, and collections can be formed from a mixture of nodes, edges, and faces.

Omitting the array of components generates an empty TopoGeometry object.

Availability: 1.1

Examples

Form from existing edges.

Create a topogeom in ri_topo schema for layer 2 (our ri_roads), of type (2) LINE, for the first edge (we loaded in `ST_CreateTopoGeo`)

Code

```
INSERT INTO ri.ri_roads(road_name, topo)
VALUES (
    'Unknown',
    topology.CreateTopoGeom(
        'ri_topo',
        2,
        2,
        '{{1,2}}'::topology.topoelementarray
    )
);
```

Convert an areal geometry to best guess topogeometry.

Lets say we have geometries that should be formed from a collection of faces. We have for example blockgroups table and want to know the topo geometry of each block group. If our data was perfectly aligned, we could do this:

Create the topogeometry column.

Code

```
SELECT topology.AddTopoGeometryColumn(
'topo_boston',
'boston', 'blockgroups', 'topo', 'POLYGON');
```

Output

```
1
```

Update the column assuming everything is perfectly aligned with the edges.

Code

```
UPDATE boston.blockgroups AS bg
SET topo = topology.CreateTopoGeom(
    'topo_boston',
    3,
    1,
    foo.bfaces
)
FROM (
    SELECT
        b.gid,
        topology.TopoElementArray_Agg(ARRAY[f.face_id, 3]) AS bfaces
    FROM boston.blockgroups AS b
    INNER JOIN topo_boston.face AS f
        ON b.geom && f.mbr
    WHERE ST_Covers(
        b.geom,
        topology.ST_GetFaceGeometry('topo_boston', f.face_id)
    )
    GROUP BY b.gid
) AS foo
WHERE foo.gid = bg.gid;
```

The world is rarely perfect, so this version allows some error. It counts a face if 50 percent of it falls within the expected block group boundary.

Code

```
WITH candidate_faces AS (
    SELECT
        b.gid,
        b.geom AS block_geom,
        f.face_id,
        topology.ST_GetFaceGeometry('topo_boston', f.face_id) AS face_geom
    FROM boston.blockgroups AS b
    INNER JOIN topo_boston.face AS f
        ON b.geom && f.mbr
),
block_faces AS (
    SELECT
        gid,
        topology.TopoElementArray_Agg(ARRAY[face_id, 3]) AS bfaces
    FROM candidate_faces
    WHERE ST_Covers(block_geom, face_geom)
        OR (
            ST_Intersects(block_geom, face_geom)
            AND ST_Area(ST_Intersection(block_geom, face_geom))
                > ST_Area(face_geom) * 0.5
        )
    GROUP BY gid
)
UPDATE boston.blockgroups AS bg
SET topo = topology.CreateTopoGeom(
    'topo_boston',
    3,
    1,
    block_faces.bfaces
)
FROM block_faces
WHERE block_faces.gid = bg.gid;
```

To convert the topogeometry back to a denormalized geometry aligned with faces and edges, cast the topogeometry to a geometry. The resulting geometries are aligned with the TIGER street centerlines.

Code

```
UPDATE boston.blockgroups SET new_geom = topo::geometry;
```

See Also

AddTopoGeometryColumn, toTopoGeom ST_CreateTopoGeo, ST_GetFaceGeometry, TopoElementArray, TopoElementArray_Agg

9.11.2 toTopoGeom

toTopoGeom — Converts a simple Geometry into a topo geometry.

Synopsis

topogeometry **toTopoGeom**(geometry geom, varchar toponame, integer layer_id, float8 tolerance=-1);
topogeometry **toTopoGeom**(geometry geom, topogeometry topogeom, float8 tolerance=-1);

Description

Converts a simple Geometry into a TopoGeometry.

Topological primitives required to represent the input geometry will be added to the underlying topology, possibly splitting existing ones, and they will be associated with the output TopoGeometry in the `relation` table.

Existing TopoGeometry objects (with the possible exception of `topogeom`, if given) will retain their shapes.

When `tolerance` is given it will be used to snap the input geometry to existing primitives. The special value -1 (default) means use the topology precision as tolerance.

In the first form a new TopoGeometry will be created for the given layer (`layer_id`) of the given topology (`toponame`).

In the second form the primitives resulting from the conversion will be added to the pre-existing TopoGeometry (`topogeom`), possibly adding space to its final shape. To have the new shape completely replace the old one see clearTopoGeom.

Availability: 2.0

Enhanced: 2.1.0 adds the version taking an existing TopoGeometry.

Changed: 3.7.0 changes meaning of tolerance 0 to really mean 0 and -1 (new default) to request using the topology precision.

Examples

This is a full self-contained workflow

Create the topology without allowing any tolerance.

Code

```
SELECT topology.CreateTopology('topo_boston_test', 2249);
```

Create a new table.

Code

```
CREATE TABLE nei_topo(gid serial primary key, nei varchar(30));
```

Add a topogeometry column to the new table.

Code

```
SELECT topology.AddTopoGeometryColumn('topo_boston_test', 'public', 'nei_topo', 'topo', ' ↩
    MULTIPOLYGON') AS new_layer_id;
```

Output

```
1
```

Use the new layer identifier to populate the topogeometry column.

Code

```
INSERT INTO nei_topo(nei, topo)
SELECT nei, topology.toTopoGeom(geom, 'topo_boston_test', 1)
FROM neighborhoods
WHERE gid BETWEEN 1 and 15;
```

Verify what happened.

Code

```
SELECT * FROM
    topology.TopologySummary('topo_boston_test');
```

Output

```
Topology topo_boston_test (5), SRID 2249, precision 0
61 nodes, 87 edges, 35 faces, 15 topogeoms in 1 layers
Layer 1, type Polygonal (3), 15 topogeoms
 Deploy: public.nei_topo.topo
```

Shrink all TopoGeometry polygons by 10 meters.

Code

```
UPDATE nei_topo SET topo = toTopoGeom(ST_Buffer(topo, -10), clearTopoGeom(topo), 0);
```

Find faces that are left without any TopoGeometry reference after the shrink operation.

Code

```
SELECT ST_GetFaceGeometry('topo_boston_test', f.face_id)
  FROM topo_boston_test.face f
  WHERE f.face_id > 0
  AND NOT EXISTS (
    SELECT * FROM topo_boston_test.relation
    WHERE layer_id = 1 AND element_id = f.face_id
  );
```

See Also

CreateTopology, AddTopoGeometryColumn, CreateTopoGeom, TopologySummary, clearTopoGeom

9.11.3 TopoElementArray_Agg

TopoElementArray_Agg — Returns a `topoelementarray` for a set of element_id, type arrays (topoelements).

Synopsis

topoelementarray **TopoElementArray_Agg**(topoelement set tefield);

Description

Used to create a TopoElementArray from a set of TopoElement.

Availability: 2.0.0

Examples

Code

```
SELECT topology.TopoElementArray_Agg(ARRAY[e, t] ORDER BY e, t) As tea
FROM generate_series(1, 3) As e CROSS JOIN generate_series(1, 4) As t;
```

Output

```
{{1,1},{1,2},{1,3},{1,4},{2,1},{2,2},{2,3},{2,4},{3,1},{3,2},{3,3},{3,4}}
```

See Also

TopoElement, TopoElementArray

9.11.4 TopoElement

TopoElement — Converts a topogeometry to a topoelement.

Synopsis

topoelement **TopoElement**(topogeometry topo);

Description

Converts a TopoGeometry to a TopoElement.

Availability: 3.4.0

Examples

This is a full self-contained workflow

Return the TopoElement representation of each topogeometry.

Code

```
SELECT TopoElement(topo)
FROM neighborhoods;
```

Use a cast to aggregate topogeometries into a TopoElementArray.

Code

```
SELECT topology.TopoElementArray_Agg(topo::topoelement)
FROM neighborhoods
GROUP BY city;
```

See Also

TopoElementArray_Agg, TopoGeometry, TopoElement

9.12 TopoGeometry Editors

9.12.1 clearTopoGeom

clearTopoGeom — Clears the content of a topo geometry.

Synopsis

topogeometry **clearTopoGeom**(topogeometry topogeom);

Description

Clears the content a TopoGeometry turning it into an empty one. Mostly useful in conjunction with toTopoGeom to replace the shape of existing objects and any dependent object in higher hierarchical levels.

Availability: 2.1

Examples

Shrink all TopoGeometry polygons by 10 meters.

Code

```
UPDATE nei_topo SET topo = toTopoGeom(ST_Buffer(topo, -10), clearTopoGeom(topo), 0);
```

See Also

toTopoGeom

9.12.2 TopoGeom_addElement

TopoGeom_addElement — Adds an element to the definition of a TopoGeometry.

Synopsis

topogeometry **TopoGeom_addElement**(topogeometry tg, topoelement el);

Description

Adds a TopoElement to the definition of a TopoGeometry object. Does not error out if the element is already part of the definition.

Availability: 2.3

Examples

Add edge 5 to TopoGeometry `tg`.

Code

```
UPDATE mylayer SET tg = TopoGeom_addElement(tg, '{5,2}');
```

See Also

TopoGeom_remElement, CreateTopoGeom

9.12.3 TopoGeom_remElement

TopoGeom_remElement — Removes an element from the definition of a TopoGeometry.

Synopsis

topogeometry **TopoGeom_remElement**(topogeometry tg, topoelement el);

Description

Removes a TopoElement from the definition of a TopoGeometry object.

Availability: 2.3

Examples

Remove face 43 from TopoGeometry `tg`.

Code

```
UPDATE mylayer SET tg = TopoGeom_remElement(tg, '{43,3}');
```

See Also

TopoGeom_addElement, CreateTopoGeom

9.12.4 TopoGeom_addTopoGeom

TopoGeom_addTopoGeom — Adds element of a TopoGeometry to the definition of another TopoGeometry.

Synopsis

topogeometry **TopoGeom_addTopoGeom**(topogeometry tgt, topogeometry src);

Description

Adds the elements of a TopoGeometry to the definition of another TopoGeometry, possibly changing its cached type (type attribute) to a collection, if needed to hold all elements in the source object.

The two TopoGeometry objects need be defined against the *same* topology and, if hierarchically defined, need be composed by elements of the same child layer.

Availability: 3.2

Examples

Set an overall TopoGeometry value to be composed of all elements from specific TopoGeometry values.

Code

```
UPDATE mylayer SET tg_overall = TopoGeom_addTopoGeom(
    TopoGeom_addTopoGeom(
        clearTopoGeom(tg_overall),
        tg_specific1
    ),
    tg_specific2
);
```

See Also

TopoGeom_addElement, clearTopoGeom, CreateTopoGeom

9.12.5 toTopoGeom

toTopoGeom — Adds a geometry shape to an existing topo geometry.

Description

Refer to toTopoGeom.

9.13 TopoGeometry Accessors

9.13.1 GetTopoGeomElementArray

GetTopoGeomElementArray — Returns a `topoelementarray` (an array of topoelements) containing the topological elements and type of the given TopoGeometry (primitive elements).

Synopsis

topoelementarray **GetTopoGeomElementArray**(varchar toponame, integer layer_id, bigint tg_id);

topoelementarray **GetTopoGeomElementArray**(topogeometry tg);

Description

Returns a TopoElementArray containing the topological elements and type of the given TopoGeometry (primitive elements). This is similar to GetTopoGeomElements except it returns the elements as an array rather than as a dataset.

tg_id is the topogeometry id of the topogeometry object in the topology in the layer denoted by `layer_id` in the topology.layer table.

Availability: 1.1

Examples

See Also

GetTopoGeomElements, TopoElementArray

9.13.2 GetTopoGeomElements

GetTopoGeomElements — Returns a set of `topoelement` objects containing the topological element_id,element_type of the given TopoGeometry (primitive elements).

Synopsis

setof topoelement **GetTopoGeomElements**(varchar toponame, integer layer_id, bigint tg_id);

setof topoelement **GetTopoGeomElements**(topogeometry tg);

Description

Returns a set of element_id,element_type (topoelements) corresponding to primitive topology elements TopoElement (1: nodes, 2: edges, 3: faces) that a given topogeometry object in `toponame` schema is composed of.

tg_id is the topogeometry id of the topogeometry object in the topology in the layer denoted by `layer_id` in the topology.layer table.

Availability: 2.0.0

Examples

See Also

GetTopoGeomElementArray, TopoElement, TopoGeom_addElement, TopoGeom_remElement

9.13.3 ST_SRID

ST_SRID — Returns the spatial reference identifier for a topogeometry.

Synopsis

integer **ST_SRID**(topogeometry tg);

Description

Returns the spatial reference identifier of the topology containing the TopoGeometry.

The identifier is recorded for the topology when it is created and refers to an entry in Section 4.5.

Availability: 3.2.0

Examples

Return the SRID of a TopoGeometry stored in the `ri_topo` topology.

Code

```
SELECT topology.ST_SRID(topo) AS srid
FROM ri.roads
WHERE road_name = 'Unknown';
```

Output

```
 srid
------
 3438
```

See Also

Section 4.5, ST_SetSRID, ST_Transform, ST_SRID

9.14 TopoGeometry Outputs

9.14.1 AsGML

AsGML — Returns the GML representation of a topogeometry.

Synopsis

text **AsGML**(topogeometry tg);
text **AsGML**(topogeometry tg, text nsprefix_in);
text **AsGML**(topogeometry tg, regclass visitedTable);
text **AsGML**(topogeometry tg, regclass visitedTable, text nsprefix);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options, regclass visitedTable);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options, regclass visitedTable, text idprefix);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options, regclass visitedTable, text idprefix, int gmlversion);

Description

Returns the GML representation of a topogeometry in version GML3 format. If no `nsprefix_in` is specified then `gml` is used. Pass in an empty string for nsprefix to get a non-qualified name space. The precision (default: 15) and options (default 1) parameters, if given, are passed untouched to the underlying call to ST_AsGML.

The `visitedTable` parameter, if given, is used for keeping track of the visited Node and Edge elements so to use cross-references (xlink:xref) rather than duplicating definitions. The table is expected to have (at least) two integer fields: 'element_type' and 'element_id'. The calling user must have both read and write privileges on the given table. For best performance, an index should be defined on `element_type` and `element_id`, in that order. Such index would be created automatically by adding a unique constraint to the fields. Example:

Code

```
CREATE TABLE visited (
  element_type integer, element_id integer,
  unique(element_type, element_id)
);
```

The `idprefix` parameter, if given, will be prepended to Edge and Node tag identifiers.

The `gmlver` parameter, if given, will be passed to the underlying ST_AsGML. Defaults to 3.

Availability: 2.0.0

Examples

This uses the topo geometry we created in CreateTopoGeom

Code

```
SELECT topology.AsGML(topo) As rdgml
  FROM ri.roads
  WHERE road_name = 'Unknown';
```

Output

```
<gml:TopoCurve>
    <gml:directedEdge>
        <gml:Edge gml:id="E1">
            <gml:directedNode orientation="-">
                <gml:Node gml:id="N1"/>
            </gml:directedNode>
            <gml:directedNode></gml:directedNode>
            <gml:curveProperty>
                <gml:Curve srsName="urn:ogc:def:crs:EPSG::3438">
                    <gml:segments>
                        <gml:LineStringSegment>
                            <gml:posList srsDimension="2">384744 236928 384750 236923 384769 236911 384799 236895 ↩
    384811 236890
                            384833 236884 384844 236882 384866 236881 384879 236883 384954 236898 385087 236932 385117 ↩
    236938
                            385167 236938 385203 236941 385224 236946 385233 236950 385241 236956 385254 236971
                            385260 236979 385268 236999 385273 237018 385273 237037 385271 237047 385267 237057 385225 ↩
    237125
                            385210 237144 385192 237161 385167 237192 385162 237202 385159 237214 385159 237227 385162 ↩
    237241
                            385166 237256 385196 237324 385209 237345 385234 237375 385237 237383 385238 237399 385236 ↩
    237407
                            385227 237419 385213 237430 385193 237439 385174 237451 385170 237455 385169 237460 385171 ↩
    237475
                            385181 237503 385190 237521 385200 237533 385206 237538 385213 237541 385221 237542 385235 ↩
    237540 385242 237541
                            385249 237544 385260 237555 385270 237570 385289 237584 385292 237589 385291 237596 385284 ↩
    237630</gml:posList>
                        </gml:LineStringSegment>
                    </gml:segments>
                </gml:Curve>
            </gml:curveProperty>
        </gml:Edge>
    </gml:directedEdge>
</gml:TopoCurve>
```

Same exercise as previous without namespace

Code

```
SELECT topology.AsGML(topo,'') As rdgml
  FROM ri.roads
  WHERE road_name = 'Unknown';
```

Output

```
<TopoCurve>
    <directedEdge>
        <Edge id="E1">
            <directedNode orientation="-">
                <Node id="N1"/>
            </directedNode>
            <directedNode></directedNode>
            <curveProperty>
                <Curve srsName="urn:ogc:def:crs:EPSG::3438">
                    <segments>
                        <LineStringSegment>
                            <posList srsDimension="2">384744 236928 384750 236923 384769 236911 384799 236895 384811 ↩
    236890
                            384833 236884 384844 236882 384866 236881 384879 236883 384954 236898 385087 236932 385117 ↩
    236938
                            385167 236938 385203 236941 385224 236946 385233 236950 385241 236956 385254 236971
                            385260 236979 385268 236999 385273 237018 385273 237037 385271 237047 385267 237057 385225 ↩
    237125
                            385210 237144 385192 237161 385167 237192 385162 237202 385159 237214 385159 237227 385162 ↩
    237241
                            385166 237256 385196 237324 385209 237345 385234 237375 385237 237383 385238 237399 385236 ↩
    237407
                            385227 237419 385213 237430 385193 237439 385174 237451 385170 237455 385169 237460 385171 ↩
    237475
                            385181 237503 385190 237521 385200 237533 385206 237538 385213 237541 385221 237542 385235 ↩
    237540 385242 237541
                            385249 237544 385260 237555 385270 237570 385289 237584 385292 237589 385291 237596 385284 ↩
    237630</posList>
                        </LineStringSegment>
                    </segments>
                </Curve>
            </curveProperty>
        </Edge>
```

```
    </directedEdge>
</TopoCurve>
```

See Also

CreateTopoGeom, ST_CreateTopoGeo

9.14.2 AsTopoJSON

AsTopoJSON — Returns the TopoJSON representation of a topogeometry.

Synopsis

text **AsTopoJSON**(topogeometry tg, regclass edgeMapTable);

Description

Returns the TopoJSON representation of a topogeometry. If `edgeMapTable` is not null, it will be used as a lookup/storage mapping of edge identifiers to arc indices. This is to be able to allow for a compact "arcs" array in the final document.

The table, if given, is expected to have an "arc_id" field of type "serial" and an "edge_id" of type integer; the code will query the table for "edge_id" so it is recommended to add an index on that field.



Note
Arc indices in the TopoJSON output are 0-based but they are 1-based in the "edgeMapTable" table.

A full TopoJSON document will be need to contain, in addition to the snippets returned by this function, the actual arcs plus some headers. See the TopoJSON specification.

Availability: 2.1.0

Enhanced: 2.2.1 added support for puntal inputs

See Also

ST_AsGeoJSON

Examples

Create a temporary edge map for the TopoJSON output.

Code

```
CREATE TEMP TABLE edgemap(arc_id serial, edge_id int unique);
```

Build the TopoJSON header, objects, arcs, and footer.

Code

```
SELECT '{ "type": "Topology", "transform": { "scale": [1,1], "translate": [0,0] }, "objects ↩
    ": {'

UNION ALL SELECT '"' || feature_name || '": ' || AsTopoJSON(feature, 'edgemap')
FROM features.big_parcels WHERE feature_name = 'P3P4';

WITH edges AS (
  SELECT m.arc_id, e.geom FROM edgemap m, city_data.edge e
  WHERE e.edge_id = m.edge_id
), points AS (
  SELECT arc_id, (ST_DumpPoints(geom)).* FROM edges
), compare AS (
  SELECT p2.arc_id,
         CASE WHEN p1.path IS NULL THEN p2.geom
              ELSE ST_Translate(p2.geom, -ST_X(p1.geom), -ST_Y(p1.geom))
         END AS geom
  FROM points p2 LEFT OUTER JOIN points p1
  ON ( p1.arc_id = p2.arc_id AND p2.path[1] = p1.path[1]+1 )
  ORDER BY arc_id, p2.path
), arcsdump AS (
  SELECT arc_id, (regexp_matches(ST_AsGeoJSON(geom), '\[.*\]'))[1] as t
  FROM compare
), arcs AS (
  SELECT arc_id, '[' || array_to_string(array_agg(t), ',') || ']' as a FROM arcsdump
  GROUP BY arc_id
  ORDER BY arc_id
)
SELECT '}, "arcs": [' UNION ALL
SELECT array_to_string(array_agg(a), E',\n') from arcs

UNION ALL SELECT ']}'::text as t;
```

Output

```
{ "type": "Topology", "transform": { "scale": [1,1], "translate": [0,0] }, "objects": {
"P3P4": { "type": "MultiPolygon", "arcs": [[[-1]],[[6,5,-5,-4,-3,1]]]}
}, "arcs": [
 [[25,30],[6,0],[0,10],[-14,0],[0,-10],[8,0]],
 [[35,6],[0,8]],
 [[35,6],[12,0]],
 [[47,6],[0,8]],
 [[47,14],[0,8]],
 [[35,22],[12,0]],
 [[35,14],[0,8]]
 ]}
```

9.15 Topology Spatial Relationships

9.15.1 Equals

Equals — Returns true if two topogeometries are composed of the same topology primitives.

Synopsis

boolean **Equals**(topogeometry tg1, topogeometry tg2);

Description

Returns true if two topogeometries are composed of the same topology primitives: faces, edges, nodes.



Note
This function not supported for topogeometries that are geometry collections. It also can not compare topogeometries from different topologies.

Availability: 1.1.0



This function supports 3d and will not drop the z-index.

Examples

The following example uses the `nei_topo` table created in the toTopoGeom examples.

Code

```
SELECT topology.Equals(a.topo, b.topo)
FROM nei_topo AS a
JOIN nei_topo AS b ON b.gid = a.gid
WHERE a.gid = 1;
```

Output

```
t
```

See Also

GetTopoGeomElements, ST_Equals

9.15.2 Intersects

Intersects — Returns true if any pair of primitives from the two topogeometries intersect.

Synopsis

boolean **Intersects**(topogeometry tg1, topogeometry tg2);

Description

Returns true if any pair of primitives from the two topogeometries intersect.



Note
This function not supported for topogeometries that are geometry collections. It also can not compare topogeometries from different topologies. Also not currently supported for hierarchical topogeometries (topogeometries composed of other topogeometries).

Availability: 1.1.0



This function supports 3d and will not drop the z-index.

Examples

The following example uses the `nei_topo` table created in the toTopoGeom examples.

Code

```
SELECT topology.Intersects(a.topo, b.topo)
FROM nei_topo AS a
JOIN nei_topo AS b ON b.gid = a.gid
WHERE a.gid = 1;
```

Output

```
t
```

See Also

ST_Intersects

9.16 Importing and exporting Topologies

Once you have created topologies, and maybe associated topological layers, you might want to export them into a file-based format for backup or transfer into another database.

Using the standard dump/restore tools of PostgreSQL is problematic because topologies are composed by a set of tables (4 for primitives, an arbitrary number for layers) and records in metadata tables (topology.topology and topology.layer). Additionally, topology identifiers are not univoque across databases so that parameter of your topology will need to be changes upon restoring it.

In order to simplify export/restore of topologies a pair of executables are provided: `pgtopo_export` and `pgtopo_import`. Example usage:

Code

```
pgtopo_export dev_db topo1 | pgtopo_import topo1 | psql staging_db
```

9.16.1 Using the Topology exporter

The `pgtopo_export` script takes the name of a database and a topology and outputs a dump file which can be used to import the topology (and associated layers) into a new database.

By default `pgtopo_export` writes the dump file to the standard output so that it can be piped to `pgtopo_import` or redirected to a file (refusing to write to terminal). You can optionally specify an output filename with the `-f` commandline switch.

By default `pgtopo_export` includes a dump of all layers defined against the given topology. This may be more data than you need, or may be non-working (in case your layer tables have complex dependencies) in which case you can request skipping the layers with the `--skip-layers` switch and deal with those separately.

Invoking `pgtopo_export` with the `--help` (or `-h` for short) switch will always print short usage string.

The dump file format is a compressed tar archive of a `pgtopo_export` directory containing at least a `pgtopo_dump_version` file with format version info. As of version `1` the directory contains tab-delimited CSV files with data of the topology primitive tables (node, edge_data, face, relation), the topology and layer records associated with it and (unless `--skip-layers` is given) a custom-format PostgreSQL dump of tables reported as being layers of the given topology.

9.16.2 Using the Topology importer

The `pgtopo_import` script takes a `pgtopo_export` format topology dump and a name to give to the topology to be created and outputs an SQL script reconstructing the topology and associated layers.

The generated SQL file will contain statements that create a topology with the given name, load primitive data in it, restores and registers all topology layers by properly linking all TopoGeometry values to their correct topology.

By default `pgtopo_import` reads the dump from the standard input so that it can be used in conjunction with `pgtopo_export` in a pipeline. You can optionally specify an input filename with the `-f` commandline switch.

By default `pgtopo_import` includes in the output SQL file the code to restore all layers found in the dump.

This may be unwanted or non-working in case your target database already have tables with the same name as the ones in the dump. In that case you can request skipping the layers with the `--skip-layers` switch and deal with those separately (or later).

SQL to only load and link layers to a named topology can be generated using the `--only-layers` switch. This can be useful to load layers AFTER resolving the naming conflicts or to link layers to a different topology (say a spatially-simplified version of the starting topology).

If the target topology already exists and you want it dropped upfront you can pass the `--drop-topology` switch (since PostGIS-3.6.0).

Chapter 10

Raster Data Management, Queries, and Applications

10.1 Loading and Creating Rasters

For most use cases, you will create PostGIS rasters by loading existing raster files using the packaged `raster2pgsql` raster loader.

10.1.1 Using raster2pgsql to load rasters

The `raster2pgsql` is a raster loader executable that loads GDAL supported raster formats into SQL suitable for loading into a PostGIS raster table. It is capable of loading folders of raster files as well as creating overviews of rasters.

Since the raster2pgsql is compiled as part of PostGIS most often (unless you compile your own GDAL library), the raster types supported by the executable will be the same as those compiled in the GDAL dependency library. To get a list of raster types your particular `raster2pgsql` supports use the `-G` switch.



Note
When creating overviews of a specific factor from a set of rasters that are aligned, it is possible for the overviews to not align. Visit http://trac.osgeo.org/postgis/ticket/1764 for an example where the overviews do not align.

10.1.1.1 Example Usage

An example session using the loader to create an input file and uploading it chunked in 100x100 tiles might look like this:

The loader assigns SRID 4326, creates an index and standard constraints, analyzes the loaded table, records source filenames, and stores the input rasters in 100 by 100 pixel tiles.

Code

```
raster2pgsql -s 4326 -I -C -M -F -t 100x100 *.tif public.demelevation > elev.sql
psql -d gisdb -f elev.sql
```



Note
If you do not specify the schema as part of the target table name, the table will be created in the default schema of the database or user you are connecting with.

A conversion and upload can be done all in one step using UNIX pipes:

Code

```
raster2pgsql -s 4326 -I -C -M -F -t 100x100 \
  *.tif public.demelevation |
  psql -d gisdb
```

Load rasters Massachusetts state plane meters aerial tiles into a schema called `aerial` and create a full view, 2 and 4 level overview tables, use copy mode for inserting (no intermediary file just straight to db), and -e don't force everything in a transaction (good if you want to see data in tables right away without waiting). Break up the rasters into 128x128 pixel tiles and apply raster constraints. Use copy mode instead of table insert. (-F) Include a field called filename to hold the name of the file the tiles were cut from.

Code

```
raster2pgsql -I -C -e -Y -F -s 26986 -t 128x128 -l 2,4 \
  bostonaerials2008/*.jpg aerials.boston |
  psql -U postgres -d gisdb -h localhost -p 5432
```

List the raster formats supported by this build:

Code

```
raster2pgsql -G
```

The `-G` command outputs a list something like:

Output

```
Available GDAL raster formats:
  Virtual Raster
  GeoTIFF
  National Imagery Transmission Format
  Raster Product Format TOC format
  ECRG TOC format
  Erdas Imagine Images (.img)
  CEOS SAR Image
  CEOS Image
  ...
  Arc/Info Export E00 GRID
  ZMap Plus Grid
  NOAA NGS Geoid Height Grids
```

10.1.1.2 raster2pgsql options

Long options that take values also accept the `--option=value` spelling.

-?, --help Display help screen. Help will also display if you don't pass in any arguments.

-G, --gdal-formats Print the supported raster formats.

(c|a|d|p) These are mutually exclusive options:

- **-c** Create new table and populate it with raster(s), *this is the default mode*
- **-a** Append raster(s) to an existing table.
- **-d** Drop table, create new one and populate it with raster(s)
- **-p** Prepare mode, only create the table.

--if-not-exists Use `IF NOT EXISTS` for table creation in `-c` and `-p` modes. When `-I` or `--create-index` is also specified, use `IF NOT EXISTS` for index creation too. This option cannot be used with `-d`. Append mode requires an explicit creation action.

Atomic loader actions The short modes are presets over explicit actions. The equivalent long options are `--drop-table`, `--create-table`, `--load-data`, `--create-index`, `--add-constraints`, `--vacuum`, `--analyze`, and `--no-transaction`. Add `--if-not-exists` to make active creation actions use `IF NOT EXISTS`. `--drop-table` emits `DROP TABLE IF EXISTS` before the selected actions. With no mode specified, the default create/load actions still apply.

Raster processing: Applying constraints for proper registering in raster catalogs

-C, --add-constraints Apply raster constraints -- srid, pixelsize etc. to ensure raster is properly registered in `raster_columns` view.

-x, --no-extent Disable setting the max extent constraint. Only applied if -C or --add-constraints is also used.

-r, --regular-blocking Set the constraints (spatially unique and coverage tile) for regular blocking. Only applied if -C or --add-constraints is also used.

Raster processing: Optional parameters used to manipulate input raster dataset

-s, --srid [<FROM_SRID>:]<SRID> Assign the output raster to the specified SRID. When FROM_SRID is also provided, raster2pgsql marks the input raster with FROM_SRID and emits SQL that reprojects the raster to the target SRID. If the source SRID is omitted or is zero, the raster metadata will be checked to determine an appropriate source SRID. Reprojection cannot be used with copy mode.

-b, --band BAND Index (1-based) of band to extract from raster. For more than one band index, separate with comma (,). If unspecified, all bands of raster will be extracted.

-t, --tile-size TILE_SIZE Cut raster into tiles to be inserted one per table row. `TILE_SIZE` is expressed as WIDTHxHEIGHT or set to the value "auto" to allow the loader to compute an appropriate tile size using the first raster and applied to all rasters.

-P, --pad Pad right-most and bottom-most tiles to guarantee that all tiles have the same width and height.

-R, --register Register the raster as a filesystem (out-db) raster.

Only the metadata of the raster and path location to the raster is stored in the database (not the pixels).

-l, --overview-factor OVERVIEW_FACTOR Create overview of the raster. For more than one factor, separate with comma(,). Overview table name follows the pattern o_`overview factor_table`, where `overview factor` is a placeholder for numerical overview factor and `table` is replaced with the base table name. If a source raster already has an overview with matching dimensions, that overview is reused. Created overview is stored in the database and is not affected by -R. Note that your generated sql file will contain both the main table and overview tables.

-N, --nodata NODATA NODATA value to use on bands without a NODATA value.

Optional parameters used to manipulate database objects

-f, --raster-column COLUMN Specify name of destination raster column, default is 'rast'

-F, --filename Add a column with the name of the file

-n, --filename-column COLUMN Specify the name of the filename column. Implies -F.

-q, --quote Wrap PostgreSQL identifiers in quotes.

-I Alias for `--create-index`.

--create-index Create a GiST index on the raster column at the end of this raster2pgsql run. With repeated `-a` append runs, create the index on the final run or after loading; add `--if-not-exists` to make reruns tolerate an existing index.

-M Vacuum analyze the raster table.

-k, --skip-nodata-check Keeps empty tiles and skips NODATA value checks for each raster band. Note you save time in checking, but could end up with far more junk rows in your database and those junk rows are not marked as empty tiles.

-T, --tablespace tablespace Specify the tablespace for the new table. Note that indices (including the primary key) will still use the default tablespace unless the -X flag is also used.

-X, --index-tablespace tablespace Specify the tablespace for the table's new index. This applies to the primary key and the spatial index if the -I flag is used.

-Y, --copy [MAX_ROWS_PER_COPY] Use copy statements instead of insert statements. Optionally specify max_rows_p default 50 when not specified.

-e, --no-transaction Execute each statement individually, do not use a transaction.

-E, --endian ENDIAN Control endianness of generated binary output of raster; specify 0 for XDR and 1 for NDR (default); only NDR output is supported now

-V, --wkb-version version Specify version of output format. Default is 0. Only 0 is supported at this time.

10.1.2 Creating rasters using PostGIS raster functions

On many occasions, you'll want to create rasters and raster tables right in the database. There are a plethora of functions to do that. The general steps to follow.

1. Create a table with a raster column to hold the new raster records which can be accomplished with:

   Code

   ```
   CREATE TABLE myrasters(rid serial primary key, rast raster);
   ```

2. There are many functions to help with that goal. If you are creating rasters not as a derivative of other rasters, you will want to start with: ST_MakeEmptyRaster, followed by ST_AddBand

   You can also create rasters from geometries. To achieve that you'll want to use ST_AsRaster perhaps accompanied with other functions such as ST_Union or ST_MapAlgebraFct or any of the family of other map algebra functions.

   There are even many more options for creating new raster tables from existing tables. For example you can create a raster table in a different projection from an existing one using ST_Transform

3. Once you are done populating your table initially, you'll want to create a spatial index on the raster column with something like:

   Code

   ```
   CREATE INDEX myrasters_rast_st_convexhull_idx ON myrasters USING gist(ST_ConvexHull( ↩
       rast));
   ```

   Note the use of ST_ConvexHull since most raster operators are based on the convex hull of the rasters.

   Note
   Pre-2.0 versions of PostGIS raster were based on the envelop rather than the convex hull. For the spatial indexes to work properly you'll need to drop those and replace with convex hull based index.

4. Apply raster constraints using AddRasterConstraints

10.1.3 Using "out db" cloud rasters

The raster2pgsql tool uses GDAL to access raster data, and can take advantage of a key GDAL feature: the ability to read from rasters that are stored remotely in cloud "object stores" (e.g. AWS S3, Google Cloud Storage).

Efficient use of cloud stored rasters requires the use of a "cloud optimized" format. The most well-known and widely used is the "cloud optimized GeoTIFF" format. Using a non-cloud format, like a JPEG, or an un-tiled TIFF will result in very poor performance, as the system will have to download the entire raster each time it needs to access a subset.

First, load your raster into the cloud storage of your choice. Once it is loaded, you will have a URI to access it with, either an "http" URI, or sometimes a URI specific to the service. (e.g., "s3://bucket/object"). To access non-public buckets, you will need to supply GDAL config options to authenticate your connection. Note that this command is *reading* from the cloud raster and *writing* to the database.

Code

```
AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxxxxxxx \
AWS_SECRET_ACCESS_KEY=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx \
raster2pgsql \
  -s 990000 \
  -t 256x256 \
  -I \
  -R \
  /vsis3/your.bucket.com/your_file.tif \
  your_table \
  | psql your_db
```

Once the table is loaded, you need to give the database permission to read from remote rasters, by setting two permissions, postgis.enable_outdb_rasters and postgis.gdal_enabled_drivers.

Code

```
SET postgis.enable_outdb_rasters = true;
SET postgis.gdal_enabled_drivers TO 'ENABLE_ALL';
```

To make the changes sticky, set them directly on your database. You will need to re-connect to experience the new settings.

Code

```
ALTER DATABASE your_db SET postgis.enable_outdb_rasters = true;
ALTER DATABASE your_db SET postgis.gdal_enabled_drivers TO 'ENABLE_ALL';
```

For non-public rasters, you may have to provide access keys to read from the cloud rasters. The same keys you used to write the `raster2pgsql` call can be set for use inside the database, with the postgis.gdal_vsi_options configuration. Note that multiple options can be set by space-separating the `key=value` pairs.

Code

```
SET postgis.gdal_vsi_options = 'AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxxxxxxx
AWS_SECRET_ACCESS_KEY=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx';
```

Once you have the data loaded and permissions set you can interact with the raster table like any other raster table, using the same functions. The database will handle all the mechanics of connecting to the cloud data when it needs to read pixel data.

10.2 Raster Coverages and Raster/Vector Analysis

A coverage is a way to interpret spatial data as a mapping from a *domain* of locations to a *range* of values. Evaluating a coverage means asking for the value at a location; an inverse query asks for the locations whose values match a condition. PostGIS does not expose a generic ISO 19123 coverage SQL type, but geometry tables and raster values provide the building blocks for both directions.

A coverage can be continuous, with values calculated or interpolated for arbitrary locations, or discrete, with values supplied by stored spatial objects or grid cells. The coverage model describes how locations map to values, independently of whether those values come from an equation, a table, or one or more raster values. A time column or function argument can add a temporal dimension without changing that distinction.

In a vector coverage, a geometry column supplies the domain and other columns supply attributes in the range. A vector range can contain text, numbers, or another spatial value, provided its meaning and type are consistent wherever the coverage is defined. In a raster, each cell location is part of the domain and the band values at that cell form the range. Other columns in the same table row describe the raster value or tile as a whole; they do not attach separate values to individual cells.

A logical raster coverage may be stored as one raster value or as many tiles in a table. Tiles that belong to one coverage should have compatible SRIDs, scales, alignment, band counts, pixel types, and NODATA conventions. The constraints exposed by Section 10.3.1 record and validate those properties.

Raster table arrangements. A raster column can represent an image warehouse, a tiled coverage, or a collection of rasterized objects. These arrangements have different assumptions about tile size, alignment, gaps, and overlaps. PostGIS does not infer the intended arrangement from the column type alone.

Arrangement	Row relationship	Typical use
Image warehouse	Independent rasters; sizes, alignment, and extents may differ or overlap	Cataloging unrelated source images
Irregular tiled coverage	One logical coverage with gaps allowed, variable tile sizes, and normally no overlap	Incremental or source-driven mosaics
Regular tiled coverage	One alignment and tile size with no overlap; gaps may remain at the boundary or inside the coverage	Indexed analysis over a large coverage
Rectangular regular coverage	Equal aligned tiles fill a rectangular extent without gaps or overlaps	Predictable block processing and overview generation
Separate tiled images	Each image is internally regular but stored in its own table or partition	Keeping source-image identity and lifecycle separate
Raster-object coverage	Each row is a rasterized feature; extents and sizes may differ and overlap	Object models where cells replace or supplement vector geometry

Use AddRasterConstraints to record common SRID, scale, alignment, band, extent, and regular-blocking properties, and use spatial indexes and application constraints for relationships that the catalog cannot express.

Queries against a tiled coverage normally have two stages:

1. Use the raster spatial index and ST_Intersects to select candidate tiles.
2. Read, clip, aggregate, or vectorize values inside those tiles with functions such as ST_Value, ST_Clip, ST_SummaryStatsAgg, and ST_DumpAsPolygons.

ST_DumpAsPolygons returns geomval rows that pair geometries with cell values. This allows ordinary geometry overlay and filtering to answer inverse coverage queries, but polygonizing many cells is usually more expensive than keeping the operation in raster space.

Choose boundary semantics deliberately. Clipping and aggregating raster pixels is usually the shorter and faster path. If a partial boundary cell must contribute in proportion to its covered area, intersect the geometry with the geomval rows and calculate an area-weighted result in an appropriate projected coordinate system. That exact partial-cell method costs more because it vectorizes the selected cells.

Tile size affects both sides of the indexed query: smaller tiles reduce the amount of raster data inspected for each candidate row, while increasing row and index overhead. Benchmark representative queries instead of treating one tile size as a universal default.

For a current end-to-end raster/vector pattern, load and tile data with Section 10.1.1, then see the building example in ST_SummaryStats. It selects intersecting tiles, clips them to each polygon, and combines per-tile statistics into one result per feature. For point observations such as animal locations, first buffer the points in an appropriate projected coordinate system, then apply the same tile selection, clipping, and aggregation pattern to an elevation or other continuous raster coverage.

10.3 Raster Catalogs

There are two raster catalog views that come packaged with PostGIS. Both views utilize information embedded in the constraints of the raster tables. As a result the catalog views are always consistent with the raster data in the tables since the constraints are enforced.

1. raster_columns this view catalogs all the raster table columns in your database.
2. raster_overviews this view catalogs all the raster table columns in your database that serve as overviews for a finer grained table. Tables of this type are generated when you use the -l switch during load.

10.3.1 Raster Columns Catalog

The raster_columns is a catalog of all raster table columns in your database that are of type raster. It is a view utilizing the constraints on the tables so the information is always consistent even if you restore one raster table from a backup of another database. The following columns exist in the raster_columns catalog.

If you created your tables not with the loader or forgot to specify the -C flag during load, you can enforce the constraints after the fact using AddRasterConstraints so that the raster_columns catalog registers the common information about your raster tiles.

- r_table_catalog The database the table is in. This will always read the current database.
- r_table_schema The database schema the raster table belongs to.
- r_table_name raster table
- r_raster_column the column in the r_table_name table that is of type raster. There is nothing in PostGIS preventing you from having multiple raster columns per table so its possible to have a raster table listed multiple times with a different raster column for each.
- srid The spatial reference identifier of the raster. Should be an entry in the Section 4.5.
- scale_x The scaling between geometric spatial coordinates and pixel. This is only available if all tiles in the raster column have the same scale_x and this constraint is applied. Refer to ST_ScaleX for more details.
- scale_y The scaling between geometric spatial coordinates and pixel. This is only available if all tiles in the raster column have the same scale_y and the scale_y constraint is applied. Refer to ST_ScaleY for more details.
- blocksize_x The width (number of pixels across) of each raster tile . Refer to ST_Width for more details.
- blocksize_y The width (number of pixels down) of each raster tile . Refer to ST_Height for more details.
- same_alignment A boolean that is true if all the raster tiles have the same alignment . Refer to ST_SameAlignment for more details.
- regular_blocking If the raster column has the spatially unique and coverage tile constraints, the value with be TRUE. Otherwise, it will be FALSE.
- num_bands The number of bands in each tile of your raster set. This is the same information as what is provided by ST_NumBands
- pixel_types An array defining the pixel type for each band. You will have the same number of elements in this array as you have number of bands. The pixel_types are one of the following defined in ST_BandPixelType.
- nodata_values An array of double precision numbers denoting the nodata_value for each band. You will have the same number of elements in this array as you have number of bands. These numbers define the pixel value for each band that should be ignored for most operations. This is similar information provided by ST_BandNoDataValue.
- out_db An array of boolean flags indicating if the raster bands data is maintained outside the database. You will have the same number of elements in this array as you have number of bands.
- extent This is the extent of all the raster rows in your raster set. If you plan to load more data that will change the extent of the set, you'll want to run the DropRasterConstraints function before load and then reapply constraints with AddRasterConstraints after load.
- spatial_index A boolean that is true if raster column has a spatial index.

10.3.2 Raster Overviews

`raster_overviews` catalogs information about raster table columns used for overviews and additional information about them that is useful to know when utilizing overviews. Overview tables are cataloged in both `raster_columns` and `raster_overvi` because they are rasters in their own right but also serve an additional special purpose of being a lower resolution caricature of a higher resolution table. These are generated along-side the main raster table when you use the `-l` switch in raster loading or can be generated manually using AddOverviewConstraints.

Overview tables contain the same constraints as other raster tables as well as additional informational only constraints specific to overviews.



Note
The information in `raster_overviews` does not duplicate the information in `raster_columns`. If you need the information about an overview table present in `raster_columns` you can join the `raster_overviews` and `raster_columns` together to get the full set of information you need.

Two main reasons for overviews are:

1. Low resolution representation of the core tables commonly used for fast mapping zoom-out.
2. Computations are generally faster to do on them than their higher resolution parents because there are fewer records and each pixel covers more territory. Though the computations are not as accurate as the high-res tables they support, they can be sufficient in many rule-of-thumb computations.

The `raster_overviews` catalog contains the following columns of information.

- `o_table_catalog` The database the overview table is in. This will always read the current database.
- `o_table_schema` The database schema the overview raster table belongs to.
- `o_table_name` raster overview table name
- `o_raster_column` the raster column in the overview table.
- `r_table_catalog` The database the raster table that this overview services is in. This will always read the current database.
- `r_table_schema` The database schema the raster table that this overview services belongs to.
- `r_table_name` raster table that this overview services.
- `r_raster_column` the raster column that this overview column services.
- `overview_factor` - this is the pyramid level of the overview table. The higher the number the lower the resolution of the table. raster2pgsql if given a folder of images, will compute overview of each image file and load separately. Level 1 is assumed and always the original file. Level 2 is will have each tile represent 4 of the original. So for example if you have a folder of 5000x5000 pixel image files that you chose to chunk 125x125, for each image file your base table will have (5000*5000)/(125*125) records = 1600, your (l=2) `o_2` table will have ceiling(1600/Power(2,2)) = 400 rows, your (l=3) `o_3` will have ceiling(1600/Power(2,3)) = 200 rows. If your pixels aren't divisible by the size of your tiles, you'll get some scrap tiles (tiles not completely filled). Note that each overview tile generated by raster2pgsql has the same number of pixels as its parent, but is of a lower resolution where each pixel of it represents (Power(2,overview_factor) pixels of the original).

10.4 Building Custom Applications with PostGIS Raster

The fact that PostGIS raster provides you with SQL functions to render rasters in known image formats gives you a lot of options for rendering them. For example you can use OpenOffice / LibreOffice for rendering as demonstrated in Rendering PostGIS Raster graphics with LibreOffice Base Reports. In addition you can use a wide variety of languages as demonstrated in this section.

10.4.1 PHP Example Outputting using ST_AsPNG in concert with other raster functions

In this section, we'll demonstrate how to use the PHP PostgreSQL driver and the ST_AsGDALRaster family of functions to output band 1,2,3 of a raster to a PHP request stream that can then be embedded in an img src html tag.

The sample query demonstrates how to combine a whole bunch of raster functions together to grab all tiles that intersect a particular wgs 84 bounding box and then unions with ST_Union the intersecting tiles together returning all bands, transforms to user specified projection using ST_Transform, and then outputs the results as a png using ST_AsPNG.

You would call the script with `http://mywebserver/test_raster.php?srid=2249` to get the raster image in Massachusetts state plane feet.

Code

```
<?php
$conn_str = 'dbname=mydb host=localhost port=5432 user=myuser password=mypwd';
$dbconn = pg_connect($conn_str);
header('Content-Type: image/png');

if (!empty($_REQUEST['srid']) &amp;&amp; is_numeric($_REQUEST['srid'])) {
    $input_srid = intval($_REQUEST['srid']);
} else {
    $input_srid = 26986;
}

$sql = "SET bytea_output = 'escape';
SELECT ST_AsPNG(
    ST_Transform(
        ST_AddBand(
            ST_Union(rast, 1),
            ARRAY[ST_Union(rast, 2), ST_Union(rast, 3)]
        ),
        $input_srid
    )
) AS new_rast
FROM aerials.boston
WHERE ST_Intersects(
    rast,
    ST_Transform(
        ST_MakeEnvelope(-71.1217, 42.227, -71.1210, 42.218, 4326),
        26986
    )
)";

$result = pg_query($sql);
$row = pg_fetch_row($result);
pg_free_result($result);
if ($row === false) return;
echo pg_unescape_bytea($row[0]);
?>
```

10.4.2 ASP.NET C# Example Outputting using ST_AsPNG in concert with other raster functions

In this section, we'll demonstrate how to use Npgsql PostgreSQL .NET driver and the ST_AsGDALRaster family of functions to output band 1,2,3 of a raster to the HTTP response stream so it can be embedded in an `img` tag.

You will need the npgsql .NET PostgreSQL driver for this exercise which you can get the latest of from http://npgsql.projects.postgresql.o . Just download the latest and drop into your ASP.NET bin folder and you'll be good to go.

The sample query demonstrates how to combine a whole bunch of raster functions together to grab all tiles that intersect a particular wgs 84 bounding box and then unions with ST_Union the intersecting tiles together returning all bands, transforms to user specified projection using ST_Transform, and then outputs the results as a png using ST_AsPNG.

This is same example as Section 10.4.1 except implemented in C#.

You would call the handler with `http://mywebserver/TestRaster.ashx?srid=2249` to get the raster image in Massachusetts state plane feet.

web.config connection string section:

Code

```
<connectionStrings>
    <add name="DSN"
        connectionString="server=localhost;database=mydb;Port=5432;User Id=myuser;password= ↩
    mypwd"/>
</connectionStrings>
```

Code for `TestRaster.ashx`:

Code

```
<%@ WebHandler Language="C#" Class="TestRaster" %>
using System;
using System.Data;
using System.Web;
using Npgsql;

public class TestRaster : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        context.Response.ContentType = "image/png";
        context.Response.BinaryWrite(GetResults(context));
    }

    public bool IsReusable {
        get { return false; }
    }

    public byte[] GetResults(HttpContext context)
    {
        byte[] result = null;
        NpgsqlCommand command;
        string sql = null;
        int input_srid = 26986;
        try {
            string connectionString = System.Configuration.ConfigurationManager
                .ConnectionStrings["DSN"].ConnectionString;
            using (NpgsqlConnection conn = new NpgsqlConnection(connectionString)) {
                conn.Open();

                if (context.Request["srid"] != null)
                {
                    input_srid = Convert.ToInt32(context.Request["srid"]);
                }
                sql = @"SELECT ST_AsPNG(
                            ST_Transform(
                                ST_AddBand(
                                    ST_Union(rast, 1),
                                    ARRAY[ST_Union(rast, 2), ST_Union(rast, 3)]
                                ),
                                :input_srid
                            )
                        ) As new_rast
                        FROM aerials.boston
```

```
                        WHERE ST_Intersects(
                            rast,
                            ST_Transform(
                                ST_MakeEnvelope(
                                    -71.1217, 42.227,
                                    -71.1210, 42.218,
                                    4326
                                ),
                                26986
                            )
                        )";
                command = new NpgsqlCommand(sql, conn);
                command.Parameters.Add(new NpgsqlParameter("input_srid", input_srid));


                result = (byte[]) command.ExecuteScalar();
                conn.Close();
            }

        }
        catch (Exception ex)
        {
            result = null;
            context.Response.Write(ex.Message.Trim());
        }
        return result;
    }
}
```

10.4.3 Java console app that outputs raster query as Image file

This is a simple java console app that takes a query that returns one image and outputs to specified file.

You can download the latest PostgreSQL JDBC drivers from http://jdbc.postgresql.org/download.html

You can compile the following code with commands like:

Code

```
CLASSPATH=".:../postgresql.jar" javac SaveQueryImage.java
jar cfm SaveQueryImage.jar Manifest.txt *.class
```

And call it from the command line with something like:

Code

```
java -jar SaveQueryImage.jar \
  "SELECT ST_AsPNG(ST_AsRaster(ST_Buffer(ST_Point(1, 5), 10, 'quad_segs=2'), 150, 150, '8BUI ↩
    ', 100));" \
  "test.png"
```

`Manifest.txt`:

Code

```
Class-Path: postgresql-9.0-801.jdbc4.jar
Main-Class: SaveQueryImage
```

Code for `SaveQueryImage.java`:

Code

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.io.*;

public class SaveQueryImage {
  public static void main(String[] argv) {
      System.out.println("Checking if Driver is registered with DriverManager.");

      try {
        //java.sql.DriverManager.registerDriver (new org.postgresql.Driver());
        Class.forName("org.postgresql.Driver");
      }
      catch (ClassNotFoundException cnfe) {
        System.out.println("Couldn't find the driver!");
        cnfe.printStackTrace();
        System.exit(1);
      }

      Connection conn = null;

      try {
        conn = DriverManager.getConnection("jdbc:postgresql://localhost:5432/mydb", "myuser" ↩
    , "mypwd");
        conn.setAutoCommit(false);

        PreparedStatement sGetImg = conn.prepareStatement(argv[0]);

        ResultSet rs = sGetImg.executeQuery();

        FileOutputStream fout;
        try
        {
            rs.next();
            /** Output to file name requested by user **/
            fout = new FileOutputStream(new File(argv[1]) );
            fout.write(rs.getBytes(1));
            fout.close();
        }
        catch(Exception e)
        {
            System.out.println("Can't create file");
            e.printStackTrace();
        }

        rs.close();
        sGetImg.close();
        conn.close();
      }
      catch (SQLException se) {
        System.out.println("Couldn't connect: print out a stack trace and exit.");
        se.printStackTrace();
        System.exit(1);
      }
  }
}
```

10.4.4 Use PLPython to dump out images via SQL

This is a plpython stored function that creates a file in the server directory for each record. Requires you have plpython installed. Should work fine with both plpythonu and plpython3u.

Code

```
CREATE OR REPLACE FUNCTION write_file (param_bytes bytea, param_filepath text)
RETURNS text
AS $$
f = open(param_filepath, 'wb+')
f.write(param_bytes)
return param_filepath
$$ LANGUAGE plpythonu;
```

Write out 5 images to the PostgreSQL server in varying sizes. The PostgreSQL daemon account needs write access to the folder; this echoes back the file names created.

Code

```
SELECT write_file(
    ST_AsPNG(
        ST_AsRaster(
            ST_Buffer(ST_Point(1, 5), j * 5, 'quad_segs=2'),
            150 * j,
            150 * j,
            '8BUI',
            100
        )
    ),
    'C:/temp/slices' || j || '.png'
)
FROM generate_series(1, 5) AS j;
```

Output

```
    write_file
--------------------
 C:/temp/slices1.png
 C:/temp/slices2.png
 C:/temp/slices3.png
 C:/temp/slices4.png
 C:/temp/slices5.png
```

10.4.5 Outputting Rasters with PSQL

Sadly PSQL doesn't have easy to use built-in functionality for outputting binaries. This is a bit of a hack that piggy backs on PostgreSQL somewhat legacy large object support. To use it, first launch your **psql** command line connected to your database.

Unlike the python approach, this approach creates the file on your local computer.

Code

```
SELECT
    oid,
    lowrite(lo_open(oid, 131072), png) AS num_bytes
FROM (VALUES (
    lo_create(0),
    ST_AsPNG((SELECT rast FROM aerials.boston WHERE rid = 1))
)) AS v(oid, png);
```

Output

```
   oid   | num_bytes
---------+-----------
 2630819 |     74860
```

Note the large object OID and export it to a file on your local computer, replacing the example path with the desired file path.

Code

```
\lo_export 2630819 'C:/temp/aerial_samp.png'
```

This deletes the file from large object storage in the database.

Code

```
SELECT lo_unlink(2630819);
```

Chapter 11

Raster Reference

`raster` is a PostGIS type for storing and analyzing raster data.

For loading rasters from raster files please refer to Section 10.1

Some examples in this reference use a raster table of dummy data, created with the following code:

Code

```
CREATE TABLE dummy_rast(rid integer, rast raster);
INSERT INTO dummy_rast(rid, rast)
VALUES (1,
('01' -- little endian (uint8 ndr)
||
'0000' -- version (uint16 0)
||
'0000' -- nBands (uint16 0)
||
'0000000000000040' -- scaleX (float64 2)
||
'0000000000000840' -- scaleY (float64 3)
||
'000000000000E03F' -- ipX (float64 0.5)
||
'000000000000E03F' -- ipY (float64 0.5)
||
'0000000000000000' -- skewX (float64 0)
||
'0000000000000000' -- skewY (float64 0)
||
'00000000' -- SRID (int32 0)
||
'0A00' -- width (uint16 10)
||
'1400' -- height (uint16 20)
)::raster
),
-- Raster: 5 x 5 pixels, 3 bands, PT_8BUI pixel type, NODATA = 0
(2, ('01000003009A9999999999A93F9A9999999999A9BF000000E02B274A' ||
'410000000077195641000000000000000000000000000000000000 ↩
    FFFFFFFF050005000400FDFEFDFEFEFDFEFEFDF9FAFEF' ||
' ↩
    EFCF9FBFDFEFEFDFCFAFEFEFE04004E627AADD16076B4F9FE6370A9F5FE59637AB0E54F5861708704004656648 7A1506
    ')::raster);
```

The functions below are ones which a user of PostGIS Raster is likely to need. There are other raster support functions which are not of interest to a general user.

11.1 Raster Support Data types

11.1.1 geomval

geomval — A spatial datatype with two fields - geom (holding a geometry object) and val (holding a double precision pixel value from a raster band).

Description

geomval is a compound data type consisting of a geometry object referenced by the .geom field and val, a double precision value that represents the pixel value at a particular geometric location in a raster band. It is used by the ST_DumpAsPolygon and Raster intersection family of functions as an output type to explode a raster band into geometry polygons.

See Also

Section 13.6

11.1.2 addbandarg

addbandarg — A composite type used as input into the ST_AddBand function defining the attributes and initial value of the new band.

Description

A composite type used as input into the ST_AddBand function defining the attributes and initial value of the new band.

Field	Type	Description
index	integer	1-based value indicating the position where the new band will be added amongst the raster's bands. If NULL, the new band will be added at the end of the raster's bands.
pixeltype	text	Pixel type of the new band. One of the pixel types described in ST_BandPixelType.
initialvalue	double precision	Initial value that all pixels of the new band will be set to.
nodataval	double precision	NODATA value of the new band. If NULL, the new band will not have a NODATA value assigned.

See Also

ST_AddBand

11.1.3 rastbandarg

rastbandarg — A composite type for use when needing to express a raster and a band index of that raster.

Description

A composite type for use when needing to express a raster and a band index of that raster.

Field	Type	Description
rast	raster	The raster in question.
nband	integer	1-based value indicating the raster band.

See Also

ST_MapAlgebra (callback function version)

11.1.4 raster

raster — raster spatial data type.

Description

raster is a spatial data type used to represent raster data such as those imported from JPEGs, TIFFs, PNGs, digital elevation models. Each raster has 1 or more bands each having a set of pixel values. Rasters can be georeferenced.



Note
Requires PostGIS be compiled with GDAL support. Currently rasters can be implicitly converted to geometry type, but the conversion returns the ST_ConvexHull of the raster. This auto casting may be removed in the near future so don't rely on it.

Casting Behavior

This section lists the automatic as well as explicit casts allowed for this data type

Cast To	Behavior
geometry	automatic

See Also

Chapter 11

11.1.5 reclassarg

reclassarg — A composite type used as input into the ST_Reclass function defining the behavior of reclassification.

Description

A composite type used as input into the ST_Reclass function defining the behavior of reclassification.

Field	Type	Description
nband	integer	The band number of band to reclassify.

Field	Type	Description
reclassexpr	text	Comma-delimited range:map_range mappings that define how old band values map to new band values.
pixeltype	text	One of defined pixel types as described in ST_BandPixelType.
nodataval	double precision	Value to treat as no data. For image outputs that support transparency, these will be blank.

The reclassexpr range delimiters mean: (greater than,) less than,] less than or equal, and [greater than or equal.

- [a-b] means a <= x <= b.
- (a-b] means a < x <= b.
- [a-b) means a <= x < b.
- (a-b) means a < x < b.

Opening-parenthesis notation is optional, so a-b means the same as (a-b).

Examples

Reclassify band 2 as an 8BUI where 255 is nodata value.

Code

```
SELECT ROW(2, '0-100:1-10, 101-500:11-150,501 - 10000: 151-254', '8BUI', 255)::reclassarg;
```

Reclassify band 1 as an 1BB and no nodata value defined.

Code

```
SELECT ROW(1, '0-100]:0, (100-255:1', '1BB', NULL)::reclassarg;
```

See Also

ST_Reclass

11.1.6 summarystats

summarystats — A composite type returned by the ST_SummaryStats and ST_SummaryStatsAgg functions.

Description

A composite type returned by the ST_SummaryStats and ST_SummaryStatsAgg functions.

count integer Number of pixels counted for the summary statistics.

sum double precision Sum of all counted pixel values.

mean double precision Arithmetic mean of all counted pixel values.

stddev double precision Standard deviation of all counted pixel values.

min double precision Minimum value of counted pixel values.

max double precision Maximum value of counted pixel values.

See Also

ST_SummaryStats, ST_SummaryStatsAgg

11.1.7 unionarg

unionarg — A composite type used as input into the ST_Union function defining the bands to be processed and behavior of the UNION operation.

Description

A composite type used as input into the ST_Union function defining the bands to be processed and behavior of the UNION operation.

Field	Type	Description
`nband`	integer	1-based value indicating the band of each input raster to be processed.
`uniontype`	text	Type of union operation. One of the types described in ST_Union.

See Also

ST_Union

11.2 Raster Management

11.2.1 AddRasterConstraints

AddRasterConstraints — Adds raster constraints to a loaded raster table for a specific column that constrains spatial ref, scaling, blocksize, alignment, bands, band type and a flag to denote if raster column is regularly blocked. The table must be loaded with data for the constraints to be inferred. Returns true if the constraint setting was accomplished and issues a notice otherwise.

Synopsis

boolean **AddRasterConstraints**(name rasttable, name rastcolumn, boolean srid=true, boolean scale_x=true, boolean scale_y=true, boolean blocksize_x=true, boolean blocksize_y=true, boolean same_alignment=true, boolean regular_blocking=false, boolean num_bands=true , boolean pixel_types=true , boolean nodata_values=true , boolean out_db=true , boolean extent=true);
boolean **AddRasterConstraints**(name rasttable, name rastcolumn, text[] VARIADIC constraints);
boolean **AddRasterConstraints**(name rastschema, name rasttable, name rastcolumn, text[] VARIADIC constraints);
boolean **AddRasterConstraints**(name rastschema, name rasttable, name rastcolumn, boolean srid=true, boolean scale_x=true, boolean scale_y=true, boolean blocksize_x=true, boolean blocksize_y=true, boolean same_alignment=true, boolean regular_blocking=fa
boolean num_bands=true, boolean pixel_types=true, boolean nodata_values=true , boolean out_db=true , boolean extent=true);

Description

Generates constraints on a raster column that are used to display information in the `raster_columns` raster catalog. The `rastschema` is the name of the table schema the table resides in. The `srid` must be an integer value reference to an entry in the SPATIAL_REF_SYS table.

`raster2pgsql` loader uses this function to register raster tables

Valid constraint names to pass in: refer to Section 10.3.1 for more details.

- `blocksize` sets both X and Y blocksize
- `blocksize_x` sets X tile (width in pixels of each tile)
- `blocksize_y` sets Y tile (height in pixels of each tile)
- `extent` computes extent of whole table and applies constraint all rasters must be within that extent
- `num_bands` number of bands
- `pixel_types` reads array of pixel types for each band ensure all band n have same pixel type
- `regular_blocking` sets spatially unique (no two rasters can be spatially the same) and coverage tile (raster is aligned to a coverage) constraints
- `same_alignment` ensures they all have same alignment meaning any two tiles you compare will return true for. Refer to ST_SameAlignment.
- `srid` ensures all have same srid
- More -- any listed as inputs into the above functions



Note
This function infers the constraints from the data already present in the table. As such for it to work, you must create the raster column first and then load it with data.



Note
If you need to load more data in your tables after you have already applied constraints, you may want to run the DropRasterConstraints if the extent of your data has changed.

Availability: 2.0.0

Examples

Apply all possible constraints on column based on data.

Verify the raster table is registered correctly in the raster_columns view.

Code

```
CREATE TABLE myrasters(rid SERIAL primary key, rast raster);
INSERT INTO myrasters(rast)
SELECT ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0.3, -0.3, 2, 2, 0, 0, 4326), 1, '8BSI':: ↩
    text, -129, NULL);

SELECT AddRasterConstraints('myrasters'::name, 'rast'::name);
SELECT srid, scale_x, scale_y, blocksize_x, blocksize_y, num_bands, pixel_types,  ↩
    nodata_values
    FROM raster_columns
    WHERE r_table_name = 'myrasters';
```

Output

```
 srid | scale_x | scale_y | blocksize_x | blocksize_y | num_bands | pixel_types| nodata_values
------+---------+---------+-------------+-------------+-----------+-------------+---------------
 4326 |       2 |       2 |        1000 |        1000 |         1 | {8BSI}      | {0}
```

Apply single constraint.

Code

```
CREATE TABLE public.myrasters2(rid SERIAL primary key, rast raster);
INSERT INTO myrasters2(rast)
SELECT ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0.3, -0.3, 2, 2, 0, 0, 4326), 1, '8BSI':: ↩
    text, -129, NULL);

SELECT AddRasterConstraints('public'::name, 'myrasters2'::name, 'rast'::name, ' ↩
    regular_blocking', 'blocksize');
```

Output

```
NOTICE:  Adding regular blocking constraint
NOTICE:  Adding blocksize-X constraint
NOTICE:  Adding blocksize-Y constraint
```

See Also

Section 10.3.1, ST_AddBand, ST_MakeEmptyRaster, DropRasterConstraints, ST_BandPixelType, ST_SRID

11.2.2 DropRasterConstraints

DropRasterConstraints — Drops PostGIS raster constraints that refer to a raster table column. Useful if you need to reload data or update your raster column data.

Synopsis

boolean **DropRasterConstraints**(name rasttable, name rastcolumn, boolean srid, boolean scale_x, boolean scale_y, boolean blocksize_x, boolean blocksize_y, boolean same_alignment, boolean regular_blocking, boolean num_bands=true, boolean pixel_types=t
boolean nodata_values=true, boolean out_db=true , boolean extent=true);
boolean **DropRasterConstraints**(name rastschema, name rasttable, name rastcolumn, boolean srid=true, boolean scale_x=true, boolean scale_y=true, boolean blocksize_x=true, boolean blocksize_y=true, boolean same_alignment=true, boolean regular_blocking=fa
boolean num_bands=true, boolean pixel_types=true, boolean nodata_values=true, boolean out_db=true , boolean extent=true);
boolean **DropRasterConstraints**(name rastschema, name rasttable, name rastcolumn, text[] constraints);

Description

Drops PostGIS raster constraints that refer to a raster table column that were added by AddRasterConstraints. Useful if you need to load more data or update your raster column data. You do not need to do this if you want to get rid of a raster table or a raster column.

To drop a raster table use standard SQL: `DROP TABLE mytable`.

To drop just a raster column and leave the rest of the table, use standard SQL: `ALTER TABLE mytable DROP COLUMN rast`.

the table will disappear from the `raster_columns` catalog if the column or table is dropped. However if only the constraints are dropped, the raster column will still be listed in the `raster_columns` catalog, but there will be no other information about it aside from the column name and table.

Availability: 2.0.0

Examples

Code

```
SELECT DropRasterConstraints ('myrasters', 'rast');
```

Output

```
 droprasterconstraints
-----------------------
 t
(1 row)
```

Verify change in raster_columns.

Code

```
SELECT srid, scale_x, scale_y, blocksize_x, blocksize_y, num_bands, pixel_types,  ↩
    nodata_values
    FROM raster_columns
    WHERE r_table_name = 'myrasters';
```

Output

```
 srid | scale_x | scale_y | blocksize_x | blocksize_y | num_bands | pixel_types| nodata_values
------+---------+---------+-------------+-------------+-----------+-------------+---------------
    0 |         |         |             |             |           |             |
```

See Also

AddRasterConstraints

11.2.3 AddOverviewConstraints

AddOverviewConstraints — Tag a raster column as being an overview of another.

Synopsis

boolean **AddOverviewConstraints**(name ovschema, name ovtable, name ovcolumn, name refschema, name reftable, name refcolumn, int ovfactor);
boolean **AddOverviewConstraints**(name ovtable, name ovcolumn, name reftable, name refcolumn, int ovfactor);

Description

Adds constraints on a raster column that are used to display information in the `raster_overviews` raster catalog.

The `ovfactor` parameter represents the scale multiplier in the overview column: higher overview factors have lower resolution.

When the `ovschema` and `refschema` parameters are omitted, the first table found scanning the `search_path` will be used.

Availability: 2.0.0

Examples

Verify the raster table is registered correctly in the raster_overviews view.

Code

```
CREATE TABLE res1 AS SELECT
ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 2),
  1, '8BSI'::text, -129, NULL
) r1;

CREATE TABLE res2 AS SELECT
ST_AddBand(ST_MakeEmptyRaster(500, 500, 0, 0, 4),
  1, '8BSI'::text, -129, NULL
) r2;

SELECT AddOverviewConstraints('res2', 'r2', 'res1', 'r1', 2);
SELECT o_table_name ot, o_raster_column oc,
       r_table_name rt, r_raster_column rc,
       overview_factor f
FROM raster_overviews WHERE o_table_name = 'res2';
```

Output

```
  ot  | oc |  rt  | rc | f
------+----+------+----+---
 res2 | r2 | res1 | r1 | 2
(1 row)
```

See Also

Section 10.3.2, DropOverviewConstraints, ST_CreateOverview, AddRasterConstraints

11.2.4 DropOverviewConstraints

DropOverviewConstraints — Untag a raster column from being an overview of another.

Synopsis

boolean **DropOverviewConstraints**(name ovschema, name ovtable, name ovcolumn);
boolean **DropOverviewConstraints**(name ovtable, name ovcolumn);

Description

Remove from a raster column the constraints used to show it as being an overview of another in the `raster_overviews` raster catalog.

When the `ovschema` parameter is omitted, the first table found scanning the `search_path` will be used.

Availability: 2.0.0

See Also

Section 10.3.2, AddOverviewConstraints, DropRasterConstraints

11.2.5 PostGIS_GDAL_Version

PostGIS_GDAL_Version — Reports the version of the GDAL library in use by PostGIS.

Synopsis

text **PostGIS_GDAL_Version**();

Description

Reports the version of the GDAL library in use by PostGIS. Will also check and report if GDAL can find its data files.

Examples

Code

```
SELECT PostGIS_GDAL_Version();
```

Output

```
GDAL 1.11dev, released 2013/04/13
```

See Also

postgis.gdal_datapath

11.2.6 PostGIS_Raster_Lib_Build_Date

PostGIS_Raster_Lib_Build_Date — Reports full raster library build date.

Synopsis

text **PostGIS_Raster_Lib_Build_Date**();

Description

Reports raster build date

Examples

Code

```
SELECT PostGIS_Raster_Lib_Build_Date();
```

Output

```
2010-04-28 21:15:10
```

See Also

PostGIS_Raster_Lib_Version

11.2.7 PostGIS_Raster_Lib_Version

PostGIS_Raster_Lib_Version — Reports full raster version and build configuration infos.

Synopsis

text **PostGIS_Raster_Lib_Version**();

Description

Reports full raster version and build configuration infos.

Examples

Code

```
SELECT PostGIS_Raster_Lib_Version();
```

Output

```
2.0.0
```

See Also

PostGIS_Lib_Version

11.2.8 ST_GDALDrivers

ST_GDALDrivers — Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster

Synopsis

setof record **ST_GDALDrivers**(integer OUT idx, text OUT short_name, text OUT long_name, text OUT can_read, text OUT can_write, text OUT create_options);

Description

Returns a list of raster formats short_name,long_name and creator options of each format supported by GDAL. Use the short_name as input in the `format` parameter of ST_AsGDALRaster. Options vary depending on what drivers your libgdal was compiled with. `create_options` returns an xml formatted set of CreationOptionList/Option consisting of name and optional `type`, `description` and set of `VALUE` for each creator option for the specific driver.

Changed: 2.5.0 - add can_read and can_write columns.

Changed: 2.0.6, 2.1.3 - by default no drivers are enabled, unless GUC or Environment variable gdal_enabled_drivers is set.

Availability: 2.0.0 - requires GDAL >= 1.6.0.

Examples

List of Drivers.

Code

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';
SELECT short_name, long_name, can_write
FROM ST_GDALDrivers()
ORDER BY short_name;
```

Output

```
   short_name    |                          long_name                          | can_write
-----------------+-------------------------------------------------------------+-----------
 AAIGrid         | Arc/Info ASCII Grid                                         | t
 ACE2            | ACE2                                                        | f
 ADRG            | ARC Digitized Raster Graphics                               | f
 AIG             | Arc/Info Binary Grid                                        | f
 AirSAR          | AirSAR Polarimetric Image                                   | f
 ARG             | Azavea Raster Grid format                                   | t
 BAG             | Bathymetry Attributed Grid                                  | f
 BIGGIF          | Graphics Interchange Format (.gif)                          | f
 BLX             | Magellan topo (.blx)                                        | t
 BMP             | MS Windows Device Independent Bitmap                        | f
 BSB             | Maptech BSB Nautical Charts                                 | f
 PAux            | PCI .aux Labelled                                           | f
 PCIDSK          | PCIDSK Database File                                        | f
 PCRaster        | PCRaster Raster File                                        | f
 PDF             | Geospatial PDF                                              | f
 PDS             | NASA Planetary Data System                                  | f
 PDS4            | NASA Planetary Data System 4                                | t
 PLMOSAIC        | Planet Labs Mosaics API                                     | f
 PLSCENES        | Planet Labs Scenes API                                      | f
 PNG             | Portable Network Graphics                                   | t
 PNM             | Portable Pixmap Format (netpbm)                             | f
 PRF             | Racurs PHOTOMOD PRF                                         | f
 R               | R Object Data Store                                         | t
 Rasterlite      | Rasterlite                                                  | t
 RDA             | DigitalGlobe Raster Data Access driver                      | f
 RIK             | Swedish Grid RIK (.rik)                                     | f
 RMF             | Raster Matrix Format                                        | f
 ROI_PAC         | ROI_PAC raster                                              | f
 RPFTOC          | Raster Product Format TOC format                            | f
 RRASTER         | R Raster                                                    | f
 RS2             | RadarSat 2 XML Product                                      | f
 RST             | Idrisi Raster A.1                                           | t
 SAFE            | Sentinel-1 SAR SAFE Product                                 | f
 SAGA            | SAGA GIS Binary Grid (.sdat, .sg-grd-z)                     | t
 SAR_CEOS        | CEOS SAR Image                                              | f
 SDTS            | SDTS Raster                                                 | f
 SENTINEL2       | Sentinel 2                                                  | f
 SGI             | SGI Image File Format 1.0                                   | f
 SNODAS          | Snow Data Assimilation System                               | f
 SRP             | Standard Raster Product (ASRP/USRP)                         | f
 SRTMHGT         | SRTMHGT File Format                                         | t
 Terragen        | Terragen heightfield                                        | f
 TIL             | EarthWatch .TIL                                             | f
 TSX             | TerraSAR-X Product                                          | f
 USGSDEM         | USGS Optional ASCII DEM (and CDED)                          | t
 VICAR           | MIPL VICAR file                                             | f
 VRT             | Virtual Raster                                              | t
 WCS             | OGC Web Coverage Service                                    | f
 WMS             | OGC Web Map Service                                         | t
 WMTS            | OGC Web Map Tile Service                                    | t
 XPM             | X11 PixMap Format                                           | t
 XYZ             | ASCII Gridded XYZ                                           | t
 ZMap            | ZMap Plus Grid                                              | t
```

List of options for each driver.

Output the create options XML column of JPEG as a table. These creator options can be used in the ST_AsGDALRaster options argument.

Code

```
SELECT (xpath('@name', g.opt))[1]::text As oname,
       (xpath('@type', g.opt))[1]::text As otype,
       (xpath('@description', g.opt))[1]::text As descrip
FROM (SELECT unnest(xpath('/CreationOptionList/Option', create_options::xml)) As opt
FROM  ST_GDALDrivers()
WHERE short_name = 'JPEG') As g;
```

Output

```
       oname        |  otype  |      descrip
--------------------+---------+-----------------------------------------------
 PROGRESSIVE        | boolean | whether to generate a progressive JPEG
 QUALITY            | int     | good=100, bad=0, default=75
 WORLDFILE          | boolean | whether to generate a worldfile
```

```
 INTERNAL_MASK      | boolean | whether to generate a validity mask
 COMMENT            | string  | Comment
 SOURCE_ICC_PROFILE | string  | ICC profile encoded in Base64
 EXIF_THUMBNAIL     | boolean | whether to generate an EXIF thumbnail(overview).
                                By default its max dimension will be 128
 THUMBNAIL_WIDTH    | int     | Forced thumbnail width
 THUMBNAIL_HEIGHT   | int     | Forced thumbnail height
(9 rows)
```

This example turns a few common GeoTIFF creator options into a compact table.

Code

```
WITH gtiff_options AS (
    SELECT
        option_name,
        option_type,
        default_value
    FROM ST_GDALDrivers() AS driver
    CROSS JOIN LATERAL XMLTABLE(
        '/CreationOptionList/Option'
        PASSING (driver.create_options::xml)
        COLUMNS
            option_name text PATH '@name',
            option_type text PATH '@type',
            default_value text PATH '@default'
    ) AS options
    WHERE driver.short_name = 'GTiff'
)
SELECT option_name AS name, option_type AS type, coalesce(default_value, '') AS ↩
    default_value
FROM gtiff_options
WHERE option_name = ANY (ARRAY[
    'COMPRESS',
    'INTERLEAVE',
    'TILED',
    'BLOCKXSIZE',
    'BLOCKYSIZE',
    'JPEG_QUALITY'
])
ORDER BY 1;
```

Output

```
     name      |     type      | default_value
---------------+---------------+---------------
 BLOCKXSIZE    | int           |
 BLOCKYSIZE    | int           |
 COMPRESS      | string-select |
 INTERLEAVE    | string-select | PIXEL
 JPEG_QUALITY  | int           | 75
 TILED         | boolean       |
(6 rows)
```

Use `XMLTABLE` to expose selected GTiff creation options as rows. The same pattern can be used without the name filter to inspect the full XML document.

Code

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';

SELECT option_name, option_type
FROM ST_GDALDrivers() AS driver
CROSS JOIN LATERAL XMLTABLE(
    '/CreationOptionList/Option'
    PASSING (driver.create_options::xml)
```

```
    COLUMNS
        option_name text PATH '@name',
        option_type text PATH '@type'
) AS options
WHERE driver.short_name = 'GTiff'
  AND option_name IN ('COMPRESS', 'TILED', 'BIGTIFF')
ORDER BY option_name;
```

Output

```
 option_name |  option_type
-------------+--------------
 BIGTIFF     | string-select
 COMPRESS    | string-select
 TILED       | boolean
(3 rows)
```

See Also

ST_AsGDALRaster, ST_SRID, postgis.gdal_enabled_drivers

11.2.9 UpdateRasterSRID

UpdateRasterSRID — Change the SRID of all rasters in the user-specified column and table.

Synopsis

raster **UpdateRasterSRID**(name schema_name, name table_name, name column_name, integer new_srid);
raster **UpdateRasterSRID**(name table_name, name column_name, integer new_srid);

Description

Change the SRID of all rasters in the user-specified column and table. The function will drop all appropriate column constraints (extent, alignment and SRID) before changing the SRID of the specified column's rasters.



Note
The data (band pixel values) of the rasters are not touched by this function. Only the raster's metadata is changed.

Availability: 2.1.0

See Also

UpdateGeometrySRID

11.2.10 ST_CreateOverview

ST_CreateOverview — Create an reduced resolution version of a given raster coverage.

Synopsis

regclass **ST_CreateOverview**(regclass tab, name col, int factor, text algo='NearestNeighbor');

Description

Create an overview table with resampled tiles from the source table. Output tiles will have the same size of input tiles and cover the same spatial extent with a lower resolution (pixel size will be 1/`factor` of the original in both directions).

The overview table will be made available in the `raster_overviews` catalog and will have raster constraints enforced.

Algorithm options are: 'NearestNeighbor', 'Bilinear', 'Cubic', 'CubicSpline', and 'Lanczos'. Refer to: GDAL Warp resampling methods for more details.

Availability: 2.2.0

Example

Output to generally better quality but slower to product format

Code

```
SELECT ST_CreateOverview('mydata.mytable'::regclass, 'rast', 2, 'Lanczos');
```

Output to faster to process default nearest neighbor

Code

```
SELECT ST_CreateOverview('mydata.mytable'::regclass, 'rast', 2);
```

See Also

ST_Retile, AddOverviewConstraints, AddRasterConstraints, Section 10.3.2

11.3 Raster Constructors

11.3.1 ST_AddBand

ST_AddBand — Returns a raster with the new band(s) of given type added with given initial value in the given index location. If no index is specified, the band is added to the end.

Synopsis

(1) raster **ST_AddBand**(raster rast, addbandarg[] addbandargset);
(2) raster **ST_AddBand**(raster rast, integer index, text pixeltype, double precision initialvalue=0, double precision nodataval=NULL);
(3) raster **ST_AddBand**(raster rast, text pixeltype, double precision initialvalue=0, double precision nodataval=NULL);
(4) raster **ST_AddBand**(raster torast, raster fromrast, integer fromband=1, integer torastindex=at_end);
(5) raster **ST_AddBand**(raster torast, raster[] fromrasts, integer fromband=1, integer torastindex=at_end);
(6) raster **ST_AddBand**(raster rast, integer index, text outdbfile, integer[] outdbindex, double precision nodataval=NULL);
(7) raster **ST_AddBand**(raster rast, text outdbfile, integer[] outdbindex, integer index=at_end, double precision nodataval=NULL);

Description

Returns a raster with a new band added in given position (index), of given type, of given initial value, and of given nodata value. If no index is specified, the band is added to the end. If no `fromband` is specified, band 1 is assumed. Pixel type is a string representation of one of the pixel types specified in ST_BandPixelType. If an existing index is specified all subsequent bands >= that index are incremented by 1. If an initial value greater than the max of the pixel type is specified, then the initial value is set to the highest value allowed by the pixel type.

For the variant that takes an array of addbandarg (Variant 1), a specific addbandarg's index value is relative to the raster at the time when the band described by that addbandarg is being added to the raster. See the Multiple New Bands example below.

For the variant that takes an array of rasters (Variant 5), if `torast` is NULL then the `fromband` band of each raster in the array is accumulated into a new raster.

For the variants that take `outdbfile` (Variants 6 and 7), the value must include the full path to the raster file. The file must also be accessible to the postgres server process.

Enhanced: 2.1.0 support for addbandarg added.

Enhanced: 2.1.0 support for new out-db bands added.

Examples

Single New Band.

This example adds another 8-bit unsigned integer band with pixels initialized to 200.

Code

```
UPDATE dummy_rast
    SET rast = ST_AddBand(rast, '8BUI'::text, 200)
WHERE rid = 1;
```

Create an empty raster 100x100 units, with upper left right at 0, add 2 bands (band 1 is 0/1 boolean bit switch, band2 allows values 0-15) Uses addbandargs.

Output metadata for the raster bands to verify they are correct.

Code

```
INSERT INTO dummy_rast(rid, rast)
    VALUES(10, ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 1, -1, 0, 0, 0),
    ARRAY[
        ROW(1, '1BB'::text, 0, NULL),
        ROW(2, '4BUI'::text, 0, NULL)
            ]::addbandarg[]
     )
    );

SELECT  (bmd).*
FROM (SELECT ST_BandMetaData(rast, generate_series(1, 2)) As bmd
    FROM dummy_rast WHERE rid = 10) AS foo;
```

Output

```
 pixeltype | nodatavalue | isoutdb | path
-----------+-------------+---------+------
 1BB       |             | f       |
 4BUI      |             | f       |
```

This example outputs metadata for the raster.

Code

```
SELECT  (rmd).width, (rmd).height, (rmd).numbands
FROM (SELECT ST_MetaData(rast) As rmd
    FROM dummy_rast WHERE rid = 10) AS foo;
```

Output

```
 width | height | numbands
-------+--------+----------
   100 |    100 |        2
```

Multiple New Bands.

Code

```
SELECT
    *
FROM ST_BandMetaData(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, -1, 0, 0, 0),
        ARRAY[
            ROW(NULL, '8BUI', 255, 0),
            ROW(NULL, '16BUI', 1, 2),
            ROW(2, '32BUI', 100, 12),
            ROW(2, '32BF', 3.14, -1)
        ]::addbandarg[]
    ),
    ARRAY[]::integer[]
);
```

Output

```
 bandnum | pixeltype | nodatavalue | isoutdb | path | outdbbandnum | filesize | filetimestamp
---------+-----------+-------------+---------+------+--------------+----------+---------------
       1 | 8BUI      |           0 | f       | null |         null |     null |          null
       2 | 32BF      |          -1 | f       | null |         null |     null |          null
       3 | 32BUI     |          12 | f       | null |         null |     null |          null
       4 | 16BUI     |           2 | f       | null |         null |     null |          null
```

Aggregate the 1st band of a table of like rasters into a single raster. With as many bands as there are test_types and as many rows (new rasters) as there are mice. Note that The ORDER BY test_type clause relies on aggregate ORDER BY support. The resulting raster will have a band for each test_type alphabetical by test_type. For mouse lovers: No mice were harmed in this exercise.

Code

```
SELECT
    mouse,
    ST_AddBand(NULL, array_agg(rast ORDER BY test_type), 1) As rast
FROM mice_studies
GROUP BY mouse;
```

New Out-db band.

Code

```
SELECT
    *
FROM ST_BandMetaData(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, -1, 0, 0, 0),
        '/home/raster/mytestraster.tif'::text, NULL::int[]
    ),
    ARRAY[]::integer[]
);
```

Output

```
 bandnum | pixeltype | nodatavalue | isoutdb | path
---------+-----------+-------------+---------+------
       1 | 8BUI      |             | t       | /home/raster/mytestraster.tif
       2 | 8BUI      |             | t       | /home/raster/mytestraster.tif
       3 | 8BUI      |             | t       | /home/raster/mytestraster.tif
```

See Also

ST_BandMetaData, ST_BandPixelType, ST_MakeEmptyRaster, ST_MetaData, ST_NumBands, ST_Reclass

11.3.2 ST_AsRaster

ST_AsRaster — Converts a PostGIS geometry to a PostGIS raster.

Synopsis

raster **ST_AsRaster**(geometry geom, raster ref, text pixeltype, double precision value=1, double precision nodataval=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, raster ref, text[] pixeltype=ARRAY['8BUI'], double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, double precision gridx, double precision gridy, text pixeltype, double precision value=1, double precision nodataval=0, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, double precision gridx=NULL, double precision gridy=NULL, text[] pixeltype=ARRAY['8BUI'], double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, text pixeltype, double precision value=1, double precision nodataval=0, double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, text[] pixeltype, double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, double precision gridx, double precision gridy, text pixeltype, double precision value=1, double precision nodataval=0, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, double precision gridx=NULL, double precision gridy=NULL, text[] pixeltype=ARRAY['8BUI'], double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, text pixeltype, double precision value=1, double precision nodataval=0, double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, text[] pixeltype, double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);

Description

Converts a PostGIS geometry to a PostGIS raster. The many variants offers three groups of possibilities for setting the alignment and pixelsize of the resulting raster.

The first group, composed of the two first variants, produce a raster having the same alignment (`scalex`, `scaley`, `gridx` and `gridy`), pixel type and nodata value as the provided reference raster. You generally pass this reference raster by joining the table containing the geometry with the table containing the reference raster.

The second group, composed of four variants, let you set the dimensions of the raster by providing the parameters of a pixel size (`scalex` & `scaley` and `skewx` & `skewy`). The `width` & `height` of the resulting raster will be adjusted to fit the extent of the geometry. In most cases, you must cast integer `scalex` & `scaley` arguments to double precision so that PostgreSQL choose the right variant.

The third group, composed of four variants, let you fix the dimensions of the raster by providing the dimensions of the raster (`width` & `height`). The parameters of the pixel size (`scalex` & `scaley` and `skewx` & `skewy`) of the resulting raster will be adjusted to fit the extent of the geometry.

The two first variants of each of those two last groups let you specify the alignment with an arbitrary corner of the alignment grid (`gridx` & `gridy`) and the two last variants takes the upper left corner (`upperleftx` & `upperlefty`).

Each group of variant allows producing a one band raster or a multiple bands raster. To produce a multiple bands raster, you must provide an array of pixel types (`pixeltype[]`), an array of initial values (`value`) and an array of nodata values (`nodataval`). If not provided pixeltyped defaults to 8BUI, values to 1 and nodataval to 0.

The output raster will be in the same spatial reference as the source geometry. The only exception is for variants with a reference raster. In this case the resulting raster will get the same SRID as the reference raster.

The optional `touched` parameter defaults to false and maps to the GDAL ALL_TOUCHED rasterization option, which determines if pixels touched by lines or polygons will be burned. Not just those on the line render path, or whose center point is within the polygon.

This is particularly useful for rendering jpegs and pngs of geometries directly from the database when using in combination with ST_AsPNG and other ST_AsGDALRaster family of functions.

Availability: 2.0.0 - requires GDAL >= 1.6.0.

Curved geometry inputs are linearized before rasterization, including NURBSCurve inputs handled by PostGIS. TIN and PolyhedralSurface inputs are not supported.

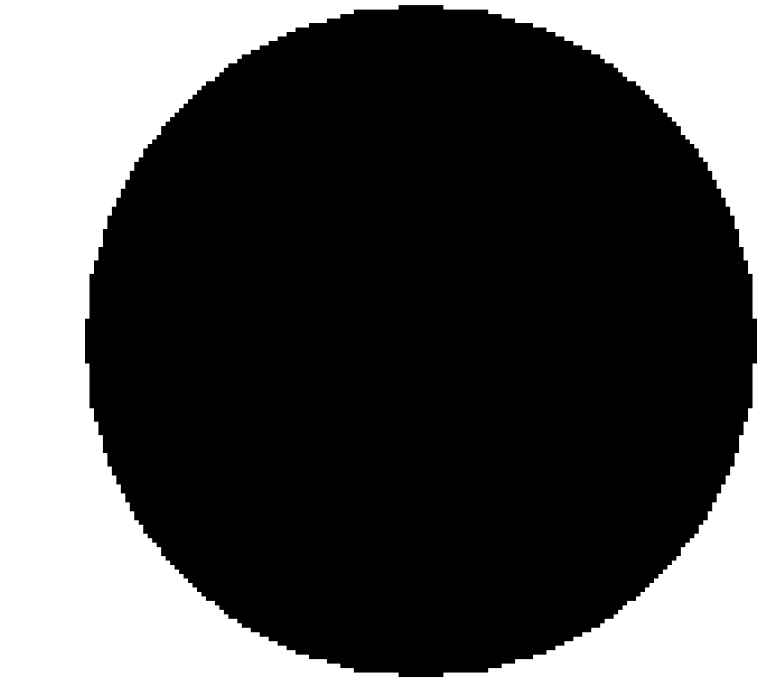
Examples: Output geometries as PNG files

This example outputs a black circle on a white background, occupying 150 by 150 pixels.

Code

```
SELECT ST_AsPNG(
    ST_AsRaster(
        ST_Buffer(ST_Point(1, 5), 10),
        150, 150,
        ARRAY['8BUI', '8BUI', '8BUI'],
        ARRAY[0, 0, 0],
        ARRAY[255, 255, 255]
    )
) AS image;
```

Figure



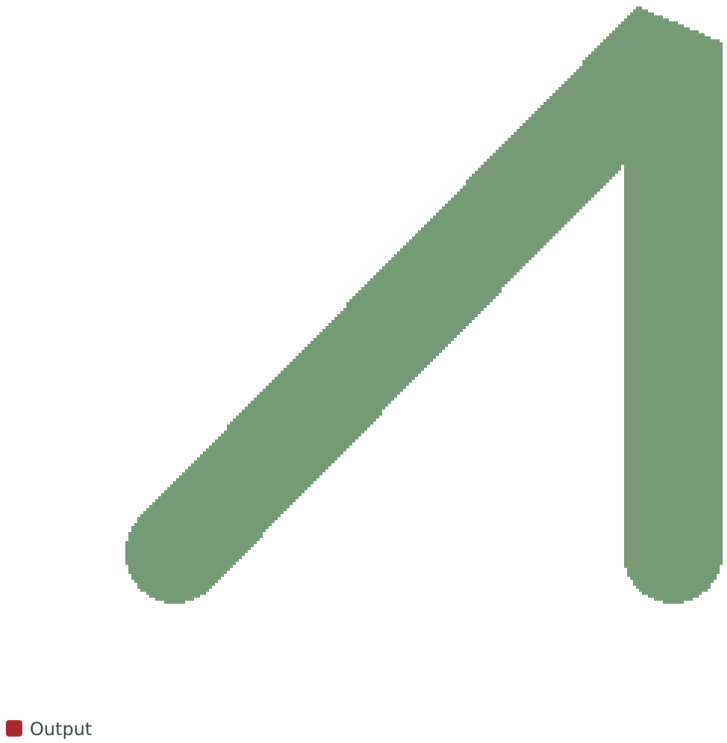
Output

This example maps bands to RGB channels, producing the teal value (118,154,118).

Code

```
SELECT ST_AsPNG(
    ST_AsRaster(
        ST_Buffer('LINESTRING(50 50,150 150,150 50)'::geometry, 10, 'join=bevel'),
        200, 200,
        ARRAY['8BUI', '8BUI', '8BUI'],
        ARRAY[118, 154, 118],
        ARRAY[255, 255, 255]
    )
) AS image;
```

Figure



See Also

ST_BandPixelType, ST_Buffer, ST_GDALDrivers, ST_AsGDALRaster, ST_AsPNG, ST_AsJPEG, ST_SRID

11.3.3 ST_AsRasterAgg

ST_AsRasterAgg — Aggregate. Renders PostGIS geometries into a new raster.

Synopsis

raster **ST_AsRasterAgg**(geometry geom, double precision val, raster ref, text pixeltype, double precision nodataval, text uniontype, boolean touched);

Description

Returns a single-band raster containing the rendered version of all incoming geometries, each with its associated value.

Availability: 3.6.0

Examples

Code

```
WITH inp(g, v) AS (
    VALUES
        ( ST_Buffer(ST_MakePoint(10, 0), 10), 1 ),
        ( ST_Buffer(ST_MakePoint(20, 0), 10), 2 )
),
agg AS (
    SELECT ST_AsRasterAgg(
        g,
        v,
        ST_MakeEmptyRaster(0, 0, 0, 0, 1.0),
        '8BUI',
        99,
        'SUM',
        true
    ) r
    FROM inp
)
SELECT
    ST_Width(r) w,
    ST_Height(r) h,
    ST_Value(r, 'POINT(5 0)') v5_0,
    ST_Value(r, 'POINT(15 0)') v15_0,
    ST_Value(r, 'POINT(25 0)') v25_0
FROM agg;
```

Output

```
 w  | h  | v5_0 | v15_0 | v25_0
----+----+------+-------+-------
 30 | 20 |    1 |     3 |     2
(1 row)
```

See Also

ST_AsRaster, ST_DumpAsPolygons, ST_Union

11.3.4 ST_Band

ST_Band — Returns one or more bands of an existing raster as a new raster. Useful for building new rasters from existing rasters.

Synopsis

raster **ST_Band**(raster rast, integer[] nbands = ARRAY[1]);
raster **ST_Band**(raster rast, integer nband);
raster **ST_Band**(raster rast, text nbands, character delimiter=,);

Description

Returns one or more bands of an existing raster as a new raster. Useful for building new rasters from existing rasters or export of only selected bands of a raster or rearranging the order of bands in a raster. If no band is specified or any of specified bands does not exist in the raster, then all bands are returned. Used as a helper function in various functions such as for deleting a band.



Warning

For the `nbands` as text variant of function, the default delimiter is `,` which means you can ask for `'1,2,3'` and if you wanted to use a different delimiter you would do `ST_Band(rast, '1@2@3', '@')`. For asking for multiple bands, we strongly suggest you use the array form of this function e.g. `ST_Band(rast, '{1,2,3}'::int[]);` since the `text` list of bands form may be removed in future versions of PostGIS.

Availability: 2.0.0

Examples

This example creates two rasters: one containing band 1 of dummy_rast, and one containing band 2 reclassified as 2BUI.

Code

```
SELECT ST_NumBands(rast1) As numb1, ST_BandPixelType(rast1) As pix1,
 ST_NumBands(rast2) As numb2, ST_BandPixelType(rast2) As pix2
FROM (
    SELECT ST_Band(rast) As rast1, ST_Reclass(ST_Band(rast, 3), '100-200):1, [200-254:2', '2 ↩
    BUI') As rast2
        FROM dummy_rast
        WHERE rid = 2) As foo;
```

Output

```
 numb1 | pix1 | numb2 | pix2
-------+------+-------+------
     1 | 8BUI |     1 | 2BUI
```

Return bands 2 and 3 using array cast syntax.

Code

```
SELECT ST_NumBands(ST_Band(rast, '{2,3}'::int[])) As num_bands
FROM dummy_rast WHERE rid=2;
```

Output

```
2
```

Return bands 2 and 3 using array syntax.

Code

```
SELECT ST_NumBands(ST_Band(rast, ARRAY[2, 3])) As num_bands
    FROM dummy_rast
WHERE rid=2;
```

Output

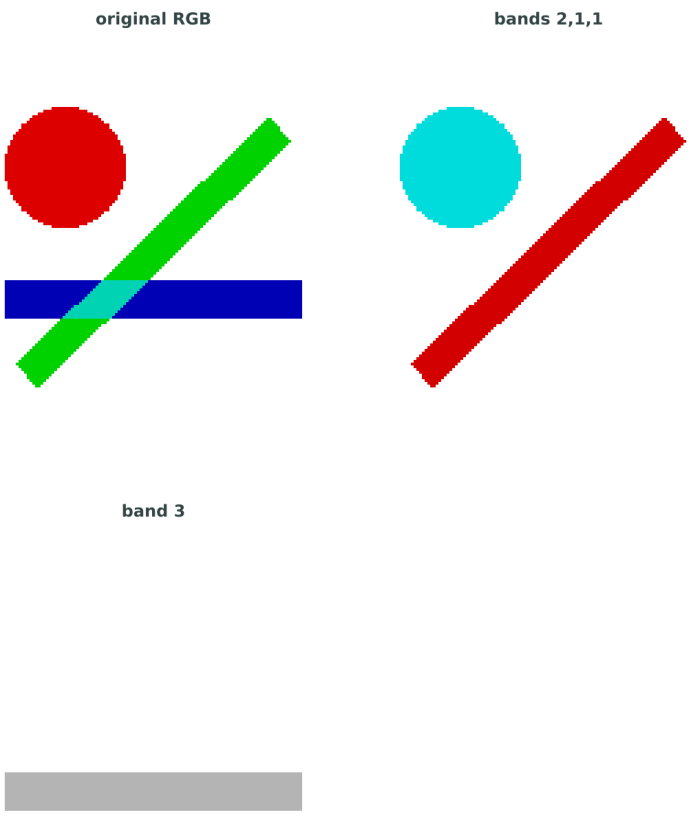
```
2
```

This example uses three distinct shapes as the red, green, and blue bands of a synthetic raster. It compares the original RGB raster with band 2 followed by band 1 twice, and with band 3 alone.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_AddBand(
            ST_AddBand(
                ST_MakeEmptyRaster(128, 128, 0, 128, 1, -1, 0, 0, 0),
                '8BUI'::text, 0::double precision, 0::double precision
            ),
            '8BUI'::text, 0::double precision, 0::double precision
        ),
        '8BUI'::text, 0::double precision, 0::double precision
    ) AS rast
), bands AS (
    SELECT ST_SetValue(
        ST_SetValue(
            ST_SetValue(rast, 1, ST_Buffer('POINT(32 96)'::geometry, 22), 220),
            2, ST_Buffer('LINESTRING(18 20,110 110)'::geometry, 6,
                         'endcap=flat join=bevel'), 210
        ),
        3, ST_Buffer('LINESTRING(10 48,118 48)'::geometry, 7,
                     'endcap=flat'), 180
    ) AS rast
    FROM canvas
)
SELECT ST_AsPNG(rast) AS "original RGB",
    ST_AsPNG(ST_Band(rast, ARRAY[2, 1, 1])) AS "bands 2,1,1",
    ST_AsPNG(ST_Band(rast, 3)) AS "band 3"
FROM bands;
```

Figure



Band selection from a multi-band raster.



original (column rast)



dupe_band



sing_band

See Also

ST_AddBand, ST_NumBands, ST_Reclass, Chapter 11

11.3.5 ST_MakeEmptyCoverage

ST_MakeEmptyCoverage — Cover georeferenced area with a grid of empty raster tiles.

Synopsis

raster **ST_MakeEmptyCoverage**(integer tilewidth, integer tileheight, integer width, integer height, double precision upperleftx, double precision upperlefty, double precision scalex, double precision scaley, double precision skewx, double precision skewy, integer srid=unknown);

Description

Create a set of raster tiles with ST_MakeEmptyRaster. Grid dimension is `width` & `height`. Tile dimension is `tilewidth` & `tileheight`. The covered georeferenced area is from upper left corner (`upperleftx`, `upperlefty`) to lower right corner (`upperleftx` + `width` * `scalex`, `upperlefty` + `height` * `scaley`).



Note
Note that scaley is generally negative for rasters and scalex is generally positive. So lower right corner will have a lower y value and higher x value than the upper left corner.

Availability: 2.4.0

Examples Basic

Create 16 tiles in a 4x4 grid to cover the WGS84 area from upper left corner (22, 77) to lower right corner (55, 33).

Code

```
SELECT (ST_MetaData(tile)).* FROM ST_MakeEmptyCoverage(1, 1, 4, 4, 22, 33, (55 - 22)/(4):: ↩
    float, (33 - 77)/(4)::float, 0., 0., 4326) tile;
```

Output

```
 upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands
------------------------------------------------------------------------------------------
         22 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
         22 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
         22 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
         22 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
```

See Also

ST_MakeEmptyRaster

11.3.6 ST_MakeEmptyRaster

ST_MakeEmptyRaster — Returns an empty raster (having no bands) of given dimensions (width & height), upperleft X and Y, pixel size and rotation (scalex, scaley, skewx & skewy) and reference system (srid). If a raster is passed in, returns a new raster with the same size, alignment and SRID. If srid is left out, the spatial ref is set to unknown (0).

Synopsis

raster **ST_MakeEmptyRaster**(raster rast);
raster **ST_MakeEmptyRaster**(integer width, integer height, float8 upperleftx, float8 upperlefty, float8 scalex, float8 scaley, float8 skewx, float8 skewy, integer srid=unknown);
raster **ST_MakeEmptyRaster**(integer width, integer height, float8 upperleftx, float8 upperlefty, float8 pixelsize);

Description

Returns an empty raster (having no band) of given dimensions (width & height) and georeferenced in spatial (or world) coordinates with upper left X (upperleftx), upper left Y (upperlefty), pixel size and rotation (scalex, scaley, skewx & skewy) and reference system (srid).

The last version use a single parameter to specify the pixel size (pixelsize). scalex is set to this argument and scaley is set to the negative value of this argument. skewx and skewy are set to 0.

If an existing raster is passed in, it returns a new raster with the same meta data settings (without the bands).

If no srid is specified it defaults to 0. After you create an empty raster you probably want to add bands to it and maybe edit it. Refer to ST_AddBand to define bands and ST_SetValue to set initial pixel values.

Examples

Use an existing raster as a template and output metadata of the rasters we added.

Code

```
INSERT INTO dummy_rast(rid, rast)
VALUES(3, ST_MakeEmptyRaster(100, 100, 0.0005, 0.0005, 1, 1, 0, 0, 4326) );
INSERT INTO dummy_rast(rid, rast)
SELECT 4, ST_MakeEmptyRaster(rast)
FROM dummy_rast WHERE rid = 3;
SELECT
    rid,
    (md).width,
    (md).height,
    (md).scalex,
    (md).scaley,
    (md).srid
FROM (
    SELECT rid, ST_MetaData(rast) AS md
    FROM dummy_rast
    WHERE rid IN (3, 4)
) AS foo
ORDER BY rid;
```

Output

```
 rid | width | height | scalex | scaley | srid
-----+-------+--------+--------+--------+------
   3 |   100 |    100 |      1 |      1 | 4326
   4 |   100 |    100 |      1 |      1 | 4326
```

See Also

ST_AddBand, ST_MetaData, ST_ScaleX, ST_ScaleY, ST_SetValue, ST_SkewX, , ST_SkewY

11.3.7 ST_Tile

ST_Tile — Returns a set of rasters resulting from the split of the input raster based upon the desired dimensions of the output rasters.

Synopsis

setof raster **ST_Tile**(raster rast, int[] nband, integer width, integer height, boolean padwithnodata=FALSE, double precision nodataval=NULL);
setof raster **ST_Tile**(raster rast, integer nband, integer width, integer height, boolean padwithnodata=FALSE, double precision nodataval=NULL);
setof raster **ST_Tile**(raster rast, integer width, integer height, boolean padwithnodata=FALSE, double precision nodataval=NULL);

Description

Returns a set of rasters resulting from the split of the input raster based upon the desired dimensions of the output rasters.

If `padwithnodata` = FALSE, edge tiles on the right and bottom sides of the raster may have different dimensions than the rest of the tiles. If `padwithnodata` = TRUE, all tiles will have the same dimensions with the possibility that edge tiles being padded with NODATA values. If raster band(s) do not have NODATA value(s) specified, one can be specified by setting `nodataval`.



Note
If a specified band of the input raster is out-of-db, the corresponding band in the output rasters will also be out-of-db.

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    1, 0), 2, '8BUI', 10, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    2, 0), 2, '8BUI', 20, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    3, 0), 2, '8BUI', 30, 0) AS rast UNION ALL

    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -3, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     4, 0), 2, '8BUI', 40, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -3, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     5, 0), 2, '8BUI', 50, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -3, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     6, 0), 2, '8BUI', 60, 0) AS rast UNION ALL

    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -6, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     7, 0), 2, '8BUI', 70, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -6, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     8, 0), 2, '8BUI', 80, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -6, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     9, 0), 2, '8BUI', 90, 0) AS rast
), bar AS (
    SELECT ST_Union(rast) AS rast FROM foo
```

```
), baz AS (
    SELECT
        row_number() OVER (
            ORDER BY ST_UpperLeftY(rast) DESC, ST_UpperLeftX(rast)
        ) AS tile_id,
        rast
    FROM (
        SELECT ST_Tile(rast, 3, 3, TRUE) AS rast
        FROM bar
    ) AS tiled
)
SELECT
    tile_id,
    ST_UpperLeftX(rast) AS ulx,
    ST_UpperLeftY(rast) AS uly,
    ST_Value(rast, 1, 1, 1) AS band1,
    ST_Value(rast, 2, 1, 1) AS band2
FROM baz;
```

Output

```
 tile_id | ulx | uly | band1 | band2
---------+-----+-----+-------+-------
       1 |   0 |   0 |     1 |    10
       2 |   3 |   0 |     2 |    20
       3 |   6 |   0 |     3 |    30
       4 |   0 |  -3 |     4 |    40
       5 |   3 |  -3 |     5 |    50
       6 |   6 |  -3 |     6 |    60
       7 |   0 |  -6 |     7 |    70
       8 |   3 |  -6 |     8 |    80
       9 |   6 |  -6 |     9 |    90
(9 rows)
```

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    1, 0), 2, '8BUI', 10, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    2, 0), 2, '8BUI', 20, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    3, 0), 2, '8BUI', 30, 0) AS rast UNION ALL

    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -3, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     4, 0), 2, '8BUI', 40, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -3, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     5, 0), 2, '8BUI', 50, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -3, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     6, 0), 2, '8BUI', 60, 0) AS rast UNION ALL

    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -6, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     7, 0), 2, '8BUI', 70, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -6, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     8, 0), 2, '8BUI', 80, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -6, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     9, 0), 2, '8BUI', 90, 0) AS rast
), bar AS (
    SELECT ST_Union(rast) AS rast FROM foo
), baz AS (
    SELECT
        row_number() OVER (
            ORDER BY ST_UpperLeftY(rast) DESC, ST_UpperLeftX(rast)
        ) AS tile_id,
        rast
```

```
    FROM (
        SELECT ST_Tile(rast, ARRAY[2], 3, 3) AS rast
        FROM bar
    ) AS tiled
)
SELECT
    tile_id,
    ST_UpperLeftX(rast) AS ulx,
    ST_UpperLeftY(rast) AS uly,
    ST_Value(rast, 1, 1, 1) AS band_value
FROM baz;
```

Output

```
 tile_id | ulx | uly | band_value
---------+-----+-----+------------
       1 |   0 |   0 |         10
       2 |   3 |   0 |         20
       3 |   6 |   0 |         30
       4 |   0 |  -3 |         40
       5 |   3 |  -3 |         50
       6 |   6 |  -3 |         60
       7 |   0 |  -6 |         70
       8 |   3 |  -6 |         80
       9 |   6 |  -6 |         90
(9 rows)
```

See Also

ST_Union, ST_Retile

11.3.8 ST_Retile

ST_Retile — Return a set of configured tiles from an arbitrarily tiled raster coverage.

Synopsis

SETOF raster **ST_Retile**(regclass tab, name col, geometry ext, float8 sfx, float8 sfy, int tw, int th, text algo='NearestNeighbor');

Description

Return a set of tiles having the specified scale (`sfx`, `sfy`) and max size (`tw`, `th`) and covering the specified extent (`ext`) with data coming from the specified raster coverage (`tab`, `col`).

Algorithm options are: 'NearestNeighbor', 'Bilinear', 'Cubic', 'CubicSpline', and 'Lanczos'. Refer to: GDAL Warp resampling methods for more details.

Availability: 2.2.0

See Also

ST_CreateOverview

11.3.9 ST_FromGDALRaster

ST_FromGDALRaster — Returns a raster from a supported GDAL raster file.

Synopsis

raster **ST_FromGDALRaster**(bytea gdaldata, integer srid=NULL);

Description

Returns a raster from a supported GDAL raster file. `gdaldata` is of type bytea and should be the contents of the GDAL raster file.

If `srid` is NULL, the function will try to automatically assign the SRID from the GDAL raster. If `srid` is provided, the value provided will override any automatically assigned SRID.

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_AsPNG(ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 0.1,  ↩
    -0.1, 0, 0, 4326), 1, '8BUI', 1, 0), 2, '8BUI', 2, 0), 3, '8BUI', 3, 0)) AS png
),
bar AS (
    SELECT 1 AS rid, ST_FromGDALRaster(png) AS rast FROM foo
    UNION ALL
    SELECT 2 AS rid, ST_FromGDALRaster(png, 3310) AS rast FROM foo
)
SELECT
    rid,
    ST_MetaData(rast) AS metadata,
    ST_SummaryStats(rast, 1) AS stats1,
    ST_SummaryStats(rast, 2) AS stats2,
    ST_SummaryStats(rast, 3) AS stats3
FROM bar
ORDER BY rid;
```

Output

```
 rid |          metadata           |    stats1     |    stats2     |     stats3
-----+-----------------------------+---------------+---------------+----------------
   1 | (0,0,2,2,1,-1,0,0,0,3)      | (4,4,1,0,1,1) | (4,8,2,0,2,2) | (4,12,3,0,3,3)
   2 | (0,0,2,2,1,-1,0,0,3310,3)   | (4,4,1,0,1,1) | (4,8,2,0,2,2) | (4,12,3,0,3,3)
(2 rows)
```

See Also

ST_AsGDALRaster

11.4 Raster Accessors

11.4.1 ST_GeoReference

ST_GeoReference — Returns the georeference meta data in GDAL or ESRI format as commonly seen in a world file. Default is GDAL.

Synopsis

text **ST_GeoReference**(raster rast, text format=GDAL);

Description

Returns the georeference meta data including carriage return in GDAL or ESRI format as commonly seen in a world file. Default is GDAL if no type specified. type is string 'GDAL' or 'ESRI'.

Difference between format representations is as follows:

GDAL:

Output

```
scalex
skewy
skewx
scaley
upperleftx
upperlefty
```

ESRI:

Output

```
scalex
skewy
skewx
scaley
upperleftx + scalex*0.5
upperlefty + scaley*0.5
```

Examples

Code

```
SELECT
    format,
    replace(ST_GeoReference(rast, format), E'\n', ' | ') AS georef
FROM dummy_rast
CROSS JOIN (VALUES ('ESRI'), ('GDAL')) AS formats(format)
WHERE rid = 1;
```

Output

```
 format | georef
--------+-------------------------------------------------------------------------
 ESRI   | 2.0000000000 | 0.0000000000 | 0.0000000000 | 3.0000000000 | 1.5000000000 | 2.0000000000
 GDAL   | 2.0000000000 | 0.0000000000 | 0.0000000000 | 3.0000000000 | 0.5000000000 | 0.5000000000
(2 rows)
```

See Also

ST_SetGeoReference, ST_ScaleX, ST_ScaleY

11.4.2 ST_Height

ST_Height — Returns the height of the raster in pixels.

Synopsis

integer **ST_Height**(raster rast);

Description

Returns the height of the raster.

Examples

Code

```
SELECT rid, ST_Height(rast) As rastheight
FROM dummy_rast;
```

Output

```
 rid | rastheight
-----+------------
   1 |         20
   2 |          5
```

See Also

ST_Width

11.4.3 ST_IsEmpty

ST_IsEmpty — Returns true if the raster is empty (width = 0 and height = 0). Otherwise, returns false.

Synopsis

boolean **ST_IsEmpty**(raster rast);

Description

Returns true if the raster is empty (width = 0 and height = 0). Otherwise, returns false.

Availability: 2.0.0

Examples

Code

```
SELECT ST_IsEmpty(ST_MakeEmptyRaster(100, 100, 0, 0, 0, 0, 0, 0))
```

Output

```
f
```

Code

```
SELECT ST_IsEmpty(ST_MakeEmptyRaster(0, 0, 0, 0, 0, 0, 0, 0))
```

Output

```
t
```

See Also

ST_HasNoBand

11.4.4 ST_MemSize

ST_MemSize — Returns the amount of space (in bytes) the raster takes.

Synopsis

integer **ST_MemSize**(raster rast);

Description

Returns the amount of space (in bytes) the raster takes.

This is a nice compliment to PostgreSQL built in functions pg_column_size, pg_size_pretty, pg_relation_size, pg_total_relation_size.



Note
pg_relation_size which gives the byte size of a table may return byte size lower than ST_MemSize. This is because pg_relation_size does not add toasted table contribution and large geometries are stored in TOAST tables. pg_column_size might return lower because it returns the compressed size.
pg_total_relation_size - includes, the table, the toasted tables, and the indexes.

Availability: 2.2.0

Examples

Code

```
SELECT ST_MemSize(ST_AsRaster(ST_Buffer(ST_Point(1, 5), 10, 1000), 150, 150, '8BUI')) AS ↩
    rast_mem;
```

Output

```
22568
```

See Also

11.4.5 ST_MetaData

ST_MetaData — Returns basic meta data about a raster object such as pixel size, rotation (skew), upper, lower left, etc.

Synopsis

record **ST_MetaData**(raster rast);

Description

Returns basic meta data about a raster object such as pixel size, rotation (skew), upper, lower left, etc. Columns returned: upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands

Examples

Code

```
SELECT
    rid,
    format('(%s, %s)', (md).upperleftx, (md).upperlefty) AS upper_left,
    format('%sx%s', (md).width, (md).height) AS size,
    format('(%s, %s)', (md).scalex, (md).scaley) AS scale,
    format('(%s, %s)', (md).skewx, (md).skewy) AS skew,
    (md).srid AS srid,
    (md).numbands AS numbands
FROM (
    SELECT rid, ST_MetaData(rast) AS md
    FROM dummy_rast
) AS foo
ORDER BY rid;
```

Output

```
 rid |        upper_left         | size  |      scale      |  skew   | srid | numbands
-----+---------------------------+-------+-----------------+---------+------+----------
   1 | (0.5, 0.5)                | 10x20 | (2, 3)          | (0, 0) |    0 |         0
   2 | (3427927.75, 5793244)     | 5x5   | (0.05, -0.05)   | (0, 0) |    0 |         3
(2 rows)
```

See Also

ST_BandMetaData, ST_NumBands

11.4.6 ST_NumBands

ST_NumBands — Returns the number of bands in the raster object.

Synopsis

integer **ST_NumBands**(raster rast);

Description

Returns the number of bands in the raster object.

Examples

Code

```
SELECT rid, ST_NumBands(rast) As numbands
FROM dummy_rast;
```

Output

```
rid | numbands
----+----------
  1 |        0
  2 |        3
```

See Also

ST_Value

11.4.7 ST_PixelHeight

ST_PixelHeight — Returns the pixel height in geometric units of the spatial reference system.

Synopsis

double precision **ST_PixelHeight**(raster rast);

Description

Returns the height of a pixel in geometric units of the spatial reference system. In the common case where there is no skew, the pixel height is just the scale ratio between geometric coordinates and raster pixels.

Refer to ST_PixelWidth for a diagrammatic visualization of the relationship.

Examples

Rasters with no skew.

Code

```
SELECT ST_Height(rast) As rastheight, ST_PixelHeight(rast) As pixheight,
 ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
        ST_SkewY(rast) As skewy
FROM dummy_rast;
```

Output

```
 rastheight | pixheight | scalex | scaley | skewx | skewy
------------+-----------+--------+--------+-------+----------
         20 |         3 |      2 |      3 |     0 |        0
          5 |      0.05 |   0.05 |  -0.05 |     0 |        0
```

Rasters with skew different than 0.

Code

```
SELECT ST_Height(rast) As rastheight, ST_PixelHeight(rast) As pixheight,
 ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
        ST_SkewY(rast) As skewy
FROM (SELECT ST_SetSkew(rast, 0.5, 0.5) As rast
        FROM dummy_rast) As skewed;
```

Output

```
rastheight |     pixheight     | scalex | scaley | skewx | skewy
-----------+-------------------+--------+--------+-------+-------
        20 |  3.04138126514911 |      2 |      3 |   0.5 |   0.5
         5 | 0.502493781056044 |   0.05 |  -0.05 |   0.5 |   0.5
```

See Also

ST_PixelWidth, ST_ScaleX, ST_ScaleY, ST_SkewX, ST_SkewY

11.4.8 ST_PixelWidth

ST_PixelWidth — Returns the pixel width in geometric units of the spatial reference system.

Synopsis

double precision **ST_PixelWidth**(raster rast);

Description

Returns the width of a pixel in geometric units of the spatial reference system. In the common case where there is no skew, the pixel width is just the scale ratio between geometric coordinates and raster pixels.

The following executable example draws the two pixel basis vectors on the same coordinate frame as a skewed raster footprint. The `i direction` vector is `(scaleX, skewY)`; the `j direction` vector is `(skewX, scaleY)`.

Examples

Pixel basis vectors and footprint for a raster with non-zero skew.

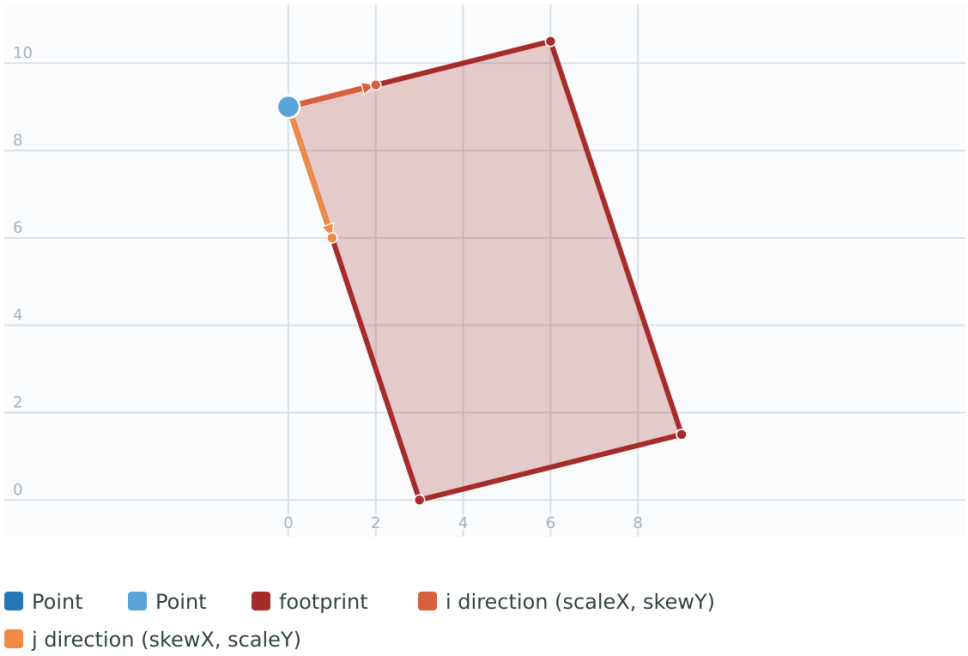
Code

```
WITH raster AS (
    SELECT ST_MakeEmptyRaster(
        3, 3, 0, 0,
        2, -3, 1, 0.5,
        0
    ) AS rast
)
SELECT
    ST_ConvexHull(rast) AS footprint,
    ST_MakeLine(
        ST_Point(0, 0),
        ST_Point(ST_ScaleX(rast), ST_SkewY(rast))
    ) AS "i direction (scaleX, skewY)",
    ST_MakeLine(
        ST_Point(0, 0),
        ST_Point(ST_SkewX(rast), ST_ScaleY(rast))
    ) AS "j direction (skewX, scaleY)"
FROM raster;
```

Output

```
POLYGON((0 0,6 1.5,9 -7.5,3 -9,0 0)) | LINESTRING(0 0,2 0.5) | LINESTRING(0 0,1 -3)
```

Figure



Rasters with no skew.

Code

```
SELECT ST_Width(rast) As rastwidth, ST_PixelWidth(rast) As pixwidth,
ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
ST_SkewY(rast) As skewy
FROM dummy_rast;
```

Output

```
    rastwidth | pixwidth | scalex | scaley | skewx | skewy
-----------+----------+--------+--------+-------+----------
10 |        2 |      2 |      3 |     0 |        0
 5 |     0.05 |   0.05 |  -0.05 |     0 |        0
```

Rasters with skew different than 0.

Code

```
SELECT ST_Width(rast) As rastwidth, ST_PixelWidth(rast) As pixwidth,
ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
ST_SkewY(rast) As skewy
FROM (SELECT ST_SetSkew(rast, 0.5, 0.5) As rast
FROM dummy_rast) As skewed;
```

Output

```
    rastwidth |     pixwidth      | scalex | scaley | skewx | skewy
-----------+-------------------+--------+--------+-------+----------
10 |  2.06155281280883 |      2 |      3 |   0.5 |      0.5
 5 | 0.502493781056044 |   0.05 |  -0.05 |   0.5 |      0.5
```

See Also

ST_PixelHeight, ST_ScaleX, ST_ScaleY, ST_SkewX, ST_SkewY

11.4.9 ST_ScaleX

ST_ScaleX — Returns the X component of the pixel width in units of coordinate reference system.

Synopsis

float8 **ST_ScaleX**(raster rast);

Description

Returns the X component of the pixel width in units of coordinate reference system. Refer to World File for more details.

Changed: 2.0.0. In WKTRaster versions this was called ST_PixelSizeX.

Examples

Code

```
SELECT rid, ST_ScaleX(rast) As rastpixwidth
FROM dummy_rast;
```

Output

```
 rid | rastpixwidth
-----+--------------
   1 |            2
   2 |         0.05
```

See Also

ST_Width

11.4.10 ST_ScaleY

ST_ScaleY — Returns the Y component of the pixel height in units of coordinate reference system.

Synopsis

float8 **ST_ScaleY**(raster rast);

Description

Returns the Y component of the pixel height in units of coordinate reference system. May be negative. Refer to World File for more details.

Changed: 2.0.0. In WKTRaster versions this was called ST_PixelSizeY.

Examples

Code

```
SELECT rid, ST_ScaleY(rast) As rastpixheight
FROM dummy_rast;
```

Output

```
 rid | rastpixheight
-----+--------------
   1 |             3
   2 |         -0.05
```

See Also

ST_Height

11.4.11 ST_RasterToWorldCoord

ST_RasterToWorldCoord — Returns the raster's upper left corner as geometric X and Y (longitude and latitude) given a column and row. Column and row starts at 1.

Synopsis

record **ST_RasterToWorldCoord**(raster rast, integer xcolumn, integer yrow);

Description

Returns the upper left corner as geometric X and Y (longitude and latitude) given a column and row. Returned X and Y are in geometric units of the georeferenced raster. Numbering of column and row starts at 1 but if either parameter is passed a zero, a negative number or a number greater than the respective dimension of the raster, it will return coordinates outside of the raster assuming the raster's grid is applicable outside the raster's bounds.

Availability: 2.1.0

Examples

This example uses a non-skewed raster.

Code

```
SELECT
    rid,
    (ST_RasterToWorldCoord(rast, 1, 1)).*,
    (ST_RasterToWorldCoord(rast, 2, 2)).*
FROM dummy_rast
```

Output

```
 rid | longitude  | latitude | longitude |  latitude
-----+------------+----------+-----------+------------
   1 |        0.5 |      0.5 |       2.5 |         3.5
   2 | 3427927.75 |  5793244 | 3427927.8 | 5793243.95
```

This example uses a skewed raster.

Code

```
SELECT
    rid,
    (ST_RasterToWorldCoord(rast, 1, 1)).*,
    (ST_RasterToWorldCoord(rast, 2, 3)).*
FROM (
    SELECT
        rid,
        ST_SetSkew(rast, 100.5, 0) As rast
    FROM dummy_rast
) As foo
```

Output

```
 rid | longitude  | latitude | longitude | latitude
-----+------------+----------+-----------+----------
   1 |        0.5 |      0.5 |     203.5 |       6.5
   2 | 3427927.75 |  5793244 | 3428128.8 | 5793243.9
```

See Also

ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SetSkew

11.4.12 ST_RasterToWorldCoordX

ST_RasterToWorldCoordX — Returns the geometric X coordinate upper left of a raster, column and row. Numbering of columns and rows starts at 1.

Synopsis

float8 **ST_RasterToWorldCoordX**(raster rast, integer xcolumn);
float8 **ST_RasterToWorldCoordX**(raster rast, integer xcolumn, integer yrow);

Description

Returns the upper left X coordinate of a raster column row in geometric units of the georeferenced raster. Numbering of columns and rows starts at 1 but if you pass in a negative number or number higher than number of columns in raster, it will give you coordinates outside of the raster file to left or right with the assumption that the skew and pixel sizes are same as selected raster.



Note
For non-skewed rasters, providing the X column is sufficient. For skewed rasters, the georeferenced coordinate is a function of the ST_ScaleX and ST_SkewX and row and column. An error will be raised if you give just the X column for a skewed raster.

Changed: 2.1.0 In prior versions, this was called ST_Raster2WorldCoordX

Examples

This example shows that providing a column is sufficient for a non-skewed raster.

Code

```
SELECT rid, ST_RasterToWorldCoordX(rast, 1) As x1coord,
    ST_RasterToWorldCoordX(rast, 2) As x2coord,
    ST_ScaleX(rast) As pixelx
FROM dummy_rast;
```

Output

```
 rid |  x1coord   |  x2coord  | pixelx
-----+------------+-----------+--------
   1 |        0.5 |       2.5 |      2
   2 | 3427927.75 | 3427927.8 |   0.05
```

For fun let's skew it.

Code

```
SELECT rid, ST_RasterToWorldCoordX(rast, 1, 1) As x1coord,
    ST_RasterToWorldCoordX(rast, 2, 3) As x2coord,
    ST_ScaleX(rast) As pixelx
FROM (SELECT rid, ST_SetSkew(rast, 100.5, 0) As rast FROM dummy_rast) As foo;
```

Output

```
 rid |  x1coord   |  x2coord  | pixelx
-----+------------+-----------+--------
   1 |        0.5 |     203.5 |      2
   2 | 3427927.75 | 3428128.8 |   0.05
```

See Also

ST_ScaleX, ST_RasterToWorldCoordY, ST_SetSkew, ST_SkewX

11.4.13 ST_RasterToWorldCoordY

ST_RasterToWorldCoordY — Returns the geometric Y coordinate upper left corner of a raster, column and row. Numbering of columns and rows starts at 1.

Synopsis

float8 **ST_RasterToWorldCoordY**(raster rast, integer yrow);
float8 **ST_RasterToWorldCoordY**(raster rast, integer xcolumn, integer yrow);

Description

Returns the upper left Y coordinate of a raster column row in geometric units of the georeferenced raster. Numbering of columns and rows starts at 1 but if you pass in a negative number or number higher than number of columns/rows in raster, it will give you coordinates outside of the raster file to left or right with the assumption that the skew and pixel sizes are same as selected raster tile.



Note
For non-skewed rasters, providing the Y column is sufficient. For skewed rasters, the georeferenced coordinate is a function of the ST_ScaleY and ST_SkewY and row and column. An error will be raised if you give just the Y row for a skewed raster.

Changed: 2.1.0 In prior versions, this was called ST_Raster2WorldCoordY

Examples

Non-skewed raster providing row is sufficient.

Code

```
SELECT rid, ST_RasterToWorldCoordY(rast, 1) As y1coord,
    ST_RasterToWorldCoordY(rast, 3) As y2coord,
    ST_ScaleY(rast) As pixely
FROM dummy_rast;
```

Output

```
 rid | y1coord |  y2coord  | pixely
-----+---------+-----------+--------
   1 |     0.5 |       6.5 |      3
   2 | 5793244 | 5793243.9 |  -0.05
```

For fun let's skew it.

Code

```
SELECT rid, ST_RasterToWorldCoordY(rast, 1, 1) As y1coord,
    ST_RasterToWorldCoordY(rast, 2, 3) As y2coord,
    ST_ScaleY(rast) As pixely
FROM (SELECT rid, ST_SetSkew(rast, 0, 100.5) As rast FROM dummy_rast) As foo;
```

Output

```
 rid | y1coord |  y2coord  | pixely
-----+---------+-----------+--------
   1 |     0.5 |       107 |      3
   2 | 5793244 | 5793344.4 |  -0.05
```

See Also

ST_ScaleY, ST_RasterToWorldCoordX, ST_SetSkew, ST_SkewY

11.4.14 ST_Rotation

ST_Rotation — Returns the rotation of the raster in radian.

Synopsis

float8 **ST_Rotation**(raster rast);

Description

Returns the uniform rotation of the raster in radian. If a raster does not have uniform rotation, NaN is returned. Refer to World File for more details.

Examples

Code

```
SELECT rid, ST_Rotation(ST_SetScale(ST_SetSkew(rast, sqrt(2)), sqrt(2))) AS rot FROM  ↩
    dummy_rast;
```

Output

```
 rid |        rot
-----+-------------------
   1 | 0.785398163397448
   2 | 0.785398163397448
```

See Also

ST_SetRotation, ST_SetScale, ST_SetSkew

11.4.15 ST_SkewX

ST_SkewX — Returns the georeference X skew (or rotation parameter).

Synopsis

float8 **ST_SkewX**(raster rast);

Description

Returns the georeference X skew (or rotation parameter). Refer to World File for more details.

Examples

Code

```
WITH refs AS (
    SELECT
        rid,
        format('(%s, %s)', ST_SkewX(rast), ST_SkewY(rast)) AS skew,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
)
SELECT
    rid,
    skew,
    term,
    georef[idx] AS value
FROM refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
) AS parts(idx, term)
ORDER BY rid, idx;
```

Output

```
 rid |  skew  |  term   |       value
-----+--------+---------+--------------------
   1 | (0, 0) | scaleX  | 2.0000000000
   1 | (0, 0) | skewY   | 0.0000000000
   1 | (0, 0) | skewX   | 0.0000000000
   1 | (0, 0) | scaleY  | 3.0000000000
   1 | (0, 0) | originX | 0.5000000000
   1 | (0, 0) | originY | 0.5000000000
   2 | (0, 0) | scaleX  | 0.0500000000
   2 | (0, 0) | skewY   | 0.0000000000
   2 | (0, 0) | skewX   | 0.0000000000
   2 | (0, 0) | scaleY  | -0.0500000000
   2 | (0, 0) | originX | 3427927.7500000000
   2 | (0, 0) | originY | 5793244.0000000000
(12 rows)
```

See Also

ST_GeoReference, ST_SkewY, ST_SetSkew

11.4.16 ST_SkewY

ST_SkewY — Returns the georeference Y skew (or rotation parameter).

Synopsis

float8 **ST_SkewY**(raster rast);

Description

Returns the georeference Y skew (or rotation parameter). Refer to World File for more details.

Examples

Code

```
WITH refs AS (
    SELECT
        rid,
        format('(%s, %s)', ST_SkewX(rast), ST_SkewY(rast)) AS skew,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
)
SELECT
    rid,
    skew,
    term,
    georef[idx] AS value
FROM refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
) AS parts(idx, term)
ORDER BY rid, idx;
```

Output

```
 rid |  skew  |  term   |       value
-----+--------+---------+--------------------
   1 | (0, 0) | scaleX  | 2.0000000000
   1 | (0, 0) | skewY   | 0.0000000000
   1 | (0, 0) | skewX   | 0.0000000000
   1 | (0, 0) | scaleY  | 3.0000000000
   1 | (0, 0) | originX | 0.5000000000
   1 | (0, 0) | originY | 0.5000000000
   2 | (0, 0) | scaleX  | 0.0500000000
   2 | (0, 0) | skewY   | 0.0000000000
   2 | (0, 0) | skewX   | 0.0000000000
   2 | (0, 0) | scaleY  | -0.0500000000
   2 | (0, 0) | originX | 3427927.7500000000
   2 | (0, 0) | originY | 5793244.0000000000
(12 rows)
```

See Also

ST_GeoReference, ST_SkewX, ST_SetSkew

11.4.17 ST_SRID

ST_SRID — Returns the spatial reference identifier of the raster as defined in spatial_ref_sys table.

Synopsis

integer **ST_SRID**(raster rast);

Description

Returns the spatial reference identifier of the raster object as defined in the spatial_ref_sys table.



Note
Non-georeferenced rasters and geometries use SRID 0.

Examples

Code

```
SELECT ST_SRID(rast) As srid
FROM dummy_rast WHERE rid=1;
```

Output

```
0
```

See Also

Section 4.5, ST_SRID

11.4.18 ST_Summary

ST_Summary — Returns a text summary of the contents of the raster.

Synopsis

text **ST_Summary**(raster rast);

Description

Returns a text summary of the contents of the raster.

Availability: 2.1.0

Examples

Code

```
SELECT ST_Summary(ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, -1, ↩
    0, 0, 0),
                1, '8BUI', 1, 0
            ),
            2, '32BF', 0, -9999
        ),
        3, '16BSI', 0, NULL
    )
);
```

Output

```
 Raster of 10x10 pixels has 3 bands and extent of BOX(0 -10,10 0)+
     band 1 of pixtype 8BUI is in-db with NODATA value of 0      +
     band 2 of pixtype 32BF is in-db with NODATA value of -9999  +
     band 3 of pixtype 16BSI is in-db with no NODATA value
(1 row)
```

See Also

ST_MetaData, ST_BandMetaData, ST_Summary ST_Extent

11.4.19 ST_UpperLeftX

ST_UpperLeftX — Returns the upper left X coordinate of raster in projected spatial ref.

Synopsis

float8 **ST_UpperLeftX**(raster rast);

Description

Returns the upper left X coordinate of raster in projected spatial ref.

Examples

Code

```
SELECt rid, ST_UpperLeftX(rast) As ulx
FROM dummy_rast;
```

Output

```
 rid |    ulx
-----+-----------
   1 |        0.5
   2 | 3427927.75
```

See Also

ST_UpperLeftY, ST_GeoReference, Box3D

11.4.20 ST_UpperLeftY

ST_UpperLeftY — Returns the upper left Y coordinate of raster in projected spatial ref.

Synopsis

float8 **ST_UpperLeftY**(raster rast);

Description

Returns the upper left Y coordinate of raster in projected spatial ref.

Examples

Code

```
SELECT rid, ST_UpperLeftY(rast) As uly
FROM dummy_rast;
```

Output

```
 rid |   uly
-----+---------
   1 |     0.5
   2 | 5793244
```

See Also

ST_UpperLeftX, ST_GeoReference, Box3D

11.4.21 ST_Width

ST_Width — Returns the width of the raster in pixels.

Synopsis

integer **ST_Width**(raster rast);

Description

Returns the width of the raster in pixels.

Examples

Code

```
SELECT ST_Width(rast) As rastwidth
FROM dummy_rast WHERE rid=1;
```

Output

```
10
```

See Also

ST_Height

11.4.22 ST_WorldToRasterCoord

ST_WorldToRasterCoord — Returns the upper left corner as column and row given geometric X and Y (longitude and latitude) or a point geometry expressed in the spatial reference coordinate system of the raster.

Synopsis

record **ST_WorldToRasterCoord**(raster rast, geometry pt);
record **ST_WorldToRasterCoord**(raster rast, double precision longitude, double precision latitude);

Description

Returns the upper left corner as column and row given geometric X and Y (longitude and latitude) or a point geometry. This function works regardless of whether or not the geometric X and Y or point geometry is outside the extent of the raster. Geometric X and Y must be expressed in the spatial reference coordinate system of the raster.

Availability: 2.1.0

Examples

Code

```
SELECT
    rid,
    (ST_WorldToRasterCoord(rast, 3427927.8, 20.5)).*,
    (ST_WorldToRasterCoord(rast, ST_GeomFromText('POINT(3427927.8 20.5)', ST_SRID(rast)))).*
FROM dummy_rast;
```

Output

```
 rid | columnx |   rowy    | columnx |   rowy
-----+---------+-----------+---------+-----------
   1 | 1713964 |         7 | 1713964 |         7
   2 |       2 | 115864471 |       2 | 115864471
```

See Also

ST_WorldToRasterCoordX, ST_WorldToRasterCoordY, ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SRID

11.4.23 ST_WorldToRasterCoordX

ST_WorldToRasterCoordX — Returns the column in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw) represented in world spatial reference system of raster.

Synopsis

integer **ST_WorldToRasterCoordX**(raster rast, geometry pt);
integer **ST_WorldToRasterCoordX**(raster rast, double precision xw);
integer **ST_WorldToRasterCoordX**(raster rast, double precision xw, double precision yw);

Description

Returns the column in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw). A point, or (both xw and yw world coordinates are required if a raster is skewed). If a raster is not skewed then xw is sufficient. World coordinates are in the spatial reference coordinate system of the raster.

Changed: 2.1.0 In prior versions, this was called ST_World2RasterCoordX

Examples

Code

```
SELECT rid, ST_WorldToRasterCoordX(rast, 3427927.8) As xcoord,
        ST_WorldToRasterCoordX(rast, 3427927.8, 20.5) As xcoord_xwyw,
        ST_WorldToRasterCoordX(rast, ST_GeomFromText('POINT(3427927.8 20.5)', ST_SRID(rast)) ↩
    ) As ptxcoord
FROM dummy_rast;
```

Output

```
 rid | xcoord  |  xcoord_xwyw   | ptxcoord
-----+---------+---------+----------
   1 | 1713964 | 1713964 |  1713964
   2 |       1 |       1 |        1
```

See Also

ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SRID

11.4.24 ST_WorldToRasterCoordY

ST_WorldToRasterCoordY — Returns the row in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw) represented in world spatial reference system of raster.

Synopsis

integer **ST_WorldToRasterCoordY**(raster rast, geometry pt);
integer **ST_WorldToRasterCoordY**(raster rast, double precision xw);
integer **ST_WorldToRasterCoordY**(raster rast, double precision xw, double precision yw);

Description

Returns the row in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw). A point, or (both xw and yw world coordinates are required if a raster is skewed). If a raster is not skewed then xw is sufficient. World coordinates are in the spatial reference coordinate system of the raster.

Changed: 2.1.0 In prior versions, this was called ST_World2RasterCoordY

Examples

Code

```
SELECT rid, ST_WorldToRasterCoordY(rast, 20.5) As ycoord,
        ST_WorldToRasterCoordY(rast, 3427927.8, 20.5) As ycoord_xwyw,
        ST_WorldToRasterCoordY(rast, ST_GeomFromText('POINT(3427927.8 20.5)', ST_SRID(rast)) ↩
    ) As ptycoord
FROM dummy_rast;
```

Output

```
 rid |  ycoord   | ycoord_xwyw | ptycoord
-----+-----------+-------------+-----------
   1 |         7 |           7 |         7
   2 | 115864471 |   115864471 | 115864471
```

See Also

ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SRID

11.5 Raster Band Accessors

11.5.1 ST_BandMetaData

ST_BandMetaData — Returns basic meta data for a specific raster band. band num 1 is assumed if none-specified.

Synopsis

(1) record **ST_BandMetaData**(raster rast, integer band=1);
(2) record **ST_BandMetaData**(raster rast, integer[] band);

Description

Returns basic meta data about a raster band. Columns returned: pixeltype, nodatavalue, isoutdb, path, outdbbandnum, filesize, filetimestamp.



Note
If raster contains no bands then an error is thrown.



Note
If band has no NODATA value, nodatavalue are NULL.



Note
If isoutdb is False, path, outdbbandnum, filesize and filetimestamp are NULL. If outdb access is disabled, filesize and filetimestamp will also be NULL.

Enhanced: 2.5.0 to include *outdbbandnum*, *filesize* and *filetimestamp* for outdb rasters.

Examples

Variant 1.

Code

```
SELECT
    rid,
    (foo.md).*
FROM (
    SELECT
        rid,
        ST_BandMetaData(rast, 1) AS md
    FROM dummy_rast
    WHERE rid=2
) As foo;
```

Output

```
 rid | pixeltype | nodatavalue | isoutdb | path | outdbbandnum
-----+-----------+-------------+---------+------+--------------
   2 | 8BUI      |           0 | f       |      |
(1 row)
```

Variant 2.

Code

```
WITH foo AS (
    SELECT
        ST_AddBand(NULL::raster, '/home/pele/devel/geo/postgis-git/raster/test/regress/ ↩
    loader/Projected.tif', NULL::int[]) AS rast
)
SELECT
    bandnum,
    pixeltype,
    isoutdb,
    regexp_replace(path, '^.*/', '') AS file,
    outdbbandnum
FROM ST_BandMetaData((SELECT rast FROM foo),
    ARRAY[1, 3, 2]::int[]
);
```

Output

```
 bandnum | pixeltype | isoutdb |     file      | outdbbandnum
---------+-----------+---------+---------------+--------------
       1 | 8BUI      | t       | Projected.tif |            1
       3 | 8BUI      | t       | Projected.tif |            3
       2 | 8BUI      | t       | Projected.tif |            2
```

See Also

ST_MetaData, ST_BandPixelType

11.5.2 ST_BandNoDataValue

ST_BandNoDataValue — Returns the value in a given band that represents no data. If no band num 1 is assumed.

Synopsis

double precision **ST_BandNoDataValue**(raster rast, integer bandnum=1);

Description

Returns the value that represents no data for the band

Examples

Code

```
SELECT ST_BandNoDataValue(rast, 1) As bnval1,
    ST_BandNoDataValue(rast, 2) As bnval2, ST_BandNoDataValue(rast, 3) As bnval3
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 bnval1 | bnval2 | bnval3
--------+--------+--------
      0 |      0 |      0
```

See Also

ST_NumBands

11.5.3 ST_BandIsNoData

ST_BandIsNoData — Returns true if the band is filled with only nodata values.

Synopsis

boolean **ST_BandIsNoData**(raster rast, integer band, boolean forceChecking=false);
boolean **ST_BandIsNoData**(raster rast, boolean forceChecking=false);

Description

Returns true if the band is filled with only nodata values. Band 1 is assumed if not specified. If the last argument is TRUE, the entire band is checked pixel by pixel. Otherwise, the function simply returns the value of the isnodata flag for the band. The default value for this parameter is FALSE, if not specified.

Availability: 2.0.0



Note
If the flag is dirty (this is, the result is different using TRUE as last parameter and not using it) you should update the raster to set this flag to true, by using ST_SetBandIsNoData(), or ST_SetBandNoDataValue() with TRUE as last argument. See ST_SetBandIsNoData.

Examples

This example creates a dummy table with one raster column. The raster has two bands and one pixel per band. In the first band, the NODATA value and pixel value are both 3. In the second band, the NODATA value is 13 and the pixel value is 4.

Code

```
create table dummy_rast (rid integer, rast raster);

insert into dummy_rast values(
1,
(
'01' -- little endian (uint8 ndr)
||
'0000' -- version (uint16 0)
||
'0200' -- nBands (uint16 0)
||
'17263529ED684A3F' -- scaleX (float64 0.000805965234044584)
||
'F9253529ED684ABF' -- scaleY (float64 -0.00080596523404458)
||
'1C9F33CE69E352C0' -- ipX (float64 -75.5533328537098)
||
'718F0E9A27A44840' -- ipY (float64 49.2824585505576)
||
'ED50EB853EC32B3F' -- skewX (float64 0.000211812383858707)
||
'7550EB853EC32B3F' -- skewY (float64 0.000211812383858704)
||
'E6100000' -- SRID (int32 4326)
||
'0100' -- width (uint16 1)
||
'0100' -- height (uint16 1)
||
'6' -- hasnodatavalue and isnodata value set to true.
||
'2' -- first band type (4BUI)
||
'03' -- novalue==3
||
'03' -- pixel(0,0)==3 (same that nodata)
||
'0' -- hasnodatavalue set to false
||
'5' -- second band type (16BSI)
||
'0D00' -- novalue==13
||
'0400' -- pixel(0,0)==4
)::raster
);

select ST_BandIsNoData(rast, 1) from dummy_rast where rid = 1; -- Expected true
select ST_BandIsNoData(rast, 2) from dummy_rast where rid = 1; -- Expected false
```

See Also

ST_BandNoDataValue, ST_NumBands, ST_SetBandNoDataValue, ST_SetBandIsNoData

11.5.4 ST_BandPath

ST_BandPath — Returns system file path to a band stored in file system. If no bandnum specified, 1 is assumed.

Synopsis

text **ST_BandPath**(raster rast, integer bandnum=1);

Description

Returns system file path to a band. Throws an error if called with an in db band.

Examples

Code

```
WITH foo AS (
    SELECT ST_AddBand(
        NULL::raster,
        '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif',
        NULL::int[]
    ) AS rast
)
SELECT bandnum, ST_BandPath(rast, bandnum) AS path
FROM foo
CROSS JOIN generate_series(1, 3) AS bandnum;
```

Output

```
 bandnum |                                 path
---------+-----------------------------------------------------------------------
       1 | /home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif
       2 | /home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif
       3 | /home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif
```

See Also

ST_BandMetaData, ST_SetBandPath, ST_SetBandIndex

11.5.5 ST_BandFileSize

ST_BandFileSize — Returns the file size of a band stored in file system. If no bandnum specified, 1 is assumed.

Synopsis

bigint **ST_BandFileSize**(raster rast, integer bandnum=1);

Description

Returns the file size of a band stored in file system. Throws an error if called with an in db band, or if outdb access is not enabled.

This function is typically used in conjunction with ST_BandPath() and ST_BandFileTimestamp() so a client can determine if the filename of a outdb raster as seen by it is the same as the one seen by the server.

Availability: 2.5.0

Examples

Code

```
SELECT ST_BandFileSize(rast, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
240574
```

11.5.6 ST_BandFileTimestamp

ST_BandFileTimestamp — Returns the file timestamp of a band stored in file system. If no bandnum specified, 1 is assumed.

Synopsis

bigint **ST_BandFileTimestamp**(raster rast, integer bandnum=1);

Description

Returns the file timestamp (number of seconds since Jan 1st 1970 00:00:00 UTC) of a band stored in file system. Throws an error if called with an in db band, or if outdb access is not enabled.

This function is typically used in conjunction with ST_BandPath() and ST_BandFileSize() so a client can determine if the filename of a outdb raster as seen by it is the same as the one seen by the server.

Availability: 2.5.0

Examples

Code

```
SELECT ST_BandFileTimestamp(rast, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
1521807257
```

11.5.7 ST_BandPixelType

ST_BandPixelType — Returns the type of pixel for given band. If no bandnum specified, 1 is assumed.

Synopsis

text **ST_BandPixelType**(raster rast, integer bandnum=1);

Description

Returns name describing data type and size of values stored in each cell of given band.

There are 11 pixel types. Pixel Types supported are as follows:

- 1BB - 1-bit boolean
- 2BUI - 2-bit unsigned integer

- 4BUI - 4-bit unsigned integer
- 8BSI - 8-bit signed integer
- 8BUI - 8-bit unsigned integer
- 16BSI - 16-bit signed integer
- 16BUI - 16-bit unsigned integer
- 32BSI - 32-bit signed integer
- 32BUI - 32-bit unsigned integer
- 32BF - 32-bit float
- 64BF - 64-bit float

Examples

Code

```
SELECT ST_BandPixelType(rast, 1) As btype1,
    ST_BandPixelType(rast, 2) As btype2, ST_BandPixelType(rast, 3) As btype3
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 btype1 | btype2 | btype3
--------+--------+--------
 8BUI   | 8BUI   | 8BUI
```

See Also

ST_NumBands

11.5.8 ST_MinPossibleValue

ST_MinPossibleValue — Returns the minimum value this pixeltype can store.

Synopsis

integer **ST_MinPossibleValue**(text pixeltype);

Description

Returns the minimum value this pixeltype can store.

Examples

Code

```
SELECT ST_MinPossibleValue('16BSI');
```

Output

```
-32768
```

Code

```
SELECT ST_MinPossibleValue('8BUI');
```

Output

```
0
```

See Also

ST_BandPixelType

11.5.9 ST_HasNoBand

ST_HasNoBand — Returns true if there is no band with given band number. If no band number is specified, then band number 1 is assumed.

Synopsis

boolean **ST_HasNoBand**(raster rast, integer bandnum=1);

Description

Returns true if there is no band with given band number. If no band number is specified, then band number 1 is assumed.

Availability: 2.0.0

Examples

Code

```
SELECT rid, ST_HasNoBand(rast) As hb1, ST_HasNoBand(rast, 2) as hb2,
ST_HasNoBand(rast, 4) as hb4, ST_NumBands(rast) As numbands
FROM dummy_rast;
```

Output

```
rid | hb1 | hb2 | hb4 | numbands
-----+-----+-----+-----+----------
1 | t   | t   | t   |        0
2 | f   | f   | t   |        3
```

See Also

ST_NumBands

11.6 Raster Pixel Accessors and Setters

11.6.1 ST_PixelAsPolygon

ST_PixelAsPolygon — Returns the polygon geometry that bounds the pixel for a particular row and column.

Synopsis

geometry **ST_PixelAsPolygon**(raster rast, integer columnx, integer rowy);

Description

Returns the polygon geometry that bounds the pixel for a particular row and column.

Availability: 2.0.0

Examples

This example returns a raster pixel polygon.

Code

```
SELECT i, j, ST_PixelAsPolygon(foo.rast, i, j) As b1pgeom
FROM dummy_rast As foo
    CROSS JOIN generate_series(1, 2) As i
    CROSS JOIN generate_series(1, 1) As j
WHERE rid=2;
```

Output

```
 i | j |                                                                              b1pgeom
---+---+-------------------------------------------------------------------------------------
 1 | 1 | POLYGON((3427927.75 5793244,3427927.8 5793244,3427927.8 5793243.95,...
 2 | 1 | POLYGON((3427927.8 5793244,3427927.85 5793244,3427927.85 5793243.95, ..
```

See Also

ST_DumpAsPolygons, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsPoints, ST_PixelAsCentroid, ST_PixelAsCentroids, ST_Intersection, ST_AsText

11.6.2 ST_PixelAsPolygons

ST_PixelAsPolygons — Returns the polygon geometry that bounds every pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel.

Synopsis

setof record **ST_PixelAsPolygons**(raster rast, integer band=1, boolean exclude_nodata_value=TRUE);

Description

Returns the polygon geometry that bounds every pixel of a raster band along with the value (double precision), the X and the Y raster coordinates (integers) of each pixel.

Return record format: *geom* geometry, *val* double precision, *x* integer, *y* integers.



Note
When *exclude_nodata_value* is TRUE, only pixels whose values are not NODATA are returned. Unlike ST_DumpAsPolygons, this function returns one polygon per pixel rather than merging adjacent pixels with the same value.

Availability: 2.0.0

Enhanced: 2.1.0 exclude_nodata_value optional argument was added.

Changed: 2.1.1 Changed behavior of exclude_nodata_value.

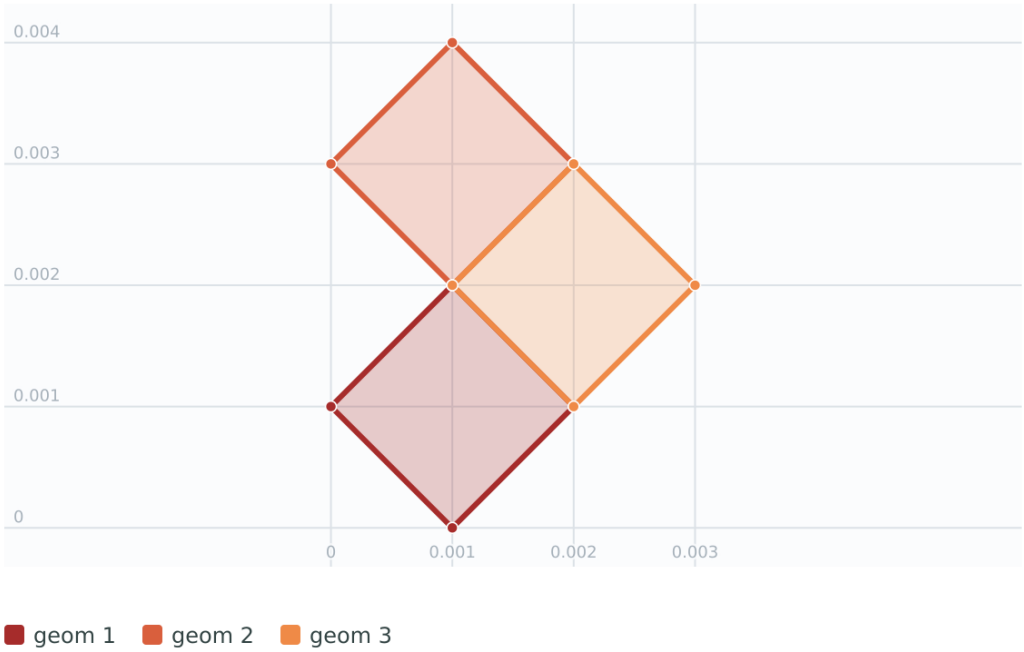
Examples

This example returns a raster pixel polygon.

Code

```
WITH raster AS (
    SELECT ST_SetValue(
        ST_SetValue(
            ST_AddBand(
                ST_MakeEmptyRaster(
                    2, 2, 0, 0, 0.001, -0.001, 0.001, 0.001, 4269
                ),
                1, '8BUI', 1, 0
            ),
            2, 2, 10
        ),
        1, 1, NULL
    ) AS rast
), pixels AS (
    SELECT (ST_PixelAsPolygons(rast)).*
    FROM raster
)
SELECT x, y, val, geom AS geom
FROM pixels
ORDER BY x, y;
```

Figure



See Also

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPoint, ST_PixelAsPoints, ST_PixelAsCentroid, ST_PixelAsCentroids, ST_AsText

11.6.3 ST_PixelAsPoint

ST_PixelAsPoint — Returns a point geometry of the pixel's upper-left corner.

Synopsis

geometry **ST_PixelAsPoint**(raster rast, integer columnx, integer rowy);

Description

Returns a point geometry of the pixel's upper-left corner.

Availability: 2.1.0

Examples

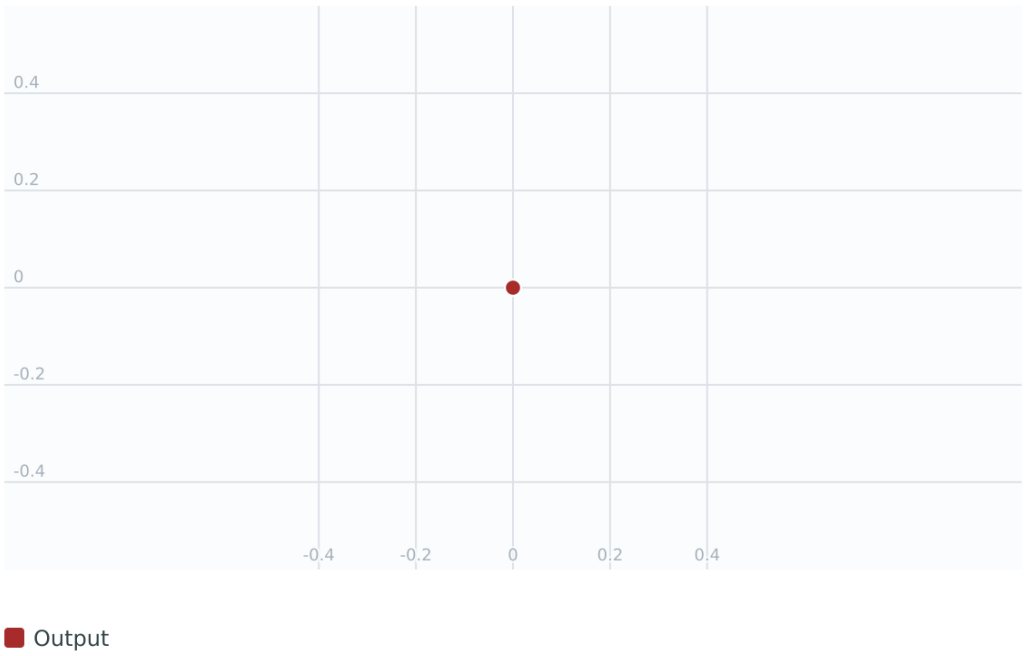
Code

```
SELECT ST_PixelAsPoint(rast, 1, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
POINT(0.5 0.5)
```

Figure



See Also

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoints, ST_PixelAsCentroid, ST_PixelAsCentroids

11.6.4 ST_PixelAsPoints

ST_PixelAsPoints — Returns a point geometry for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The coordinates of the point geometry are of the pixel's upper-left corner.

Synopsis

setof record **ST_PixelAsPoints**(raster rast, integer band=1, boolean exclude_nodata_value=TRUE);

Description

Returns a point geometry for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The coordinates of the point geometry are of the pixel's upper-left corner.

Return record format: *geom* geometry, *val* double precision, *x* integer, *y* integers.



Note

When *exclude_nodata_value* = TRUE, only those pixels whose values are not NODATA are returned as points.

Availability: 2.1.0

Changed: 2.1.1 Changed behavior of exclude_nodata_value.

Examples

This example returns point features together with a derived marker size that a client can use for value-scaled point rendering.

Code

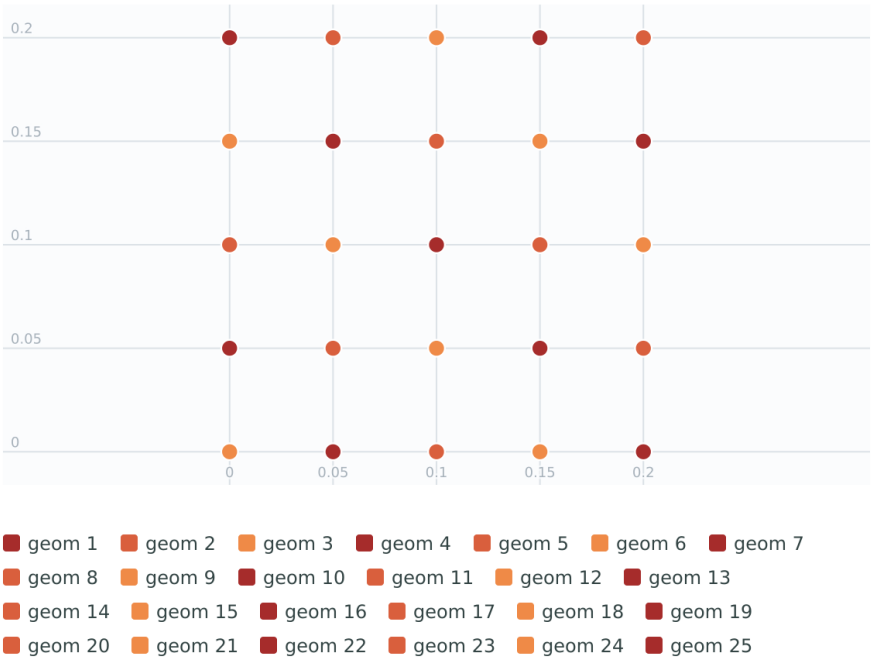
```
SELECT
    x,
    y,
    val,
    round((val / 32.0)::numeric, 2) AS marker_scale,geom AS geom
FROM (
    SELECT (ST_PixelAsPoints(rast, 1)).*
    FROM dummy_rast
    WHERE rid = 2
) AS foo
ORDER BY y, x;
```

Output

```
 x | y | val | marker_scale |          geom
---+---+-----+--------------+----------------------------
 1 | 1 | 253 |         7.91 | POINT(3427927.75 5793244)
 2 | 1 | 254 |         7.94 | POINT(3427927.8 5793244)
 3 | 1 | 253 |         7.91 | POINT(3427927.85 5793244)
 4 | 1 | 254 |         7.94 | POINT(3427927.9 5793244)
 5 | 1 | 254 |         7.94 | POINT(3427927.95 5793244)
 1 | 2 | 253 |         7.91 | POINT(3427927.75 5793243.95)
 2 | 2 | 254 |         7.94 | POINT(3427927.8 5793243.95)
 3 | 2 | 254 |         7.94 | POINT(3427927.85 5793243.95)
```

```
 4 | 2 | 253 |          7.91 | POINT(3427927.9 5793243.95)
 5 | 2 | 249 |          7.78 | POINT(3427927.95 5793243.95)
 1 | 3 | 250 |          7.81 | POINT(3427927.75 5793243.9)
 2 | 3 | 254 |          7.94 | POINT(3427927.8 5793243.9)
 3 | 3 | 254 |          7.94 | POINT(3427927.85 5793243.9)
 4 | 3 | 252 |          7.88 | POINT(3427927.9 5793243.9)
 5 | 3 | 249 |          7.78 | POINT(3427927.95 5793243.9)
 1 | 4 | 251 |          7.84 | POINT(3427927.75 5793243.85)
 2 | 4 | 253 |          7.91 | POINT(3427927.8 5793243.85)
 3 | 4 | 254 |          7.94 | POINT(3427927.85 5793243.85)
 4 | 4 | 254 |          7.94 | POINT(3427927.9 5793243.85)
 5 | 4 | 253 |          7.91 | POINT(3427927.95 5793243.85)
 1 | 5 | 252 |          7.88 | POINT(3427927.75 5793243.8)
 2 | 5 | 250 |          7.81 | POINT(3427927.8 5793243.8)
 3 | 5 | 254 |          7.94 | POINT(3427927.85 5793243.8)
 4 | 5 | 254 |          7.94 | POINT(3427927.9 5793243.8)
 5 | 5 | 254 |          7.94 | POINT(3427927.95 5793243.8)
```

Figure



See Also

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsCentroid, ST_PixelAsCentroids

11.6.5 ST_PixelAsCentroid

ST_PixelAsCentroid — Returns the centroid (point geometry) of the area represented by a pixel.

Synopsis

geometry **ST_PixelAsCentroid**(raster rast, integer x, integer y);

Description

Returns the centroid (point geometry) of the area represented by a pixel.

Enhanced: 3.2.0 Faster now implemented in C.

Availability: 2.1.0

Examples

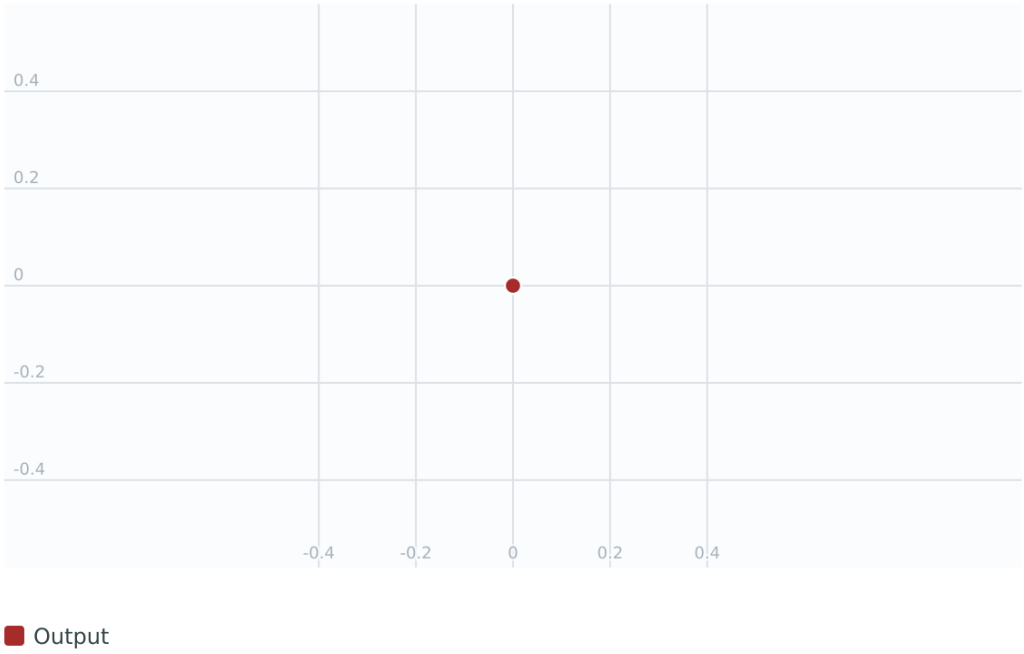
Code

```
SELECT ST_PixelAsCentroid(rast, 1, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
POINT(1.5 2)
```

Figure



See Also

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsPoints, ST_PixelAsCentroids

11.6.6 ST_PixelAsCentroids

ST_PixelAsCentroids — Returns the centroid (point geometry) for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The point geometry is the centroid of the area represented by a pixel.

Synopsis

setof record **ST_PixelAsCentroids**(raster rast, integer band=1, boolean exclude_nodata_value=TRUE);

Description

Returns the centroid (point geometry) for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The point geometry is the centroid of the area represented by a pixel.

Return record format: *geom* geometry, *val* double precision, *x* integer, *y* integers.



Note

When *exclude_nodata_value* = TRUE, only those pixels whose values are not NODATA are returned as points.

Enhanced: 3.2.0 Faster now implemented in C.

Changed: 2.1.1 Changed behavior of exclude_nodata_value.

Availability: 2.1.0

Examples

This example uses LATERAL syntax.

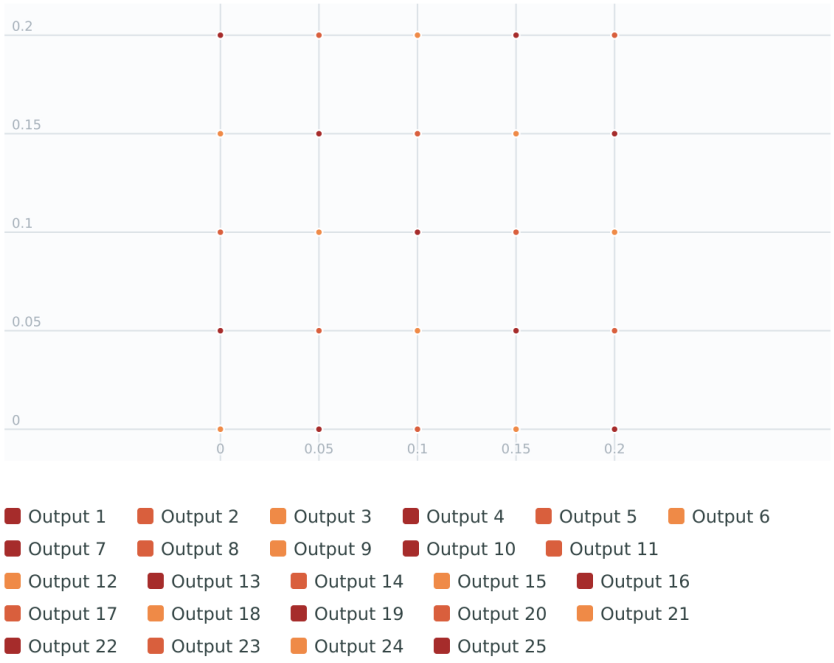
Code

```
SELECT x, y, val,geom
    FROM (SELECT dp.* FROM dummy_rast, LATERAL ST_PixelAsCentroids(rast, 1) AS dp WHERE rid ↩
    = 2) foo;
```

Output

```
 x | y | val |           st_astext
---+---+-----+-------------------------------
 1 | 1 | 253 | POINT(3427927.775 5793243.975)
 2 | 1 | 254 | POINT(3427927.825 5793243.975)
 3 | 1 | 253 | POINT(3427927.875 5793243.975)
 4 | 1 | 254 | POINT(3427927.925 5793243.975)
 5 | 1 | 254 | POINT(3427927.975 5793243.975)
 1 | 2 | 253 | POINT(3427927.775 5793243.925)
 2 | 2 | 254 | POINT(3427927.825 5793243.925)
 3 | 2 | 254 | POINT(3427927.875 5793243.925)
 4 | 2 | 253 | POINT(3427927.925 5793243.925)
 5 | 2 | 249 | POINT(3427927.975 5793243.925)
 1 | 3 | 250 | POINT(3427927.775 5793243.875)
 2 | 3 | 254 | POINT(3427927.825 5793243.875)
 3 | 3 | 254 | POINT(3427927.875 5793243.875)
 4 | 3 | 252 | POINT(3427927.925 5793243.875)
 5 | 3 | 249 | POINT(3427927.975 5793243.875)
 1 | 4 | 251 | POINT(3427927.775 5793243.825)
 2 | 4 | 253 | POINT(3427927.825 5793243.825)
 3 | 4 | 254 | POINT(3427927.875 5793243.825)
 4 | 4 | 254 | POINT(3427927.925 5793243.825)
 5 | 4 | 253 | POINT(3427927.975 5793243.825)
 1 | 5 | 252 | POINT(3427927.775 5793243.775)
 2 | 5 | 250 | POINT(3427927.825 5793243.775)
 3 | 5 | 254 | POINT(3427927.875 5793243.775)
 4 | 5 | 254 | POINT(3427927.925 5793243.775)
 5 | 5 | 254 | POINT(3427927.975 5793243.775)
```

Figure



See Also

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsPoints, ST_PixelAsCentroid

11.6.7 ST_Value

ST_Value — Returns the value of a given band in a given columnx, rowy pixel or at a particular geometric point. Band numbers start at 1 and assumed to be 1 if not specified. If `exclude_nodata_value` is set to false, then all pixels include `nodata` pixels are considered to intersect and return value. If `exclude_nodata_value` is not passed in then reads it from metadata of raster.

Synopsis

double precision **ST_Value**(raster rast, geometry pt, boolean exclude_nodata_value=true);
double precision **ST_Value**(raster rast, integer band, geometry pt, boolean exclude_nodata_value=true, text resample='nearest');
double precision **ST_Value**(raster rast, integer x, integer y, boolean exclude_nodata_value=true);
double precision **ST_Value**(raster rast, integer band, integer x, integer y, boolean exclude_nodata_value=true);

Description

Returns the value of a given band in a given columnx, rowy pixel or at a given geometry point. Band numbers start at 1 and band is assumed to be 1 if not specified.

If `exclude_nodata_value` is set to true, then only non `nodata` pixels are considered. If `exclude_nodata_value` is set to false, then all pixels are considered.

The allowed values of the `resample` parameter are "nearest" which performs the default nearest-neighbor resampling, "bilinear" which performs a bilinear interpolation to estimate the value between pixel centers, and the nearest-neighbor boundary options "nearest-UL", "nearest-UR", "nearest-LL", and "nearest-LR". Boundary options choose which pixel is returned when the point lies on a horizontal or vertical pixel boundary, including corner intersections.

Enhanced: 3.7.0 resample accepts nearest-neighbor boundary options "nearest-UL", "nearest-UR", "nearest-LL", and "nearest-LR".

Enhanced: 3.2.0 resample optional argument was added.

Enhanced: 2.0.0 exclude_nodata_value optional argument was added.

Examples

Get raster values at particular postgis geometry points. The srid of your geometry should be same as for your raster.

Code

```
SELECT rid, ST_Value(rast, foo.pt_geom) As b1pval, ST_Value(rast, 2, foo.pt_geom) As b2pval
FROM dummy_rast CROSS JOIN (SELECT ST_SetSRID(ST_Point(3427927.77, 5793243.76), 0) As ↩
    pt_geom) As foo
WHERE rid=2;
```

Output

```
 rid | b1pval | b2pval
-----+--------+--------
   2 |    252 |     79
```

General fictitious example using a real table.

Code

```
SELECT rid, ST_Value(rast, 3, sometable.geom) As b3pval
FROM sometable
WHERE ST_Intersects(rast, sometable.geom);
```

Code

```
SELECT rid, ST_Value(rast, 1, 1, 1) As b1pval,
    ST_Value(rast, 2, 1, 1) As b2pval, ST_Value(rast, 3, 1, 1) As b3pval
FROM dummy_rast
WHERE rid=2;
```

Output

```
 rid | b1pval | b2pval | b3pval
-----+--------+--------+--------
   2 |    253 |     78 |     70
```

Get all values in bands 1,2,3 of each pixel.

Code

```


SELECT x, y, ST_Value(rast, 1, x, y) As b1val,
    ST_Value(rast, 2, x, y) As b2val, ST_Value(rast, 3, x, y) As b3val
FROM dummy_rast CROSS JOIN
generate_series(1, 1000) As x CROSS JOIN generate_series(1, 1000) As y
WHERE rid =  2 AND x <= ST_Width(rast) AND y <= ST_Height(rast);
```

Output

```
 x | y | b1val | b2val | b3val
---+---+-------+-------+-------
 1 | 1 |   253 |    78 |    70
 1 | 2 |   253 |    96 |    80
 1 | 3 |   250 |    99 |    90
 1 | 4 |   251 |    89 |    77
 1 | 5 |   252 |    79 |    62
 2 | 1 |   254 |    98 |    86
 2 | 2 |   254 |   118 |   108
 :
 :
```

Get all values in bands 1,2,3 of each pixel same as above but returning the upper left point point of each pixel.

Code

```
SELECT ST_SetSRID(ST_Point(ST_UpperLeftX(rast) + ST_ScaleX(rast)*x,
        ST_UpperLeftY(rast) + ST_ScaleY(rast)*y),
        ST_SRID(rast)) As uplpt,
    ST_Value(rast, 1, x, y) As b1val,
    ST_Value(rast, 2, x, y) As b2val, ST_Value(rast, 3, x, y) As b3val
FROM dummy_rast CROSS JOIN
generate_series(1,1000) As x CROSS JOIN generate_series(1,1000) As y
WHERE rid =  2 AND x <= ST_Width(rast) AND y <= ST_Height(rast);
```

Output

```
            uplpt            | b1val | b2val | b3val
-----------------------------+-------+-------+-------
 POINT(3427929.25 5793245.5) |   253 |    78 |    70
 POINT(3427929.25 5793247)   |   253 |    96 |    80
 POINT(3427929.25 5793248.5) |   250 |    99 |    90
:
```

Get a polygon formed by union of all pixels that fall in a particular value range and intersect a particular polygon.

Code

```
SELECT ST_Union(pixpolyg) As shadow
FROM (SELECT ST_Translate(ST_MakeEnvelope(ST_UpperLeftX(rast), ST_UpperLeftY(rast),
            ST_UpperLeftX(rast) + ST_ScaleX(rast),
            ST_UpperLeftY(rast) + ST_ScaleY(rast), 0
            ), ST_ScaleX(rast)*x, ST_ScaleY(rast)*y
        ) As pixpolyg, ST_Value(rast, 2, x, y) As b2val
    FROM dummy_rast CROSS JOIN
generate_series(1,1000) As x CROSS JOIN generate_series(1,1000) As y
WHERE rid =  2
    AND x <= ST_Width(rast) AND y <= ST_Height(rast)) As foo
WHERE
    ST_Intersects(
        pixpolyg,
        ST_GeomFromText('POLYGON((3427928 5793244,3427927.75 5793243.75,3427928  ↩
    5793243.75,3427928 5793244))',0)
        ) AND b2val != 254;
```

Output

```
        shadow
-------------------------------------------------------------------------------
 MULTIPOLYGON(((3427928 5793243.9,3427928 5793243.85,3427927.95 5793243.85,3427927.95 5793243.9,
 3427927.95 5793243.95,3427928 5793243.95,3427928.05 5793243.95,3427928.05 5793243.9,3427928 5793243.9)),((3427927.95  ↩
    5793243.9,3427927.95 579324
3.85,3427927.9 5793243.85,3427927.85 5793243.85,3427927.85 5793243.9,3427927.9 5793243.9,3427927.9 5793243.95,
3427927.95 5793243.95,3427927.95 5793243.9)),((3427927.85 5793243.75,3427927.85 5793243.7,3427927.8 5793243.7,3427927.8 ↩
     5793243.75,
```

```
3427927.8 5793243.8,3427927.8 5793243.85,3427927.85 5793243.85,3427927.85 5793243.8,3427927.85 5793243.75)),
((3427928.05 5793243.75,3427928.05 5793243.7,3427928 5793243.7,3427927.95 5793243.7,3427927.95 5793243.75,3427927.95 ↩
    5793243.8,3427
927.95 5793243.85,3427928 5793243.85,3427928 5793243.8,3427928.05 5793243.8,
3427928.05 5793243.75)),((3427927.95 5793243.75,3427927.95 5793243.7,3427927.9 5793243.7,3427927.85 5793243.7,
3427927.85 5793243.75,3427927.85 5793243.8,3427927.85 5793243.85,3427927.9 5793243.85,
3427927.95 5793243.85,3427927.95 5793243.8,3427927.95 5793243.75)))
```

Checking all the pixels of a large raster tile can take a long time. You can dramatically improve speed at some lose of precision by orders of magnitude. By sampling pixels using the step optional parameter of generate_series. This next example does the same as previous but by checking 1 for every 4 (2x2) pixels and putting in the last checked. Putting in the checked pixel as the value for subsequent 4.

Code

```
SELECT ST_Union(pixpolyg) As shadow
FROM (SELECT ST_Translate(ST_MakeEnvelope(ST_UpperLeftX(rast), ST_UpperLeftY(rast),
            ST_UpperLeftX(rast) + ST_ScaleX(rast)*2,
            ST_UpperLeftY(rast) + ST_ScaleY(rast)*2, 0
            ), ST_ScaleX(rast)*x, ST_ScaleY(rast)*y
        ) As pixpolyg, ST_Value(rast, 2, x, y) As b2val
    FROM dummy_rast CROSS JOIN
generate_series(1,1000,2) As x CROSS JOIN generate_series(1,1000,2) As y
WHERE rid =  2
    AND x <= ST_Width(rast)  AND y <= ST_Height(rast)  ) As foo
WHERE
    ST_Intersects(
        pixpolyg,
        ST_GeomFromText('POLYGON((3427928 5793244,3427927.75 5793243.75,3427928 ↩
    5793243.75,3427928 5793244))',0)
        ) AND b2val != 254;
```

Output

```
        shadow
-----------------------------------------------------------------------------------
 MULTIPOLYGON(((3427927.9 5793243.85,3427927.8 5793243.85,3427927.8 5793243.95,
 3427927.9 5793243.95,3427928 5793243.95,3427928.1 5793243.95,3427928.1 5793243.85,3427928 ↩
    5793243.85,3427927.9 5793243.85)),
 ((3427927.9 5793243.65,3427927.8 5793243.65,3427927.8 5793243.75,3427927.8 ↩
    5793243.85,3427927.9 5793243.85,
 3427928 5793243.85,3427928 5793243.75,3427928.1 5793243.75,3427928.1 5793243.65,3427928 ↩
    5793243.65,3427927.9 5793243.65)))
```

See Also

ST_DumpValues, ST_SetValue, ST_DumpAsPolygons, ST_NumBands, ST_PixelAsPolygon, ST_ScaleX, ST_ScaleY, ST_UpperLeftX ST_UpperLeftY, ST_SRID, ST_AsText, ST_Point, ST_MakeEnvelope, ST_Intersects, ST_Intersection

11.6.8 ST_NearestValue

ST_NearestValue — Returns the nearest non-NODATA value of a given band's pixel specified by a columnx and rowy or a geometric point expressed in the same spatial reference coordinate system as the raster.

Synopsis

double precision **ST_NearestValue**(raster rast, integer bandnum, geometry pt, boolean exclude_nodata_value=true);
double precision **ST_NearestValue**(raster rast, geometry pt, boolean exclude_nodata_value=true);
double precision **ST_NearestValue**(raster rast, integer bandnum, integer columnx, integer rowy, boolean exclude_nodata_value=true);
double precision **ST_NearestValue**(raster rast, integer columnx, integer rowy, boolean exclude_nodata_value=true);

Description

Returns the nearest non-NODATA value of a given band in a given columnx, rowy pixel or at a specific geometric point. If the columnx, rowy pixel or the pixel at the specified geometric point is NODATA, the function will find the nearest pixel to the columnx, rowy pixel or geometric point whose value is not NODATA.

Band numbers start at 1 and bandnum is assumed to be 1 if not specified. If exclude_nodata_value is set to false, then all pixels include nodata pixels are considered to intersect and return value. If exclude_nodata_value is not passed in then reads it from metadata of raster.

Availability: 2.1.0



Note
ST_NearestValue is a drop-in replacement for ST_Value.

Examples

This example reads a value from pixel 2x2.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
            1, '8BUI', 1, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 0],
            [1, 1, 0, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_Value(rast, 2, 2) AS value,
    ST_NearestValue(rast, 2, 2) AS nearestvalue
FROM source
```

Output

```
 value | nearestvalue
-------+--------------
     1 |            1
```

This example reads a NODATA value from pixel 2x3.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
            1, '8BUI', 1, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 0],
            [1, 1, 0, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_Value(rast, 2, 3) AS value,
    ST_NearestValue(rast, 2, 3) AS nearestvalue
FROM source
```

Output

```
 value | nearestvalue
-------+--------------
       |            1
```

See Also

ST_Neighborhood, ST_Value

11.6.9 ST_SetZ

ST_SetZ — Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimension using the requested resample algorithm.

Synopsis

geometry **ST_SetZ**(raster rast, geometry geom, text resample=nearest, integer band=1);

Description

Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimensions using the requested resample algorithm.

The `resample` parameter can be set to "nearest" to copy the values from the cell each vertex falls within, or "bilinear" to use bilinear interpolation to calculate a value that takes neighboring cells into account also.

Availability: 3.2.0

Examples

2x2 test raster with values. 10 50. 40 20.

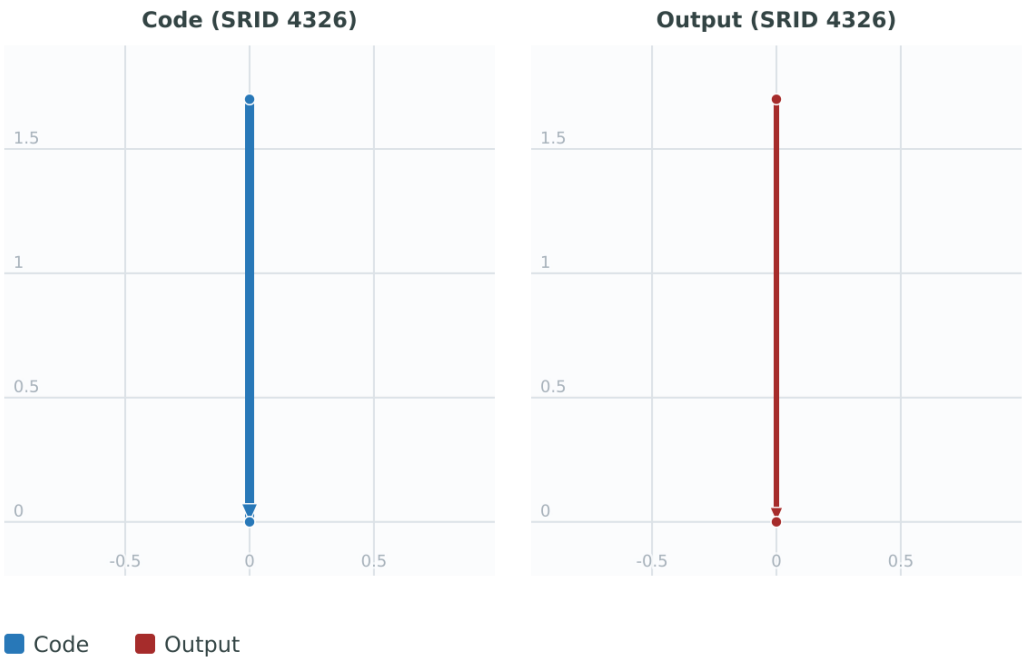
Code

```
WITH test_raster AS (
SELECT
ST_SetValues(
ST_AddBand(
ST_MakeEmptyRaster(
      width => 2,
      height => 2,
      upperleftx => 0,
      upperlefty => 2,
      scalex => 1.0,
      scaley => -1.0,
      skewx => 0,
      skewy => 0,
      srid => 4326),
    index => 1,
    pixeltype => '16BSI',
    initialvalue => 0,
    nodataval => -999
),
  1,
  1,
  1,
  newvalueset => ARRAY[ARRAY[10.0::float8, 50.0::float8], ARRAY[40.0::float8, 20.0::float8 ↩
    ]]) AS rast
)
SELECT
ST_SetZ(
    rast,
    band => 1,
    geom => 'SRID=4326;LINESTRING(1.0 1.9,1.0 0.2)'::geometry,
    resample => 'bilinear'
)
FROM test_raster
```

Output

```
LINESTRING Z (1 1.9 38,1 0.2 27)
```

Figure



See Also

ST_Value, ST_SetM

11.6.10 ST_SetM

ST_SetM — Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimension using the requested resample algorithm.

Synopsis

geometry **ST_SetM**(raster rast, geometry geom, text resample=nearest, integer band=1);

Description

Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimensions using the requested resample algorithm.

The `resample` parameter can be set to "nearest" to copy the values from the cell each vertex falls within, or "bilinear" to use bilinear interpolation to calculate a value that takes neighboring cells into account also.

Availability: 3.2.0

Examples

2x2 test raster with values. 10 50. 40 20.

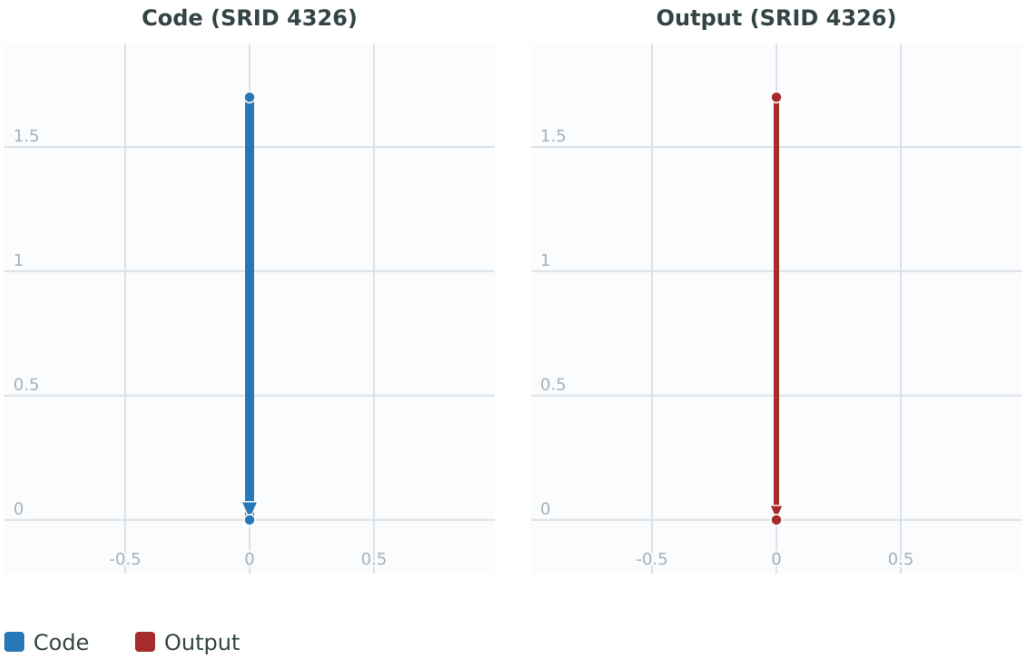
Code

```
WITH test_raster AS (
SELECT
ST_SetValues(
ST_AddBand(
ST_MakeEmptyRaster(
      width => 2,
      height => 2,
      upperleftx => 0,
      upperlefty => 2,
      scalex => 1.0,
      scaley => -1.0,
      skewx => 0,
      skewy => 0,
      srid => 4326),
    index => 1,
    pixeltype => '16BSI',
    initialvalue => 0,
    nodataval => -999
),
  1,
  1,
  1,
  newvalueset => ARRAY[ARRAY[10.0::float8, 50.0::float8], ARRAY[40.0::float8, 20.0::float8 ↩
    ]]) AS rast
)
SELECT
ST_SetM(
    rast,
    band => 1,
    geom => 'SRID=4326;LINESTRING(1.0 1.9,1.0 0.2)'::geometry,
    resample => 'bilinear'
)
FROM test_raster
```

Output

```
LINESTRING M (1 1.9 38,1 0.2 27)
```

Figure



See Also

ST_Value, ST_SetZ

11.6.11 ST_Neighborhood

ST_Neighborhood — Returns a 2-D double precision array of the non-NODATA values around a given band's pixel specified by either a columnX and rowY or a geometric point expressed in the same spatial reference coordinate system as the raster.

Synopsis

double precision[][] **ST_Neighborhood**(raster rast, integer bandnum, integer columnX, integer rowY, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);
double precision[][] **ST_Neighborhood**(raster rast, integer columnX, integer rowY, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);
double precision[][] **ST_Neighborhood**(raster rast, integer bandnum, geometry pt, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);
double precision[][] **ST_Neighborhood**(raster rast, geometry pt, integer distanceX, integer distanceY, boolean exclude_nodata_value=tru

Description

Returns a 2-D double precision array of the non-NODATA values around a given band's pixel specified by either a columnX and rowY or a geometric point expressed in the same spatial reference coordinate system as the raster. The distanceX and distanceY parameters define the number of pixels around the specified pixel in the X and Y axes, e.g. I want all values within 3 pixel distance along the X axis and 2 pixel distance along the Y axis around my pixel of interest. The center value of the 2-D array will be the value at the pixel specified by the columnX and rowY or the geometric point.

Band numbers start at 1 and bandnum is assumed to be 1 if not specified. If exclude_nodata_value is set to false, then all pixels include nodata pixels are considered to intersect and return value. If exclude_nodata_value is not passed in then reads it from metadata of raster.



Note
The number of elements along each axis of the returning 2-D array is 2 * (distanceX|distanceY) + 1. So for a distanceX and distanceY of 1, the returning array will be 3x3.



Note
The 2-D array output can be passed to any of the raster processing builtin functions, e.g. ST_Min4ma, ST_Sum4ma, ST_Mean4ma.

Availability: 2.1.0

Examples

This example reads a value from pixel 2x2.

Code

```
SELECT
    ST_Neighborhood(rast, 2, 2, 1, 1)
FROM (
    SELECT
        ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
                '8BUI'::text, 1, 0
            ),
            1, 1, 1, ARRAY[
                [0, 1, 1, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 1]
            ]::double precision[],
            1
        ) AS rast
) AS foo
```

Output

```
{{NULL,1,1},{1,1,1},{1,NULL,1}}
```

This example reads a NODATA value from pixel 2x3.

Code

```
SELECT
    ST_Neighborhood(rast, 2, 3, 1, 1)
FROM (
    SELECT
        ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
                '8BUI'::text, 1, 0
            ),
            1, 1, 1, ARRAY[
                [0, 1, 1, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 1]
            ]::double precision[],
```

```
            1
        ) AS rast
) AS foo
```

Output

```
{{1,1,1},{1,NULL,1},{1,1,1}}
```

Pixel 3x3 has value. Exclude_nodata_value = FALSE.

Code

```
SELECT
    ST_Neighborhood(rast, 3, 3, 1, 1, false)
FROM ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
                '8BUI'::text, 1, 0
            ),
            1, 1, 1, ARRAY[
                [0, 1, 1, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 1]
            ]::double precision[],
            1
        ) AS rast
```

Output

```
{{1,1,0},{0,1,1},{1,1,1}}
```

See Also

ST_NearestValue, ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.6.12 ST_SetValue

ST_SetValue — Returns modified raster resulting from setting the value of a given band in a given columnx, rowy pixel or the pixels that intersect a particular geometry. Band numbers start at 1 and assumed to be 1 if not specified.

Synopsis

raster **ST_SetValue**(raster rast, integer bandnum, geometry geom, double precision newvalue);
raster **ST_SetValue**(raster rast, geometry geom, double precision newvalue);
raster **ST_SetValue**(raster rast, integer bandnum, integer columnx, integer rowy, double precision newvalue);
raster **ST_SetValue**(raster rast, integer columnx, integer rowy, double precision newvalue);

Description

Returns modified raster resulting from setting the specified pixels' values to new value for the designated band given the raster's row and column or a geometry. If no band is specified, then band 1 is assumed.

Enhanced: 2.1.0 Geometry variant of ST_SetValue() now supports any geometry type, not just point. The geometry variant is a wrapper around the geomval[] variant of ST_SetValues()

Examples

This example uses geometry input.

Code

```
                SELECT (foo.geomval).val, ST_Union((foo.geomval).geom)
FROM (SELECT ST_DumpAsPolygons(ST_SetValue(
            rast,
            1,
            ST_Point(3427927.75, 5793243.95),
            50)
            ) As geomval
FROM dummy_rast
where rid = 2) As foo
WHERE (foo.geomval).val < 250
GROUP BY (foo.geomval).val;
```

Output

```
 val |                                                    st_astext
-----+-----------------------------------------------------------------------
  50 | POLYGON((3427927.75 5793244,3427927.75 5793243.95,3427927.8 579324 ...
 249 | POLYGON((3427927.95 5793243.95,3427927.95 5793243.85,3427928 57932 ...
```

This example stores the changed raster.

Code

```
UPDATE dummy_rast SET rast = ST_SetValue(rast, 1, ST_Point(3427927.75, 5793243.95), 100)
    WHERE rid = 2   ;
```

See Also

ST_Value, ST_DumpAsPolygons

11.6.13 ST_SetValues

ST_SetValues — Returns modified raster resulting from setting the values of a given band.

Synopsis

raster **ST_SetValues**(raster rast, integer nband, integer columnx, integer rowy, double precision[][] newvalueset, boolean[][] noset=NULL, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer nband, integer columnx, integer rowy, double precision[][] newvalueset, double precision nosetvalue, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer nband, integer columnx, integer rowy, integer width, integer height, double precision newvalue, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer columnx, integer rowy, integer width, integer height, double precision newvalue, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer nband, geomval[] geomvalset, boolean keepnodata=FALSE);

Description

Returns modified raster resulting from setting specified pixels to new value(s) for the designated band. `columnx` and `rowy` are 1-indexed.

If `keepnodata` is TRUE, those pixels whose values are NODATA will not be set with the corresponding value in `newvalueset`.

For Variant 1, the specific pixels to be set are determined by the `columnx`, `rowy` pixel coordinates and the dimensions of the `newvalueset` array. `noset` can be used to prevent pixels with values present in `newvalueset` from being set (due to PostgreSQL not permitting ragged/jagged arrays). See example Variant 1.

Variant 2 is like Variant 1 but with a simple double precision `nosetvalue` instead of a boolean `noset` array. Elements in `newvalueset` with the `nosetvalue` value with be skipped. See example Variant 2.

For Variant 3, the specific pixels to be set are determined by the `columnx`, `rowy` pixel coordinates, `width` and `height`. See example Variant 3.

Variant 4 is the same as Variant 3 with the exception that it assumes that the first band's pixels of `rast` will be set.

For Variant 5, an array of geomval is used to determine the specific pixels to be set. If all the geometries in the array are of type POINT or MULTIPOINT, the function uses a shortcut where the longitude and latitude of each point is used to set a pixel directly. Otherwise, the geometries are converted to rasters and then iterated through in one pass. See example Variant 5.

Availability: 2.1.0

Examples

Variant 1.

These examples pass `FALSE` for *`exclude_nodata_value`* so that every raster cell remains visible. NODATA cells display the band's NODATA value, which is `0` here.

Set the lower-right 2 by 2 block of a 3 by 3 raster to `9`.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 2, 2, ARRAY[[9, 9], [9, 9]]::double precision[][]
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

Replace the complete raster from an array; the central `NULL` becomes NODATA.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1, ARRAY[[9, 9, 9], [9, NULL, 9], [9, 9, 9]]::double precision[][]
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   9
 1 | 2 |   9
 1 | 3 |   9
 2 | 1 |   9
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   9
 3 | 2 |   9
 3 | 3 |   9
```

Use the *noset* mask to retain the left-middle source pixel while replacing the other cells.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1,
                ARRAY[[9, 9, 9], [9, NULL, 9], [9, 9, 9]]::double precision[][],
                ARRAY[[false], [true]]::boolean[][]
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   9
 1 | 2 |   1
 1 | 3 |   9
 2 | 1 |   9
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   9
 3 | 2 |   9
 3 | 3 |   9
```

With *keepnodata* enabled, an existing NODATA pixel is not overwritten.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, ↩
     -1, 0, 0, 0),
                    1, '8BUI', 1, 0
                ),
                1, 1, 1, NULL
            ),
            1, 1, 1,
                ARRAY[[9, 9, 9], [9, NULL, 9], [9, 9, 9]]::double precision[][],
                ARRAY[[false], [true]]::boolean[][],
                TRUE
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   0
 1 | 2 |   1
 1 | 3 |   9
 2 | 1 |   9
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   9
 3 | 2 |   9
 3 | 3 |   9
```

Variant 2.

Use `-1` as the value that should not be written; only the lower-right 2 by 2 block changes.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1, ARRAY[[-1, -1, -1], [-1, 9, 9], [-1, 9, 9]]::double precision[][], -1
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

The same update can use `NULL` as the value that should not be written.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1, ARRAY[[NULL, NULL, NULL], [NULL, 9, 9], [NULL, 9, 9]]::double precision ↩
    [][], NULL::double precision
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

Variant 3.

Set a 2 by 2 rectangle starting at column 2, row 2.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 2, 2, 2, 2, 9
        ),
        1, FALSE
```

```
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

When *keepnodata* is true, the NODATA cell inside the rectangle remains unchanged.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, ↩
     -1, 0, 0, 0),
                    1, '8BUI', 1, 0
                ),
                1, 2, 2, NULL
            ),
            1, 2, 2, 2, 2, 9, TRUE
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

Variant 5.

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    0, 0) AS rast
), bar AS (
    SELECT 1 AS gid, 'SRID=0;POINT(2.5 -2.5)'::geometry geom UNION ALL
    SELECT 2 AS gid, 'SRID=0;POLYGON((1 -1,4 -1,4 -4,1 -4,1 -1))'::geometry geom UNION ALL
    SELECT 3 AS gid, 'SRID=0;POLYGON((0 0,5 0,5 -1,1 -1,1 -4,0 -4,0 0))'::geometry geom  ↩
    UNION ALL
```

```
    SELECT 4 AS gid, 'SRID=0;MULTIPOINT(0 0,4 4,4 -4)'::geometry
), changed AS (
    SELECT rid, gid, ST_SetValue(rast, 1, geom, gid) AS rast
    FROM foo
    CROSS JOIN bar
)
SELECT
    rid,
    gid,
    y,
    array_agg(ST_Value(rast, 1, x, y) ORDER BY x) AS values
FROM changed
CROSS JOIN generate_series(1, 5) AS x
CROSS JOIN generate_series(1, 5) AS y
GROUP BY rid, gid, y
ORDER BY rid, gid, y;
```

Output

```
 rid | gid | y |           values
-----+-----+---+----------------------------
   1 |   1 | 1 | {NULL,NULL,NULL,NULL,NULL}
   1 |   1 | 2 | {NULL,NULL,NULL,NULL,NULL}
   1 |   1 | 3 | {NULL,NULL,1,NULL,NULL}
   1 |   1 | 4 | {NULL,NULL,NULL,NULL,NULL}
   1 |   1 | 5 | {NULL,NULL,NULL,NULL,NULL}
   1 |   2 | 1 | {NULL,NULL,NULL,NULL,NULL}
   1 |   2 | 2 | {NULL,2,2,2,NULL}
   1 |   2 | 3 | {NULL,2,2,2,NULL}
   1 |   2 | 4 | {NULL,2,2,2,NULL}
   1 |   2 | 5 | {NULL,NULL,NULL,NULL,NULL}
   1 |   3 | 1 | {3,3,3,3,3}
   1 |   3 | 2 | {3,NULL,NULL,NULL,NULL}
   1 |   3 | 3 | {3,NULL,NULL,NULL,NULL}
   1 |   3 | 4 | {3,NULL,NULL,NULL,NULL}
   1 |   3 | 5 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 1 | {4,NULL,NULL,NULL,NULL}
   1 |   4 | 2 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 3 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 4 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 5 | {NULL,NULL,NULL,NULL,4}
(20 rows)
```

The following shows that geomvals later in the array can overwrite prior geomvals

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
    0, 0) AS rast
), bar AS (
    SELECT 1 AS gid, 'SRID=0;POINT(2.5 -2.5)'::geometry geom UNION ALL
    SELECT 2 AS gid, 'SRID=0;POLYGON((1 -1,4 -1,4 -4,1 -4,1 -1))'::geometry geom UNION ALL
    SELECT 3 AS gid, 'SRID=0;POLYGON((0 0,5 0,5 -1,1 -1,1 -4,0 -4,0 0))'::geometry geom ↩
    UNION ALL
    SELECT 4 AS gid, 'SRID=0;MULTIPOINT(0 0,4 4,4 -4)'::geometry
), changed AS (
    SELECT ST_SetValues(
        rast,
        1,
        ARRAY[
            ROW((SELECT geom FROM bar WHERE gid = 1), 1),
            ROW((SELECT geom FROM bar WHERE gid = 2), 2)
        ]::geomval[]
    ) AS rast
    FROM foo
)
SELECT
```

```
    y,
    array_agg(ST_Value(rast, 1, x, y) ORDER BY x) AS values
FROM changed
CROSS JOIN generate_series(1, 5) AS x
CROSS JOIN generate_series(1, 5) AS y
GROUP BY y
ORDER BY y;
```

Output

```
 y |          values
---+--------------------------
 1 | {NULL,NULL,NULL,NULL,NULL}
 2 | {NULL,2,2,2,NULL}
 3 | {NULL,2,2,2,NULL}
 4 | {NULL,2,2,2,NULL}
 5 | {NULL,NULL,NULL,NULL,NULL}
(5 rows)
```

This example is the opposite of the prior example

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
    0, 0) AS rast
), bar AS (
    SELECT 1 AS gid, 'SRID=0;POINT(2.5 -2.5)'::geometry geom UNION ALL
    SELECT 2 AS gid, 'SRID=0;POLYGON((1 -1,4 -1,4 -4,1 -4,1 -1))'::geometry geom UNION ALL
    SELECT 3 AS gid, 'SRID=0;POLYGON((0 0,5 0,5 -1,1 -1,1 -4,0 -4,0 0))'::geometry geom ↩
    UNION ALL
    SELECT 4 AS gid, 'SRID=0;MULTIPOINT(0 0,4 4,4 -4)'::geometry
), changed AS (
    SELECT ST_SetValues(
        rast,
        1,
        ARRAY[
            ROW((SELECT geom FROM bar WHERE gid = 2), 2),
            ROW((SELECT geom FROM bar WHERE gid = 1), 1)
        ]::geomval[]
    ) AS rast
    FROM foo
)
SELECT
    y,
    array_agg(ST_Value(rast, 1, x, y) ORDER BY x) AS values
FROM changed
CROSS JOIN generate_series(1, 5) AS x
CROSS JOIN generate_series(1, 5) AS y
GROUP BY y
ORDER BY y;
```

Output

```
 y |          values
---+--------------------------
 1 | {NULL,NULL,NULL,NULL,NULL}
 2 | {NULL,2,2,2,NULL}
 3 | {NULL,2,1,2,NULL}
 4 | {NULL,2,2,2,NULL}
 5 | {NULL,NULL,NULL,NULL,NULL}
(5 rows)
```

See Also

ST_Value, ST_SetValue, ST_PixelAsPolygons

11.6.14 ST_DumpValues

ST_DumpValues — Get the values of the specified band as a 2-dimension array.

Synopsis

setof record **ST_DumpValues**(raster rast , integer[] nband=NULL , boolean exclude_nodata_value=true);
double precision[][] **ST_DumpValues**(raster rast , integer nband , boolean exclude_nodata_value=true);

Description

Get the values of the specified band as a 2-dimension array (first index is row, second is column). If `nband` is NULL or not provided, all raster bands are processed.

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0),  ↩
    1, '8BUI'::text, 1, 0), 2, '32BF'::text, 3, -9999), 3, '16BSI', 0, 0) AS rast
)
SELECT
    (ST_DumpValues(rast)).*
FROM foo;
```

Output

```
 nband |                      valarray
-------+-----------------------------------------------------
     1 | {{1,1,1},{1,1,1},{1,1,1}}
     2 | {{3,3,3},{3,3,3},{3,3,3}}
     3 | {{NULL,NULL,NULL},{NULL,NULL,NULL},{NULL,NULL,NULL}}
(3 rows)
```

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0),  ↩
    1, '8BUI'::text, 1, 0), 2, '32BF'::text, 3, -9999), 3, '16BSI', 0, 0) AS rast
)
SELECT
    (ST_DumpValues(rast, ARRAY[3, 1])).*
FROM foo;
```

Output

```
 nband |                      valarray
-------+-----------------------------------------------------
     3 | {{NULL,NULL,NULL},{NULL,NULL,NULL},{NULL,NULL,NULL}}
     1 | {{1,1,1},{1,1,1},{1,1,1}}
(2 rows)
```

Code

```
WITH foo AS (
    SELECT ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     1, 0), 1, 2, 5) AS rast
)
SELECT
    (ST_DumpValues(rast, 1))[2][1]
FROM foo;
```

Output

```
5
```

See Also

ST_Value, ST_SetValue, ST_SetValues

11.6.15 ST_PixelOfValue

ST_PixelOfValue — Get the columnx, rowy coordinates of the pixel whose value equals the search value.

Synopsis

setof record **ST_PixelOfValue**(raster rast , integer nband , double precision[] search , boolean exclude_nodata_value=true);
setof record **ST_PixelOfValue**(raster rast , double precision[] search , boolean exclude_nodata_value=true);
setof record **ST_PixelOfValue**(raster rast , integer nband , double precision search , boolean exclude_nodata_value=true);
setof record **ST_PixelOfValue**(raster rast , double precision search , boolean exclude_nodata_value=true);

Description

Get the columnx, rowy coordinates of the pixel whose value equals the search value. If no band is specified, then band 1 is assumed.

Availability: 2.1.0

Examples

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
            1, '8BUI', 1, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 255],
            [1, 1, 0, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT (ST_PixelOfValue(rast, 1, ARRAY[1, 255])).*
FROM source
```

Output

```
 val | x | y
-----+---+---
   1 | 1 | 2
   1 | 1 | 3
   1 | 1 | 4
   1 | 1 | 5
   1 | 2 | 1
   1 | 2 | 2
   1 | 2 | 4
   1 | 2 | 5
   1 | 3 | 1
   1 | 3 | 2
   1 | 3 | 3
   1 | 3 | 4
   1 | 4 | 1
   1 | 4 | 3
   1 | 4 | 4
   1 | 4 | 5
   1 | 5 | 1
   1 | 5 | 2
   1 | 5 | 3
 255 | 5 | 4
   1 | 5 | 5
```

11.7 Raster Editors

11.7.1 ST_SetGeoReference

ST_SetGeoReference — Set Georeference 6 georeference parameters in a single call. Numbers should be separated by white space. Accepts inputs in GDAL or ESRI format. Default is GDAL.

Synopsis

raster **ST_SetGeoReference**(raster rast, text georefcoords, text format=GDAL);
raster **ST_SetGeoReference**(raster rast, double precision upperleftx, double precision upperlefty, double precision scalex, double precision scaley, double precision skewx, double precision skewy);

Description

Set Georeference 6 georeference parameters in a single call. Accepts inputs in 'GDAL' or 'ESRI' format. Default is GDAL. If 6 coordinates are not provided will return null.

Difference between format representations is as follows:

`GDAL` uses the value order shown below.

Code

```
scalex skewy skewx scaley upperleftx upperlefty
```

`ESRI` uses the value order shown below.

Code

```
scalex skewy skewx scaley upperleftx + scalex*0.5 upperlefty + scaley*0.5
```



Note
If the raster has out-db bands, changing the georeference may result in incorrect access of the band's externally stored data.

Enhanced: 2.1.0 Addition of ST_SetGeoReference(raster, double precision, ...) variant

Examples

Code

```
WITH source AS (
    SELECT ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0) AS rast
), variants AS (
    SELECT 0 AS rid, rast FROM source
    UNION ALL
    SELECT 1, ST_SetGeoReference(rast, '10 0 0 -10 0.1 0.1', 'GDAL')
    FROM source
    UNION ALL
    SELECT 2, ST_SetGeoReference(rast, '10 0 0 -10 5.1 -4.9', 'ESRI')
    FROM source
    UNION ALL
    SELECT 3, ST_SetGeoReference(rast, 1, 1, 10, -10, 0.001, 0.001)
    FROM source
)
SELECT
    rid,
    round(ST_UpperLeftX(rast)::numeric, 3) AS upperleftx,
    round(ST_UpperLeftY(rast)::numeric, 3) AS upperlefty,
    ST_ScaleX(rast) AS scalex,
    ST_ScaleY(rast) AS scaley,
    ST_SkewX(rast) AS skewx,
    ST_SkewY(rast) AS skewy
FROM variants
ORDER BY rid;
```

Output

```
 rid | upperleftx | upperlefty | scalex | scaley | skewx | skewy
-----+------------+------------+--------+--------+-------+-------
   0 |      0.000 |      0.000 |      1 |     -1 |     0 |     0
   1 |      0.100 |      0.100 |     10 |    -10 |     0 |     0
   2 |      0.100 |      0.100 |     10 |    -10 |     0 |     0
   3 |      1.000 |      1.000 |     10 |    -10 | 0.001 | 0.001
(4 rows)
```

See Also

ST_GeoReference, ST_ScaleX, ST_ScaleY, ST_UpperLeftX, ST_UpperLeftY

11.7.2 ST_SetRotation

ST_SetRotation — Set the rotation of the raster in radian.

Synopsis

raster **ST_SetRotation**(raster rast, float8 rotation);

Description

Uniformly rotate the raster. Rotation is in radian. Refer to World File for more details.

Examples

Code

```
WITH variants AS (
    SELECT rid, 'rotated' AS kind, ST_SetRotation(rast, 15) AS rast
    FROM dummy_rast
    UNION ALL
    SELECT rid, 'original', rast
    FROM dummy_rast
)
SELECT
    rid,
    kind,
    round(ST_ScaleX(rast)::numeric, 4) AS scale_x,
    round(ST_ScaleY(rast)::numeric, 4) AS scale_y,
    round(ST_SkewX(rast)::numeric, 4) AS skew_x,
    round(ST_SkewY(rast)::numeric, 4) AS skew_y
FROM variants
ORDER BY rid, kind;
```

Output

```
 rid |   kind   | scale_x | scale_y | skew_x | skew_y
-----+----------+---------+---------+--------+--------
   1 | original |  2.0000 |  3.0000 | 0.0000 | 0.0000
   1 | rotated  | -1.5194 | -2.2791 | 1.9509 | 1.3006
   2 | original |  0.0500 | -0.0500 | 0.0000 | 0.0000
   2 | rotated  | -0.0380 | -0.0380 | 0.0325 | 0.0325
(4 rows)
```

See Also

ST_Rotation, ST_ScaleX, ST_ScaleY, ST_SkewX, ST_SkewY

11.7.3 ST_SetScale

ST_SetScale — Sets the X and Y size of pixels in units of coordinate reference system. Number units/pixel width/height.

Synopsis

raster **ST_SetScale**(raster rast, float8 xy);
raster **ST_SetScale**(raster rast, float8 x, float8 y);

Description

Sets the X and Y size of pixels in units of coordinate reference system. Number units/pixel width/height. If only one unit passed in, assumed X and Y are the same number.



Note
ST_SetScale is different from ST_Rescale in that ST_SetScale do not resample the raster to match the raster extent. It only changes the metadata (or georeference) of the raster to correct an originally mis-specified scaling. ST_Rescale results in a raster having different width and height computed to fit the geographic extent of the input raster. ST_SetScale do not modify the width, nor the height of the raster.

Changed: 2.0.0 In WKTRaster versions this was called ST_SetPixelSize. This was changed in 2.0.0.

Examples

Code

```
UPDATE dummy_rast
    SET rast = ST_SetScale(rast, 1.5)
WHERE rid = 2;

SELECT ST_ScaleX(rast) As pixx, ST_ScaleY(rast) As pixy, Box3D(rast) As newbox
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 pixx | pixy |                    newbox
------+------+-------------------------------------------------
  1.5 |  1.5 | BOX(3427927.75 5793244 0, 3427935.25 5793251.5 0)
```

Code

```
UPDATE dummy_rast
    SET rast = ST_SetScale(rast, 1.5, 0.55)
WHERE rid = 2;

SELECT ST_ScaleX(rast) As pixx, ST_ScaleY(rast) As pixy, Box3D(rast) As newbox
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 pixx | pixy |                   newbox
------+------+-----------------------------------------------
  1.5 | 0.55 | BOX(3427927.75 5793244 0,3427935.25 5793247 0)
```

See Also

ST_ScaleX, ST_ScaleY, Box3D

11.7.4 ST_SetSkew

ST_SetSkew — Sets the georeference X and Y skew (or rotation parameter). If only one is passed in, sets X and Y to the same value.

Synopsis

raster **ST_SetSkew**(raster rast, float8 skewxy);
raster **ST_SetSkew**(raster rast, float8 skewx, float8 skewy);

Description

Sets the georeference X and Y skew (or rotation parameter). If only one is passed in, sets X and Y to the same value. Refer to World File for more details.

Examples

This is the first example.

Code

```
UPDATE dummy_rast SET rast = ST_SetSkew(rast, 1, 2) WHERE rid = 1;
SELECT
    rid,
    format('(%s, %s)', skewx, skewy) AS skew,
    term,
    georef[idx] AS value
FROM (
    SELECT
        rid,
        ST_SkewX(rast) AS skewx,
        ST_SkewY(rast) AS skewy,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
    WHERE rid = 1
) AS refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
) AS parts(idx, term)
ORDER BY idx;
```

Output

```
 rid |  skew  |  term   |    value
-----+--------+---------+-------------
   1 | (1, 2) | scaleX  | 2.0000000000
   1 | (1, 2) | skewY   | 2.0000000000
   1 | (1, 2) | skewX   | 1.0000000000
   1 | (1, 2) | scaleY  | 3.0000000000
   1 | (1, 2) | originX | 0.5000000000
   1 | (1, 2) | originY | 0.5000000000
(6 rows)
```

This second example sets both values to the same number.

Code

```
UPDATE dummy_rast SET rast = ST_SetSkew(rast, 0) WHERE rid = 1;
SELECT
    rid,
    format('(%s, %s)', skewx, skewy) AS skew,
    term,
    georef[idx] AS value
FROM (
    SELECT
        rid,
        ST_SkewX(rast) AS skewx,
        ST_SkewY(rast) AS skewy,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
    WHERE rid = 1
) AS refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
```

```
) AS parts(idx, term)
ORDER BY idx;
```

Output

```
 rid |  skew  |  term   |    value
-----+--------+---------+-------------
   1 | (0, 0) | scaleX  | 2.0000000000
   1 | (0, 0) | skewY   | 0.0000000000
   1 | (0, 0) | skewX   | 0.0000000000
   1 | (0, 0) | scaleY  | 3.0000000000
   1 | (0, 0) | originX | 0.5000000000
   1 | (0, 0) | originY | 0.5000000000
(6 rows)
```

See Also

ST_GeoReference, ST_SetGeoReference, ST_SkewX, ST_SkewY

11.7.5 ST_SetSRID

ST_SetSRID — Sets the SRID of a raster to a particular integer srid defined in the spatial_ref_sys table.

Synopsis

raster **ST_SetSRID**(raster rast, integer srid);

Description

Sets the SRID on a raster to a particular integer value.



Note
This function does not transform the raster in any way - it simply sets meta data defining the spatial ref of the coordinate reference system that it's currently in. Useful for transformations later.

See Also

Section 4.5, ST_SRID

11.7.6 ST_SetUpperLeft

ST_SetUpperLeft — Sets the value of the upper left corner of the pixel of the raster to projected X and Y coordinates.

Synopsis

raster **ST_SetUpperLeft**(raster rast, double precision x, double precision y);

Description

Set the value of the upper left corner of raster to the projected X and Y coordinates

Examples

Code

```
SELECT ST_SetUpperLeft(rast, -71.01, 42.37)
FROM dummy_rast
WHERE rid = 2;
```

See Also

ST_UpperLeftX, ST_UpperLeftY

11.7.7 ST_Resample

ST_Resample — Resample a raster using a specified resampling algorithm, new dimensions, an arbitrary grid corner and a set of raster georeferencing attributes defined or borrowed from another raster.

Synopsis

raster **ST_Resample**(raster rast, integer width, integer height, double precision gridx=NULL, double precision gridy=NULL, double precision skewx=0, double precision skewy=0, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resample**(raster rast, double precision scalex=0, double precision scaley=0, double precision gridx=NULL, double precision gridy=NULL, double precision skewx=0, double precision skewy=0, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resample**(raster rast, raster ref, text algorithm=NearestNeighbor, double precision maxerr=0.125, boolean usescale=true);
raster **ST_Resample**(raster rast, raster ref, boolean usescale, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

Resample a raster using a specified resampling algorithm, new dimensions (width & height), a grid corner (gridx & gridy) and a set of raster georeferencing attributes (scalex, scaley, skewx & skewy) defined or borrowed from another raster. If using a reference raster, the two rasters must have the same SRID.

New pixel values are computed using one of the following resampling algorithms:

- NearestNeighbor (english or american spelling)
- Bilinear
- Cubic
- CubicSpline
- Lanczos
- Max
- Min

The default is NearestNeighbor which is the fastest but results in the worst interpolation.

A maxerror percent of 0.125 is used if no `maxerr` is specified.



Note
Refer to: GDAL Warp resampling methods for more details.

Availability: 2.0.0 Requires GDAL 1.6.1+

Enhanced: 3.4.0 max and min resampling options added

Examples

Code

```
SELECT
    ST_Width(orig) AS orig_width,
    ST_Width(reduce_100) AS new_width
FROM (
    SELECT
        rast AS orig,
        ST_Resample(rast, 100, 100) AS reduce_100
    FROM aerials.boston
    WHERE ST_Intersects(
        rast,
        ST_Transform(ST_MakeEnvelope(-71.128, 42.2392, -71.1277, 42.2397, 4326),
            26986)
    )
    LIMIT 1
) AS foo;
```

Output

```
 orig_width | new_width
------------+-------------
        200 |         100
```

See Also

ST_Rescale, ST_Resize, ST_Transform

11.7.8 ST_Rescale

ST_Rescale — Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.

Synopsis

raster **ST_Rescale**(raster rast, double precision scalexy, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Rescale**(raster rast, double precision scalex, double precision scaley, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using one of the following resampling algorithms:

- NearestNeighbor (english or american spelling)
- Bilinear
- Cubic
- CubicSpline
- Lanczos

- Max
- Min

The default is NearestNeighbor which is the fastest but results in the worst interpolation.

`scalex` and `scaley` define the new pixel size. scaley must often be negative to get well oriented raster.

When the new scalex or scaley is not a divisor of the raster width or height, the extent of the resulting raster is expanded to encompass the extent of the provided raster. If you want to be sure to retain exact input extent see ST_Resize

`maxerr` is the threshold for transformation approximation by the resampling algorithm (in pixel units). A default of 0.125 is used if no `maxerr` is specified, which is the same value used in GDAL gdalwarp utility. If set to zero, no approximation takes place.



Note
Refer to: GDAL Warp resampling methods for more details.



Note
ST_Rescale is different from ST_SetScale in that ST_SetScale do not resample the raster to match the raster extent. ST_SetScale only changes the metadata (or georeference) of the raster to correct an originally mis-specified scaling. ST_Rescale results in a raster having different width and height computed to fit the geographic extent of the input raster. ST_SetScale do not modify the width, nor the height of the raster.

Availability: 2.0.0 Requires GDAL 1.6.1+

Enhanced: 3.4.0 max and min resampling options added

Changed: 2.1.0 Works on rasters with no SRID

Examples

A simple example rescaling a raster from a pixel size of 0.001 degree to a pixel size of 0.0015 degree.

This example returns the original raster pixel size.

Code

```
SELECT ST_PixelWidth(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001, 0, 0,  ↩
    4269), '8BUI'::text, 1, 0)) width
```

Output

```
0.001
```

This example returns the pixel size of the rescaled raster.

Code

```
SELECT ST_PixelWidth(ST_Rescale(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001, ↩
     0, 0, 4269), '8BUI'::text, 1, 0), 0.0015)) width
```

Output

```
0.0015
```

See Also

ST_Resize, ST_Resample, ST_SetScale, ST_ScaleX, ST_ScaleY, ST_Transform

11.7.9 ST_Reskew

ST_Reskew — Resample a raster by adjusting only its skew (or rotation parameters). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.

Synopsis

raster **ST_Reskew**(raster rast, double precision skewxy, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Reskew**(raster rast, double precision skewx, double precision skewy, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

Resample a raster by adjusting only its skew (or rotation parameters). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. The default is NearestNeighbor which is the fastest but results in the worst interpolation.

`skewx` and `skewy` define the new skew.

The extent of the new raster will encompass the extent of the provided raster.

A maxerror percent of 0.125 if no `maxerr` is specified.



Note
Refer to: GDAL Warp resampling methods for more details.



Note
ST_Reskew is different from ST_SetSkew in that ST_SetSkew do not resample the raster to match the raster extent. ST_SetSkew only changes the metadata (or georeference) of the raster to correct an originally mis-specified skew. ST_Reskew results in a raster having different width and height computed to fit the geographic extent of the input raster. ST_SetSkew do not modify the width, nor the height of the raster.

Availability: 2.0.0 Requires GDAL 1.6.1+

Changed: 2.1.0 Works on rasters with no SRID

Examples

A simple example reskewing a raster from a skew of 0.0 to a skew of 0.0015.

This example uses the original non-rotated raster.

Code

```
SELECT ST_Rotation(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001, 0, 0, 4269), ↩
    '8BUI'::text, 1, 0));
```

Output

```
 st_rotation
-------------
           0
(1 row)
```

The reskewed raster rotation.

Code

```
SELECT ST_Rotation(ST_Reskew(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001, 0, ↩
    0, 4269), '8BUI'::text, 1, 0), 0.0015));
```

Output

```
    st_rotation
-------------------
 -0.982793723247329
(1 row)
```

See Also

ST_Resample, ST_Rescale, ST_SetSkew, ST_SetRotation, ST_SkewX, ST_SkewY, ST_Transform

11.7.10 ST_SnapToGrid

ST_SnapToGrid — Resample a raster by snapping it to a grid. New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.

Synopsis

raster **ST_SnapToGrid**(raster rast, double precision gridx, double precision gridy, text algorithm=NearestNeighbor, double precision maxerr=0.125, double precision scalex=DEFAULT 0, double precision scaley=DEFAULT 0);
raster **ST_SnapToGrid**(raster rast, double precision gridx, double precision gridy, double precision scalex, double precision scaley, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_SnapToGrid**(raster rast, double precision gridx, double precision gridy, double precision scalexy, text algorithm=NearestNeigh
double precision maxerr=0.125);

Description

Resample a raster by snapping it to a grid defined by an arbitrary pixel corner (gridx & gridy) and optionally a pixel size (scalex & scaley). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. The default is NearestNeighbor which is the fastest but results in the worst interpolation.

`gridx` and `gridy` define any arbitrary pixel corner of the new grid. This is not necessarily the upper left corner of the new raster and it does not have to be inside or on the edge of the new raster extent.

You can optionally define the pixel size of the new grid with `scalex` and `scaley`.

The extent of the new raster will encompass the extent of the provided raster.

A maxerror percent of 0.125 if no `maxerr` is specified.



Note
Refer to: GDAL Warp resampling methods for more details.



Note
Use ST_Resample if you need more control over the grid parameters.

Availability: 2.0.0 Requires GDAL 1.6.1+

Changed: 2.1.0 Works on rasters with no SRID

Examples

A simple example snapping a raster to a slightly different grid.

This example returns the upper-left X coordinate of the original raster.

Code

```
SELECT ST_UpperLeftX(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 0.001, -0.001, 0, 0, 4269), ↩
    '8BUI'::text, 1, 0));
```

Output

```
 st_upperleftx
---------------
             0
(1 row)
```

The upper left of raster after snapping.

Code

```
SELECT ST_UpperLeftX(ST_SnapToGrid(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 0.001,  ↩
   -0.001, 0, 0, 4269), '8BUI'::text, 1, 0), 0.0002, 0.0002));
```

Output

```
 st_upperleftx
---------------
       -0.0008
(1 row)
```

See Also

ST_Resample, ST_Rescale, ST_UpperLeftX, ST_UpperLeftY

11.7.11 ST_Resize

ST_Resize — Resize a raster to a new width/height

Synopsis

raster **ST_Resize**(raster rast, integer width, integer height, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resize**(raster rast, double precision percentwidth, double precision percentheight, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resize**(raster rast, text width, text height, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

Resize a raster to a new width/height. The new width/height can be specified in exact number of pixels or a percentage of the raster's width/height. The extent of the the new raster will be the same as the extent of the provided raster.

New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. The default is NearestNeighbor which is the fastest but results in the worst interpolation.

Variant 1 expects the actual width/height of the output raster.

Variant 2 expects decimal values between zero (0) and one (1) indicating the percentage of the input raster's width/height.

Variant 3 takes either the actual width/height of the output raster or a textual percentage ("20%") indicating the percentage of the input raster's width/height.

Availability: 2.1.0 Requires GDAL 1.6.1+

Examples

Code

```
WITH foo AS(
SELECT
    1 AS rid,
    ST_Resize(ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 1, -1, 0, 0, 0),
            1, '8BUI', 255, 0
        ),
    '50%', '500') AS rast
UNION ALL
SELECT
    2 AS rid,
    ST_Resize(ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 1, -1, 0, 0, 0),
            1, '8BUI', 255, 0
        ),
    500, 100) AS rast
UNION ALL
SELECT
    3 AS rid,
    ST_Resize(ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 1, -1, 0, 0, 0),
            1, '8BUI', 255, 0
        ),
    0.25, 0.9) AS rast
)
SELECT
    rid,
    ST_Width(rast) AS width,
    ST_Height(rast) AS height,
    ST_ScaleX(rast) AS scale_x,
    round(ST_ScaleY(rast)::numeric, 4) AS scale_y
FROM foo
ORDER BY rid
```

Output

```
 rid | width | height | scale_x | scale_y
-----+-------+--------+---------+---------
   1 |   500 |    500 |       2 | -2.0000
   2 |   500 |    100 |       2 | -10.0000
   3 |   250 |    900 |       4 | -1.1111
(3 rows)
```

See Also

ST_Resample, ST_Rescale, ST_Reskew, ST_SnapToGrid

11.7.12 ST_Transform

ST_Transform — Reprojects a raster in a known spatial reference system to another known spatial reference system using specified resampling algorithm. Options are NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos defaulting to NearestNeighbor.

Synopsis

raster **ST_Transform**(raster rast, integer srid, text algorithm=NearestNeighbor, double precision maxerr=0.125, double precision scalex, double precision scaley);
raster **ST_Transform**(raster rast, integer srid, double precision scalex, double precision scaley, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Transform**(raster rast, raster alignto, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

Reprojects a raster in a known spatial reference system to another known spatial reference system using specified pixel warping algorithm. Uses 'NearestNeighbor' if no algorithm is specified and maxerror percent of 0.125 if no maxerr is specified.

Algorithm options are: 'NearestNeighbor', 'Bilinear', 'Cubic', 'CubicSpline', and 'Lanczos'. Refer to: GDAL Warp resampling methods for more details.

ST_Transform is often confused with ST_SetSRID(). ST_Transform actually changes the coordinates of a raster (and resamples the pixel values) from one spatial reference system to another, while ST_SetSRID() simply changes the SRID identifier of the raster.

Unlike the other variants, Variant 3 requires a reference raster as `alignto`. The transformed raster will be transformed to the spatial reference system (SRID) of the reference raster and be aligned (ST_SameAlignment = TRUE) to the reference raster.



Note
If you find your transformation support is not working right, you may need to set the environment variable PROJSO to the projection library your PostGIS build is using. This just needs to have the name of the file. So for example on windows, you would in Control Panel -> System -> Environment Variables add a system variable called `PROJSO` and set it to `libproj.dll`. You'll have to restart your PostgreSQL service/daemon after this change.



Warning
When transforming a coverage of tiles, you almost always want to use a reference raster to insure same alignment and no gaps in your tiles as demonstrated in example: Variant 3.

Availability: 2.0.0 Requires GDAL 1.6.1+

Enhanced: 2.1.0 Addition of ST_Transform(rast, alignto) variant

Examples

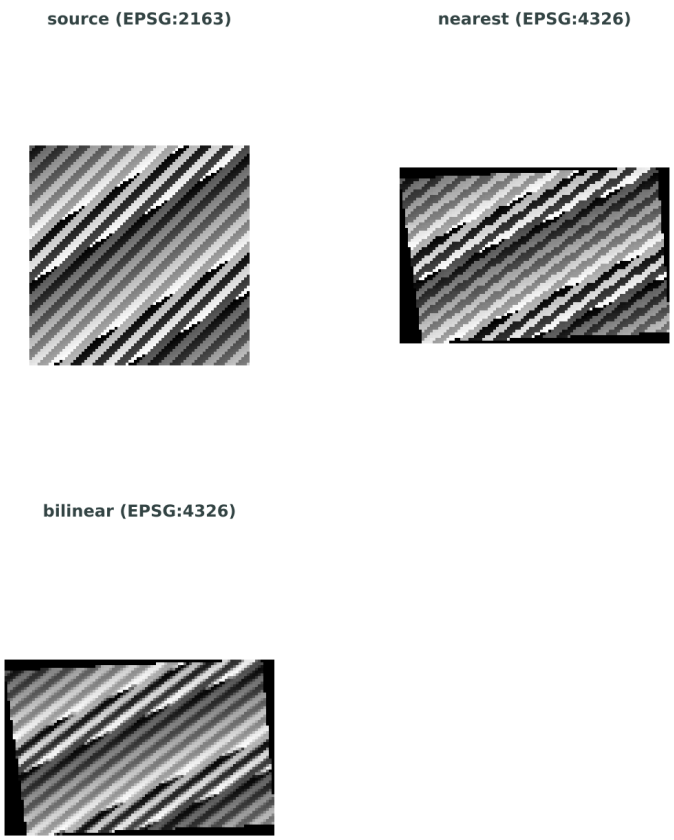
Compare a source raster with nearest-neighbor and bilinear reprojections. The diagonal bands make the resampling behavior visible: nearest-neighbor keeps hard pixel steps, while bilinear smooths values between pixels. All three figures are generated from this query at manual build time.

Code

```
WITH rows AS (
    SELECT
        y,
        array_agg(
```

```
            ((
                    x * 3 + y * 5
                    + CASE WHEN (x + y) % 8 < 4 THEN 80 ELSE 0 END
                ) % 255)::double precision
            ORDER BY x
        ) AS row_values
    FROM generate_series(1, 80) AS y
    CROSS JOIN generate_series(1, 80) AS x
    GROUP BY y
), source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(
                80, 80,
                -500000, 600000,
                2500, -2500,
                0, 0,
                2163
            ),
            1, '8BUI', 0, NULL
        ),
        1, 1, 1,
        array_agg(row_values ORDER BY y)::double precision[][]
    ) AS rast
    FROM rows
), transformed AS (
    SELECT
        rast,
        ST_Transform(rast, 4326, 'NearestNeighbor') AS nearest,
        ST_Transform(rast, 4326, 'Bilinear') AS bilinear
    FROM source
)
SELECT
    ST_AsPNG(rast) AS "source (EPSG:2163)",
    ST_AsPNG(nearest) AS "nearest (EPSG:4326)",
    ST_AsPNG(bilinear) AS "bilinear (EPSG:4326)"
FROM transformed;
```

Figure



Raster reprojection and resampling.



Massachusetts State Plane source



WGS 84 nearest-neighbor



WGS 84 bilinear

Variant 3.

The following shows the difference between using ST_Transform(raster, srid) and ST_Transform(raster, alignto)

Code

```
WITH foo AS (
    SELECT 0 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, -500000, 600000, 100, -100, 0, 0,
    2163), 1, '16BUI', 1, 0) AS rast UNION ALL
    SELECT 1, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499800, 600000, 100, -100, 0, 0, 2163),
    1, '16BUI', 2, 0) AS rast UNION ALL
    SELECT 2, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499600, 600000, 100, -100, 0, 0, 2163),
    1, '16BUI', 3, 0) AS rast UNION ALL

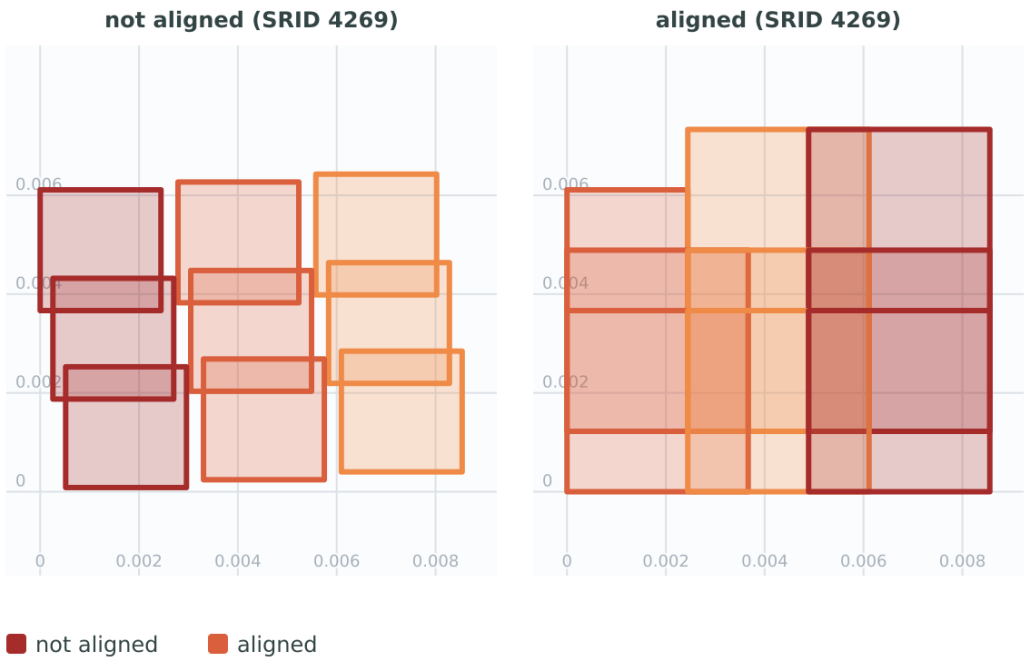
    SELECT 3, ST_AddBand(ST_MakeEmptyRaster(2, 2, -500000, 599800, 100, -100, 0, 0, 2163),
    1, '16BUI', 10, 0) AS rast UNION ALL
    SELECT 4, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499800, 599800, 100, -100, 0, 0, 2163),
    1, '16BUI', 20, 0) AS rast UNION ALL
    SELECT 5, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499600, 599800, 100, -100, 0, 0, 2163),
    1, '16BUI', 30, 0) AS rast UNION ALL

    SELECT 6, ST_AddBand(ST_MakeEmptyRaster(2, 2, -500000, 599600, 100, -100, 0, 0, 2163),
    1, '16BUI', 100, 0) AS rast UNION ALL
    SELECT 7, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499800, 599600, 100, -100, 0, 0, 2163),
    1, '16BUI', 200, 0) AS rast UNION ALL
    SELECT 8, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499600, 599600, 100, -100, 0, 0, 2163),
    1, '16BUI', 300, 0) AS rast
), bar AS (
    SELECT
        ST_Transform(rast, 4269) AS alignto
    FROM foo
    LIMIT 1
), baz AS (
    SELECT
        rid,
        rast,
        ST_Transform(rast, 4269) AS not_aligned,
        ST_Transform(rast, alignto) AS aligned
    FROM foo
    CROSS JOIN bar
)
SELECT
    ST_Collect(ST_ConvexHull(not_aligned) ORDER BY rid) AS not_aligned,
    ST_Collect(ST_ConvexHull(aligned) ORDER BY rid) AS aligned
FROM baz
```

Output

```
MULTIPOLYGON(((-107.007 50.2,-107.005 50.2,-107.005 50.198,-107.007 50.198,-107.007 50.2)),((-107.004 50.201,-107.002
    50.201,-107.002 50.198,-107.004 50.198,-107.004 50.201)),((-107.001 50.201,-106.999 50.201,-106.999
    50.198,-107.001 50.198,-107.001 50.201)),((-107.007 50.199,-107.004 50.199,-107.004 50.196,-107.007
    50.196,-107.007 50.199)),((-107.004 50.199,-107.001 50.199,-107.001 50.196,-107.004 50.196,-107.004 50.199))
    ,((-107.001 50.199,-106.999 50.199,-106.999 50.196,-107.001 50.196,-107.001 50.199)),((-107.006 50.197,-107.004
    50.197,-107.004 50.194,-107.006 50.194,-107.006 50.197)),((-107.004 50.197,-107.001 50.197,-107.001
    50.194,-107.004 50.194,-107.004 50.197)),((-107.001 50.197,-106.998 50.197,-106.998 50.195,-107.001
    50.195,-107.001 50.197))) | MULTIPOLYGON(((-107.007 50.2,-107.005 50.2,-107.005 50.198,-107.007 50.198,-107.007
    50.2)),((-107.005 50.202,-107.001 50.202,-107.001 50.198,-107.005 50.198,-107.005 50.202)),((-107.002
    50.202,-106.998 50.202,-106.998 50.198,-107.002 50.198,-107.002 50.202)),((-107.007 50.199,-107.003
    50.199,-107.003 50.195,-107.007 50.195,-107.007 50.199)),((-107.005 50.199,-107.001 50.199,-107.001
    50.195,-107.005 50.195,-107.005 50.199)),((-107.002 50.199,-106.998 50.199,-106.998 50.195,-107.002
    50.195,-107.002 50.199)),((-107.007 50.198,-107.003 50.198,-107.003 50.194,-107.007 50.194,-107.007 50.198))
    ,((-107.005 50.198,-107.001 50.198,-107.001 50.194,-107.005 50.194,-107.005 50.198)),((-107.002 50.198,-106.998
    50.198,-106.998 50.194,-107.002 50.194,-107.002 50.198)))
```

Figure



See Also

ST_Transform, ST_SetSRID

11.8 Raster Band Editors

11.8.1 ST_SetBandNoDataValue

ST_SetBandNoDataValue — Sets the value for the given band that represents no data. Band 1 is assumed if no band is specified. To mark a band as having no nodata value, set the nodata value = NULL.

Synopsis

raster **ST_SetBandNoDataValue**(raster rast, double precision nodatavalue);
raster **ST_SetBandNoDataValue**(raster rast, integer band, double precision nodatavalue, boolean forcechecking=false);

Description

Sets the value that represents no data for the band. Band 1 is assumed if not specified. This will affect results from ST_Polygon, ST_DumpAsPolygons, and the ST_PixelAs...() functions.

Examples

This example changes only the first band NODATA value.

Code

```
UPDATE dummy_rast
    SET rast = ST_SetBandNoDataValue(rast, 1, 254)
WHERE rid = 2;
```

This example changes the NODATA value for bands 1, 2, and 3.

Code

```
UPDATE dummy_rast
    SET rast =
        ST_SetBandNoDataValue(ST_SetBandNoDataValue(ST_SetBandNoDataValue(rast, 1, 254),
                2, 99),
                3, 108)
        WHERE rid = 2;
```

This example removes the NODATA value so that all pixels are considered by processing functions.

Code

```
UPDATE dummy_rast
    SET rast = ST_SetBandNoDataValue(rast, 1, NULL)
WHERE rid = 2;
```

See Also

ST_BandNoDataValue, ST_NumBands

11.8.2 ST_SetBandIsNoData

ST_SetBandIsNoData — Sets the isnodata flag of the band to TRUE.

Synopsis

raster **ST_SetBandIsNoData**(raster rast, integer band=1);

Description

Sets the isnodata flag for the band to true. Band 1 is assumed if not specified. This function should be called only when the flag is considered dirty. That is, when the result calling ST_BandIsNoData is different using TRUE as last argument and without using it

Availability: 2.0.0

Examples

This example creates a dummy table with one raster column. The raster has two bands and one pixel per band. In the first band, the NODATA value and pixel value are both 3. In the second band, the NODATA value is 13 and the pixel value is 4.

Code

```
create table dummy_rast (rid integer, rast raster);

insert into dummy_rast values(
1,
(
'01' -- little endian (uint8 ndr)
||
'0000' -- version (uint16 0)
||
'0200' -- nBands (uint16 0)
||
'17263529ED684A3F' -- scaleX (float64 0.000805965234044584)
||
'F9253529ED684ABF' -- scaleY (float64 -0.00080596523404458)
||
'1C9F33CE69E352C0' -- ipX (float64 -75.5533328537098)
||
'718F0E9A27A44840' -- ipY (float64 49.2824585505576)
||
'ED50EB853EC32B3F' -- skewX (float64 0.000211812383858707)
||
'7550EB853EC32B3F' -- skewY (float64 0.000211812383858704)
||
'E6100000' -- SRID (int32 4326)
||
'0100' -- width (uint16 1)
||
'0100' -- height (uint16 1)
||
'4' -- hasnodatavalue set to true, isnodata value set to false (when it should be true)
||
'2' -- first band type (4BUI)
||
'03' -- novalue==3
||
'03' -- pixel(0,0)==3 (same that nodata)
||
'0' -- hasnodatavalue set to false
||
'5' -- second band type (16BSI)
||
'0D00' -- novalue==13
||
'0400' -- pixel(0,0)==4
)::raster
);

select ST_BandIsNoData(rast, 1) from dummy_rast where rid = 1; -- Expected false
select ST_BandIsNoData(rast, 1, TRUE) from dummy_rast where rid = 1; -- Expected true
```

The `isnodata` flag is dirty, so set it to true.

Code

```
update dummy_rast set rast = ST_SetBandIsNoData(rast, 1) where rid = 1;


select ST_BandIsNoData(rast, 1) from dummy_rast where rid = 1; -- Expected true
```

See Also

ST_BandNoDataValue, ST_NumBands, ST_SetBandNoDataValue, ST_BandIsNoData

11.8.3 ST_SetBandPath

ST_SetBandPath — Update the external path and band number of an out-db band

Synopsis

raster **ST_SetBandPath**(raster rast, integer band, text outdbpath, integer outdbindex, boolean force=false);

Description

Updates an out-db band's external raster file path and external band number.



Note
If `force` is set to true, no tests are done to ensure compatibility (e.g. alignment, pixel support) between the external raster file and the PostGIS raster. This mode is intended for file system changes where the external raster resides.

Availability: 2.5.0

Examples

Code

```
WITH foo AS (
    SELECT ST_AddBand(
        NULL::raster,
        '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif',
        NULL::int[]
    ) AS rast
)
SELECT
    1 AS query,
    bandnum,
    regexp_replace(path, '^.*/', '') AS path,
    outdbbandnum
FROM ST_BandMetaData(
    (SELECT rast FROM foo),
    ARRAY[1, 3, 2]::int[]
) AS metadata
UNION ALL
SELECT
    2, bandnum,
    regexp_replace(path, '^.*/', ''),
    outdbbandnum
FROM ST_BandMetaData((
        SELECT
            ST_SetBandPath(rast,
                2,
                '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected2.tif' ↩
    ,
                1
```

```
            ) AS rast
        FROM foo
    ),
    ARRAY[1, 3, 2]::int[]
) AS metadata
ORDER BY 1, 2;
```

In the second result set, band 2 now points at `Projected2.tif` and uses out-db band 1.

Output

```
 query | bandnum | path           | outdbbandnum
-------+---------+----------------+--------------
     1 |       1 | Projected.tif  |            1
     1 |       2 | Projected.tif  |            2
     1 |       3 | Projected.tif  |            3
     2 |       1 | Projected.tif  |            1
     2 |       2 | Projected2.tif |            1
     2 |       3 | Projected.tif  |            3
```

See Also

ST_BandMetaData, ST_SetBandIndex

11.8.4 ST_SetBandIndex

ST_SetBandIndex — Update the external band number of an out-db band

Synopsis

raster **ST_SetBandIndex**(raster rast, integer band, integer outdbindex, boolean force=false);

Description

Updates an out-db band's external band number. This does not touch the external raster file associated with the out-db band



Note
If `force` is set to true, no tests are done to ensure compatibility (e.g. alignment, pixel support) between the external raster file and the PostGIS raster. This mode is intended for where bands are moved around in the external raster file.



Note
Internally, this method replaces the PostGIS raster's band at index `band` with a new band instead of updating the existing path information.

Availability: 2.5.0

Examples

Code

```
WITH foo AS (
    SELECT ST_AddBand(
        NULL::raster,
        '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif',
        NULL::int[]
    ) AS rast
)
SELECT
    1 AS query,
    bandnum,
    regexp_replace(path, '^.*/', '') AS path,
    outdbbandnum
FROM ST_BandMetaData(
    (SELECT rast FROM foo),
    ARRAY[1, 3, 2]::int[]
) AS metadata
UNION ALL
SELECT
    2, bandnum,
    regexp_replace(path, '^.*/', ''),
    outdbbandnum
FROM ST_BandMetaData((
        SELECT
            ST_SetBandIndex(rast,
                2,
                1
            ) AS rast
        FROM foo
    ),
    ARRAY[1, 3, 2]::int[]
) AS metadata
ORDER BY 1, 2;
```

In the second result set, band 2 still points at the same file but now reads external band 1.

Output

```
 query | bandnum | path          | outdbbandnum
-------+---------+---------------+--------------
     1 |       1 | Projected.tif |            1
     1 |       2 | Projected.tif |            2
     1 |       3 | Projected.tif |            3
     2 |       1 | Projected.tif |            1
     2 |       2 | Projected.tif |            1
     2 |       3 | Projected.tif |            3
```

See Also

ST_BandMetaData, ST_SetBandPath

11.9 Raster Band Statistics and Analytics

11.9.1 ST_Count

ST_Count — Returns the number of pixels in a given band of a raster or raster coverage. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the nodata value.

Synopsis

bigint **ST_Count**(raster rast, integer nband=1, boolean exclude_nodata_value=true);
bigint **ST_Count**(raster rast, boolean exclude_nodata_value);

Description

Returns the number of pixels in a given band of a raster or raster coverage. If no band is specified `nband` defaults to 1.



Note
If `exclude_nodata_value` is set to true, will only count pixels with value not equal to the `nodata` value of the raster. Set `exclude_nodata_value` to false to get count all pixels

Changed: 3.1.0 - The ST_Count(rastertable, rastercolumn, ...) variants removed. Use ST_CountAgg instead.

Availability: 2.0.0

Examples

Example will count all pixels not 249 and one will count all pixels.

Code

```
SELECT rid, ST_Count(ST_SetBandNoDataValue(rast, 249)) As exclude_nodata,
        ST_Count(ST_SetBandNoDataValue(rast, 249), false) As include_nodata
    FROM dummy_rast WHERE rid=2;
```

Output

```
rid | exclude_nodata | include_nodata
-----+----------------+----------------
   2 |             23 |             25
```

See Also

ST_CountAgg, ST_SummaryStats, ST_SetBandNoDataValue

11.9.2 ST_CountAgg

ST_CountAgg — Aggregate. Returns the number of pixels in a given band of a set of rasters. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the NODATA value.

Synopsis

bigint **ST_CountAgg**(raster rast, integer nband, boolean exclude_nodata_value, double precision sample_percent);
bigint **ST_CountAgg**(raster rast, integer nband, boolean exclude_nodata_value);
bigint **ST_CountAgg**(raster rast, boolean exclude_nodata_value);

Description

Returns the number of pixels in a given band of a set of rasters. If no band is specified `nband` defaults to 1.

If `exclude_nodata_value` is set to true, will only count pixels with value not equal to the `NODATA` value of the raster. Set `exclude_nodata_value` to false to get count all pixels

By default will sample all pixels. To get faster response, set `sample_percent` to value between zero (0) and one (1)

Availability: 2.2.0

Examples

Code

```
WITH foo AS (
    SELECT
        rast.rast
    FROM (
        SELECT ST_SetValue(ST_SetValue(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(10, 10, 10, ↩
     10, 2, 2, 0, 0, 0),
                        1, '64BF', 0, 0
                    ),
                    1, 1, 1, -10
                ),
                1, 5, 4, 0
            ),
            1, 5, 5, 3.14159
        ) AS rast
    ) AS rast
    FULL JOIN (
        SELECT generate_series(1, 10) AS id
    ) AS id
        ON 1 = 1
)
SELECT
    ST_CountAgg(rast, 1, TRUE) AS non_nodata_pixels
FROM foo;
```

Output

```
 non_nodata_pixels
-------------------
                20
(1 row)
```

See Also

ST_Count, ST_SummaryStats, ST_SetBandNoDataValue

11.9.3 ST_Histogram

ST_Histogram — Returns a set of record summarizing a raster or raster coverage data distribution separate bin ranges. Number of bins are autocomputed if not specified.

Synopsis

SETOF record **ST_Histogram**(raster rast, integer nband=1, boolean exclude_nodata_value=true, integer bins=autocomputed, double precision[] width=NULL, boolean right=false);
SETOF record **ST_Histogram**(raster rast, integer nband, integer bins, double precision[] width=NULL, boolean right=false);
SETOF record **ST_Histogram**(raster rast, integer nband, boolean exclude_nodata_value, integer bins, boolean right);
SETOF record **ST_Histogram**(raster rast, integer nband, integer bins, boolean right);

Description

Returns set of records consisting of min, max, count, percent for a given raster band for each bin. If no band is specified `nband` defaults to 1.



Note
By default only considers pixel values not equal to the `nodata` value . Set `exclude_nodata_value` to false to get count all pixels.

`width` width: an array indicating the width of each category/bin. If the number of bins is greater than the number of widths, the widths are repeated.

Example: 9 bins, widths are [a, b, c] will have the output be [a, b, c, a, b, c, a, b, c]

`bins` Number of breakouts -- this is the number of records you'll get back from the function if specified. If not specified then the number of breakouts is autocomputed.

`right` compute the histogram from the right rather than from the left (default). This changes the criteria for evaluating a value x from [a, b) to (a, b]

Changed: 3.1.0 Removed ST_Histogram(table_name, column_name) variant.

Availability: 2.0.0

Examples

Single raster tile - compute histograms for bands 1, 2, 3 and autocompute bins.

Code

```
SELECT band, (stats).*
FROM (SELECT rid, band, ST_Histogram(rast, band) As stats
    FROM dummy_rast CROSS JOIN generate_series(1, 3) As band
     WHERE rid=2) As foo;
```

Output

```
 band |  min   |  max   | count | percent
------+--------+--------+-------+---------
    1 |   249  |   250  |     2 |    0.08
    1 |   250  |   251  |     2 |    0.08
    1 |   251  |   252  |     1 |    0.04
    1 |   252  |   253  |     2 |    0.08
    1 |   253  |   254  |    18 |    0.72
    2 |    78  | 113.2  |    11 |    0.44
    2 | 113.2  | 148.4  |     4 |    0.16
    2 | 148.4  | 183.6  |     4 |    0.16
    2 | 183.6  | 218.8  |     1 |    0.04
    2 | 218.8  |   254  |     5 |     0.2
    3 |    62  | 100.4  |    11 |    0.44
    3 | 100.4  | 138.8  |     5 |     0.2
    3 | 138.8  | 177.2  |     4 |    0.16
    3 | 177.2  | 215.6  |     1 |    0.04
    3 | 215.6  |   254  |     4 |    0.16
```

Just band 2 but for 6 bins.

Code

```
SELECT (stats).*
FROM (SELECT rid, ST_Histogram(rast, 2, 6) As stats
    FROM dummy_rast
     WHERE rid=2) As foo;
```

Output

```
    min     |    max     | count | percent
------------+------------+-------+---------
         78 | 107.333333 |     9 |    0.36
 107.333333 | 136.666667 |     6 |    0.24
 136.666667 |        166 |     0 |       0
        166 | 195.333333 |     4 |    0.16
 195.333333 | 224.666667 |     1 |    0.04
 224.666667 |        254 |     5 |     0.2
(6 rows)
```

Same as previous but we explicitly control the pixel value range of each bin.

Code

```
SELECT (stats).*
FROM (SELECT rid, ST_Histogram(rast, 2, 6, ARRAY[0.5, 1, 4, 100, 5]) As stats
    FROM dummy_rast
     WHERE rid=2) As foo;
```

Output

```
  min   |  max   | count | percent
--------+--------+-------+----------
    78  |  78.5  |     1 |     0.08
  78.5  |  79.5  |     1 |     0.04
  79.5  |  83.5  |     0 |        0
  83.5  | 183.5  |    17 |   0.0068
 183.5  | 188.5  |     0 |        0
 188.5  |   254  |     6 | 0.003664
(6 rows)
```

See Also

ST_Count, ST_SummaryStats, ST_SummaryStatsAgg

11.9.4 ST_Quantile

ST_Quantile — Compute quantiles for a raster or raster table coverage in the context of the sample or population. Thus, a value could be examined to be at the raster's 25%, 50%, 75% percentile.

Synopsis

SETOF record **ST_Quantile**(raster rast, integer nband=1, boolean exclude_nodata_value=true, double precision[] quantiles=NULL);
SETOF record **ST_Quantile**(raster rast, double precision[] quantiles);
SETOF record **ST_Quantile**(raster rast, integer nband, double precision[] quantiles);
double precision **ST_Quantile**(raster rast, double precision quantile);
double precision **ST_Quantile**(raster rast, boolean exclude_nodata_value, double precision quantile=NULL);
double precision **ST_Quantile**(raster rast, integer nband, double precision quantile);
double precision **ST_Quantile**(raster rast, integer nband, boolean exclude_nodata_value, double precision quantile);
double precision **ST_Quantile**(raster rast, integer nband, double precision quantile);

Description

Compute quantiles for a raster or raster table coverage in the context of the sample or population. Thus, a value could be examined to be at the raster's 25%, 50%, 75% percentile.



Note
If `exclude_nodata_value` is set to false, will also count pixels with no data.

Changed: 3.1.0 Removed ST_Quantile(table_name, column_name) variant.

Availability: 2.0.0

Examples

Example will consider only pixels of band 1 that are not 249 and in named quantiles.

Code

```
UPDATE dummy_rast SET rast = ST_SetBandNoDataValue(rast, 249) WHERE rid=2;

SELECT (pvq).*
FROM (SELECT ST_Quantile(rast, ARRAY[0.25, 0.75]) As pvq
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvq).quantile;
```

Output

```
 quantile | value
----------+-------
     0.25 |   253
     0.75 |   254
```

Code

```
SELECT ST_Quantile(rast, 0.75) As value
FROM dummy_rast WHERE rid=2;
```

Output

```
254
```

Real live example. Quantile of all pixels in band 2 intersecting a geometry.

Code

```
SELECT rid, (ST_Quantile(rast, 2)).* As pvc
    FROM o_4_boston
        WHERE ST_Intersects(
            rast,
            ST_GeomFromText('POLYGON((224486 892151,224486 892200,224706 892200,224706 ↩
    892151,224486 892151))', 26986)
            )
ORDER BY value, quantile, rid
;
```

Output

```
 rid | quantile | value
-----+----------+-------
   1 |        0 |     0
   2 |        0 |     0
  14 |        0 |     1
  15 |        0 |     2
  14 |     0.25 |    37
   1 |     0.25 |    42
  15 |     0.25 |    47
   2 |     0.25 |    50
  14 |      0.5 |    56
   1 |      0.5 |    64
  15 |      0.5 |    66
   2 |      0.5 |    77
  14 |     0.75 |    81
  15 |     0.75 |    87
   1 |     0.75 |    94
   2 |     0.75 |   106
  14 |        1 |   199
   1 |        1 |   244
   2 |        1 |   255
  15 |        1 |   255
```

See Also

ST_Count, ST_SummaryStats, ST_SummaryStatsAgg, ST_SetBandNoDataValue

11.9.5 ST_SummaryStats

ST_SummaryStats — Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.

Synopsis

summarystats **ST_SummaryStats**(raster rast, boolean exclude_nodata_value);
summarystats **ST_SummaryStats**(raster rast, integer nband, boolean exclude_nodata_value);

Description

Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. If no band is specified `nband` defaults to 1.



Note
By default only considers pixel values not equal to the `nodata` value. Set `exclude_nodata_value` to false to get count of all pixels.



Note
By default will sample all pixels. To get faster response, set `sample_percent` to lower than 1

Changed: 3.1.0 ST_SummaryStats(rastertable, rastercolumn, ...) variants are removed. Use ST_SummaryStatsAgg instead.

Availability: 2.0.0

Examples

Single raster tile.

Code

```
SELECT rid, band, (stats).*
FROM (SELECT rid, band, ST_SummaryStats(rast, band) As stats
    FROM dummy_rast CROSS JOIN generate_series(1, 3) As band
     WHERE rid=2) As foo;
```

Output

```
 rid | band | count | sum  |    mean    |  stddev   | min | max
-----+------+-------+------+------------+-----------+-----+-----
   2 |    1 |    23 | 5821 | 253.086957 |  1.248061 | 250 | 254
   2 |    2 |    25 | 3682 |     147.28 | 59.862188 |  78 | 254
   2 |    3 |    25 | 3290 |      131.6 | 61.647384 |  62 | 254
```

Summarize pixels that intersect buildings of interest.

This example selects the target buildings, clips band 2 of the raster tiles to each building boundary, computes statistics for each clipped region, and then summarizes those statistics by building. It took 574ms on PostGIS Windows 64-bit with all of Boston Buildings and aerial Tiles (tiles each 150x150 pixels ~ 134,000 tiles), ~102,000 building records.

Code

```
WITH
-- our features of interest
feat AS (
    SELECT gid AS building_id, geom_26986 AS geom
    FROM buildings AS b
    WHERE gid IN (100, 103, 150)
),
-- clip band 2 of raster tiles to boundaries of builds
-- then get stats for these clipped regions
b_stats AS (
    SELECT building_id, (stats).*
    FROM (
        SELECT
            building_id,
            ST_SummaryStats(ST_Clip(rast, 2, geom)) AS stats
        FROM aerials.boston
        INNER JOIN feat
            ON ST_Intersects(feat.geom, rast)
    ) AS foo
)
-- finally summarize stats
SELECT
    building_id,
    SUM(count) AS num_pixels,
    MIN(min) AS min_pval,
    MAX(max) AS max_pval,
    SUM(mean * count) / SUM(count) AS avg_pval
FROM b_stats
WHERE count > 0
GROUP BY building_id
ORDER BY building_id;
```

Output

```
 building_id | num_pixels | min_pval | max_pval |     avg_pval
-------------+------------+----------+----------+------------------
         100 |       1090 |        1 |      255 | 61.0697247706422
         103 |        655 |        7 |      182 | 70.5038167938931
         150 |        895 |        2 |      252 | 185.642458100559
```

Raster coverage.

This example returns statistics for each band.

Code

```
SELECT band, (stats).*
FROM (SELECT band, ST_SummaryStats('o_4_boston', 'rast', band) As stats
    FROM generate_series(1, 3) As band) As foo;
```

Output

```
 band |  count  |  sum   |       mean       |      stddev      | min | max
------+---------+--------+------------------+------------------+-----+-----
    1 | 8450000 | 725799 | 82.7064349112426 | 45.6800222638537 |   0 | 255
    2 | 8450000 | 700487 | 81.4197705325444 | 44.2161184161765 |   0 | 255
    3 | 8450000 | 575943 |  74.682739408284 | 44.2143885481407 |   0 | 255
```

For a table, sampling less than 100% can improve speed. Here the sampling is set to 25%.

Code

```
SELECT band, (stats).*
FROM (SELECT band, ST_SummaryStats('o_4_boston', 'rast', band, true, 0.25) As stats
    FROM generate_series(1, 3) As band) As foo;
```

Output

```
 band |  count  |  sum   |       mean       |      stddev      | min | max
------+---------+--------+------------------+------------------+-----+-----
    1 | 2112500 | 180686 | 82.6890480473373 | 45.6961043857248 |   0 | 255
    2 | 2112500 | 174571 |  81.448503668639 | 44.2252623171821 |   0 | 255
    3 | 2112500 | 144364 | 74.6765884023669 | 44.2014869384578 |   0 | 255
```

See Also

summarystats, ST_SummaryStatsAgg, ST_Count, ST_Clip

11.9.6 ST_SummaryStatsAgg

ST_SummaryStatsAgg — Aggregate. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a set of raster. Band 1 is assumed if no band is specified.

Synopsis

summarystats **ST_SummaryStatsAgg**(raster rast, integer nband, boolean exclude_nodata_value, double precision sample_percent);
summarystats **ST_SummaryStatsAgg**(raster rast, boolean exclude_nodata_value, double precision sample_percent);
summarystats **ST_SummaryStatsAgg**(raster rast, integer nband, boolean exclude_nodata_value);

Description

Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. If no band is specified `nband` defaults to 1.



Note
By default only considers pixel values not equal to the `NODATA` value. Set `exclude_nodata_value` to False to get count of all pixels.



Note
By default will sample all pixels. To get faster response, set `sample_percent` to value between 0 and 1

Availability: 2.2.0

Examples

Code

```
WITH foo AS (
    SELECT
        rast.rast
    FROM (
        SELECT ST_SetValue(ST_SetValue(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(10, 10, 10, ↩
     10, 2, 2, 0, 0, 0),
                            1, '64BF', 0, 0
                        ),
                        1, 1, 1, -10
                    ),
                    1, 5, 4, 0
                ),
                1, 5, 5, 3.14159
        ) AS rast
    ) AS rast
    FULL JOIN (
        SELECT generate_series(1, 10) AS id
    ) AS id
        ON 1 = 1
)
SELECT
    (stats).count AS count,
    round((stats).sum::numeric, 3) AS sum,
    round((stats).mean::numeric, 3) AS mean,
    round((stats).stddev::numeric, 3) AS stddev,
    round((stats).min::numeric, 3) AS min,
    round((stats).max::numeric, 3) AS max
FROM (
    SELECT
        ST_SummaryStatsAgg(rast, 1, TRUE, 1) AS stats
    FROM foo
) bar;
```

Output

```
 count |   sum    |  mean  | stddev |   min   |  max
-------+----------+--------+--------+---------+-------
    20 | -68.584 | -3.429 |  6.571 | -10.000 | 3.142
(1 row)
```

See Also

summarystats, ST_SummaryStats, ST_Count, ST_Clip

11.9.7 ST_ValueCount

ST_ValueCount — Returns a set of records containing a pixel band value and count of the number of pixels in a given band of a raster (or a raster coverage) that have a given set of values. If no band is specified defaults to band 1. By default nodata value pixels are not counted. and all other values in the pixel are output and pixel band values are rounded to the nearest integer.

Synopsis

SETOF record **ST_ValueCount**(raster rast, integer nband=1, boolean exclude_nodata_value=true, double precision[] searchvalues=NULL, double precision roundto=0, double precision OUT value, integer OUT count);
SETOF record **ST_ValueCount**(raster rast, integer nband, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);

SETOF record **ST_ValueCount**(raster rast, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
bigint **ST_ValueCount**(raster rast, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(raster rast, integer nband, boolean exclude_nodata_value, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(raster rast, integer nband, double precision searchvalue, double precision roundto=0);
SETOF record **ST_ValueCount**(text rastertable, text rastercolumn, integer nband=1, boolean exclude_nodata_value=true, double precision[] searchvalues=NULL, double precision roundto=0, double precision OUT value, integer OUT count);
SETOF record **ST_ValueCount**(text rastertable, text rastercolumn, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
SETOF record **ST_ValueCount**(text rastertable, text rastercolumn, integer nband, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
bigint**ST_ValueCount**(text rastertable, text rastercolumn, integer nband, boolean exclude_nodata_value, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(text rastertable, text rastercolumn, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(text rastertable, text rastercolumn, integer nband, double precision searchvalue, double precision roundto=0);

Description

Returns a set of records with columns `value count` which contain the pixel band value and count of pixels in the raster tile or raster coverage of selected band.

If no band is specified `nband` defaults to 1. If no `searchvalues` are specified, will return all pixel values found in the raster or raster coverage. If one searchvalue is given, will return an integer instead of records denoting the count of pixels having that pixel band value



Note
If `exclude_nodata_value` is set to false, will also count pixels with no data.

Availability: 2.0.0

Examples

Example will count only pixels of band 1 that are not 249.

Code

```
UPDATE dummy_rast SET rast = ST_SetBandNoDataValue(rast, 249) WHERE rid=2;

SELECT (pvc).*
FROM (SELECT ST_ValueCount(rast) As pvc
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvc).value;
```

Output

```
 value | count
-------+-------
   250 |     2
   251 |     1
   252 |     2
   253 |     6
   254 |    12
```

Example will coount all pixels of band 1 including 249.

Code

```
SELECT (pvc).*
FROM (SELECT ST_ValueCount(rast, 1, false) As pvc
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvc).value;
```

Output

```
 value | count
-------+-------
   249 |     2
   250 |     2
   251 |     1
   252 |     2
   253 |     6
   254 |    12
```

Example will count only non-nodata value pixels of band 2.

Code

```
SELECT (pvc).*
FROM (SELECT ST_ValueCount(rast, 2) As pvc
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvc).value;
```

Output

```
 value | count
-------+-------
    78 |     1
    79 |     1
    88 |     1
    89 |     1
    96 |     1
    97 |     1
    98 |     1
    99 |     2
   112 |     2
:
```

Real live example. Count all the pixels in an aerial raster tile band 2 intersecting a geometry. And return only the pixel band values that have a count > 500.

Code

```
SELECT (pvc).value, SUM((pvc).count) As total
FROM (SELECT ST_ValueCount(rast, 2) As pvc
    FROM o_4_boston
        WHERE ST_Intersects(
            rast,
            ST_GeomFromText('POLYGON((224486 892151,224486 892200,224706 892200,224706 ↩
    892151,224486 892151))', 26986)
            )
        ) As foo
    GROUP BY (pvc).value
    HAVING SUM((pvc).count) > 500
    ORDER BY (pvc).value;
```

Output

```
 value | total
-------+-------
    51 | 502
    54 | 521
```

Just return count of pixels in each raster tile that have value of 100 of tiles that intersect a specific geometry.

Code

```
SELECT rid, ST_ValueCount(rast, 2, 100) As count
    FROM o_4_boston
        WHERE ST_Intersects(
            rast,
            ST_GeomFromText('POLYGON((224486 892151,224486 892200,224706 892200,224706 ↩
    892151,224486 892151))', 26986)
             ) ;
```

Output

```
 rid | count
-----+-------
   1 |    56
   2 |    95
  14 |    37
  15 |    64
```

See Also

ST_Count, ST_SetBandNoDataValue

11.10 Raster Inputs

11.10.1 ST_RastFromWKB

ST_RastFromWKB — Return a raster value from a Well-Known Binary (WKB) raster.

Synopsis

raster **ST_RastFromWKB**(bytea wkb);

Description

Given a Well-Known Binary (WKB) raster, return a raster.

Availability: 2.5.0

Examples

Code

```
SELECT (ST_MetaData(ST_RastFromWKB('\001\000\000\000\000\000\000\000\000\000\000\000@ ↩
    \000\000\000\000\000\000\010@ ↩
    \000\000\000\000\000\000\340?\000\000\000\000\000\000\340?\000\000\000\000\000\000\000\000\000\0
    '::bytea
    )
)).* AS metadata;
```

Output

```
 upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands
------------+------------+-------+--------+--------+--------+-------+-------+------+----------
        0.5 |        0.5 |    10 |     20 |      2 |      3 |     0 |     0 |   10 |        0
```

See Also

ST_MetaData, ST_RastFromHexWKB, ST_AsBinary/ST_AsWKB, ST_AsHexWKB

11.10.2 ST_RastFromHexWKB

ST_RastFromHexWKB — Return a raster value from a Hex representation of Well-Known Binary (WKB) raster.

Synopsis

raster **ST_RastFromHexWKB**(text wkb);

Description

Given a Well-Known Binary (WKB) raster in Hex representation, return a raster.

Availability: 2.5.0

Examples

Code

```
SELECT (ST_MetaData(ST_RastFromHexWKB(' ↩
    01000000000000000000000040000000000000084000000000000000 ↩
    E03F000000000000E03F0000000000000000000000000000000000A0000000A001400'
    )
)).* AS metadata;
```

Output

```
 upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands
------------+------------+-------+--------+--------+--------+-------+-------+------+----------
        0.5 |        0.5 |    10 |     20 |      2 |      3 |     0 |     0 |   10 |        0
```

See Also

ST_MetaData, ST_RastFromWKB, ST_AsBinary/ST_AsWKB, ST_AsHexWKB

11.11 Raster Outputs

11.11.1 ST_AsBinary/ST_AsWKB

ST_AsBinary/ST_AsWKB — Return the Well-Known Binary (WKB) representation of the raster.

Synopsis

bytea **ST_AsBinary**(raster rast, boolean outasin=FALSE);
bytea **ST_AsWKB**(raster rast, boolean outasin=FALSE);

Description

Returns the Binary representation of the raster. If `outasin` is TRUE, out-db bands are treated as in-db. Refer to raster/doc/RFC2-WellKnownBinaryFormat located in the PostGIS source folder for details of the representation.

This is useful in binary cursors to pull data out of the database without converting it to a string representation.



Note
By default, WKB output contains the external file path for out-db bands. If the client does not have access to the raster file underlying an out-db band, set `outasin` to TRUE.

Enhanced: 2.1.0 Addition of `outasin`

Enhanced: 2.5.0 Addition of `ST_AsWKB`

Examples

Code

```
SELECT ST_AsBinary(rast) AS rastbin FROM dummy_rast WHERE rid=1;
```

Output

```
\001\000\000\000\000\000\000\000\000\000\000@\000\000\000\000\000\000\010@ ↩
     \000\000\000\000\000\000\340?\000\000\000\000\000\000\340?\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\0
```

See Also

ST_RastFromWKB, ST_AsHexWKB

11.11.2 ST_AsHexWKB

ST_AsHexWKB — Return the Well-Known Binary (WKB) in Hex representation of the raster.

Synopsis

bytea **ST_AsHexWKB**(raster rast, boolean outasin=FALSE);

Description

Returns the Binary representation in Hex representation of the raster. If `outasin` is TRUE, out-db bands are treated as in-db. Refer to raster/doc/RFC2-WellKnownBinaryFormat located in the PostGIS source folder for details of the representation.



Note
By default, Hex WKB output contains the external file path for out-db bands. If the client does not have access to the raster file underlying an out-db band, set `outasin` to TRUE.

Availability: 2.5.0

Examples

Code

```
SELECT ST_AsHexWKB(rast) AS rastbin FROM dummy_rast WHERE rid=1;
```

Output

```
01000000000000000000000040000000000000084000000000000000 ↩
    E03F000000000000E03F000000000000000000000000000000000A0000000A001400
```

See Also

ST_RastFromHexWKB, ST_AsBinary/ST_AsWKB

11.11.3 ST_AsGDALRaster

ST_AsGDALRaster — Return the raster tile in the designated GDAL Raster format. Raster formats are one of those supported by your compiled library. Use ST_GDALDrivers() to get a list of formats supported by your library.

Synopsis

bytea **ST_AsGDALRaster**(raster rast, text format, text[] options=NULL, integer srid=sameassource);

Description

Returns the raster tile in the designated format. Arguments are itemized below:

- `format` format to output. This is dependent on the drivers compiled in your libgdal library. Generally available are 'JPEG', 'GTIff', 'PNG'. Use ST_GDALDrivers to get a list of formats supported by your library.
- `options` text array of GDAL options. Valid options are dependent on the format. Refer to GDAL Raster format options for more details.
- `srid` the spatial reference identifier to embed in the output raster

Availability: 2.0.0 - requires GDAL >= 1.6.0.

Examples

Export multiple raster tiles as a single JPEG image with 50 percent quality.

Code

```
SELECT ST_AsGDALRaster(ST_Union(rast), 'JPEG', ARRAY['QUALITY=50']) As rastjpg
FROM dummy_rast
WHERE rast && ST_MakeEnvelope(10, 10, 11, 11);
```

One way to export raster into another format is using PostgreSQL large object export functions. The following repeats the JPEG example and exports the result. Note that this requires superuser access because it uses server-side large object functions. It will also export to path on server network. If you need export locally, use the psql equivalent lo_ functions which export to the local file system instead of the server file system.

Code

```
DROP TABLE IF EXISTS tmp_out ;

CREATE TABLE tmp_out AS
SELECT lo_from_bytea(
       0,
       ST_AsGDALRaster(ST_Union(rast), 'JPEG', ARRAY['QUALITY=50'])
        ) AS loid
  FROM dummy_rast
WHERE rast && ST_MakeEnvelope(10, 10, 11, 11);

SELECT lo_export(loid, '/tmp/dummy.jpg')
   FROM tmp_out;

SELECT lo_unlink(loid)
  FROM tmp_out;
```

Export a raster as a GeoTIFF using the default creation options.

Code

```
SELECT ST_AsGDALRaster(rast, 'GTiff') AS rasttiff
FROM dummy_rast WHERE rid = 2;
```

Export a GeoTIFF using JPEG compression at 90 percent quality and embed SRID 4269.

Code

```
SELECT ST_AsGDALRaster(
  rast,
  'GTiff',
  ARRAY['COMPRESS=JPEG', 'JPEG_QUALITY=90'],
  4269) AS rasttiff
FROM dummy_rast WHERE rid = 2;
```

See Also

Section 10.4, ST_GDALDrivers, ST_SRID

11.11.4 ST_AsJPEG

ST_AsJPEG — Return the raster tile selected bands as a single Joint Photographic Exports Group (JPEG) image (byte array). If no band is specified and 1 or more than 3 bands, then only the first band is used. If only 3 bands then all 3 bands are used and mapped to RGB.

Synopsis

bytea **ST_AsJPEG**(raster rast, text[] options=NULL);
bytea **ST_AsJPEG**(raster rast, integer nband, integer quality);
bytea **ST_AsJPEG**(raster rast, integer nband, text[] options=NULL);
bytea **ST_AsJPEG**(raster rast, integer[] nbands, text[] options=NULL);
bytea **ST_AsJPEG**(raster rast, integer[] nbands, integer quality);

Description

Returns the selected bands of the raster as a single Joint Photographic Exports Group Image (JPEG). Use ST_AsGDALRaster if you need to export as less common raster types. If no band is specified and 1 or more than 3 bands, then only the first band is used. If 3 bands then all 3 bands are used. There are many variants of the function with many options. These are itemized below:

- `nband` is for single band exports.
- `nbands` is an array of bands to export (note that max is 3 for JPEG) and the order of the bands is RGB. e.g ARRAY[3,2,1] means map band 3 to Red, band 2 to green and band 1 to blue
- `quality` number from 0 to 100. The higher the number the crisper the image.
- `options` text Array of GDAL options as defined for JPEG (look at create_options for JPEG ST_GDALDrivers). For JPEG valid ones are `PROGRESSIVE` ON or OFF and `QUALITY` a range from 0 to 100 and default to 75. Refer to GDAL Raster format options for more details.

Availability: 2.0.0 - requires GDAL >= 1.6.0.

Examples: Output

Output the first three bands at 75 percent quality.

Code

```
SELECT ST_AsJPEG(rast) As rastjpg
FROM dummy_rast WHERE rid=2;
```

Output only the first band at 90 percent quality.

Code

```
SELECT ST_AsJPEG(rast, 1, 90) As rastjpg
FROM dummy_rast WHERE rid=2;
```

Output the first three bands, mapping band 2 to red, band 1 to green, and band 3 to blue, using progressive encoding at 90 percent quality.

Code

```
SELECT ST_AsJPEG(rast, ARRAY[2, 1, 3], ARRAY['QUALITY=90', 'PROGRESSIVE=ON']) As rastjpg
FROM dummy_rast WHERE rid=2;
```

See Also

Section 10.4, ST_GDALDrivers, ST_AsGDALRaster, ST_AsPNG, ST_AsTIFF

11.11.5 ST_AsPNG

ST_AsPNG — Return the raster tile selected bands as a single portable network graphics (PNG) image (byte array). If 1, 3, or 4 bands in raster and no bands are specified, then all bands are used. If more 2 or more than 4 bands and no bands specified, then only band 1 is used. Bands are mapped to RGB or RGBA space.

Synopsis

bytea **ST_AsPNG**(raster rast, text[] options=NULL);
bytea **ST_AsPNG**(raster rast, integer nband, integer compression);
bytea **ST_AsPNG**(raster rast, integer nband, text[] options=NULL);
bytea **ST_AsPNG**(raster rast, integer[] nbands, integer compression);
bytea **ST_AsPNG**(raster rast, integer[] nbands, text[] options=NULL);

Description

Returns the selected bands of the raster as a single Portable Network Graphics Image (PNG). Use ST_AsGDALRaster if you need to export as less common raster types. If no band is specified, then the first 3 bands are exported. There are many variants of the function with many options. If no `srid` is specified then then srid of the raster is used. These are itemized below:

- `nband` is for single band exports.
- `nbands` is an array of bands to export (note that max is 4 for PNG) and the order of the bands is RGBA. e.g ARRAY[3,2,1] means map band 3 to Red, band 2 to green and band 1 to blue
- `compression` number from 1 to 9. The higher the number the greater the compression.
- `options` text Array of GDAL options as defined for PNG (look at create_options for PNG of ST_GDALDrivers). For PNG valid one is only ZLEVEL (amount of time to spend on compression -- default 6) e.g. ARRAY['ZLEVEL=9']. WORLDFILE is not allowed since the function would have to output two outputs. Refer to GDAL Raster format options for more details.

Availability: 2.0.0 - requires GDAL >= 1.6.0.

Examples

Export the first 3 bands and map band 3 to red, band 1 to green, and band 2 to blue.

Code

```
SELECT ST_AsPNG(rast) As rastpng
FROM dummy_rast WHERE rid=2;
SELECT ST_AsPNG(rast, ARRAY[3, 1, 2]) As rastpng
FROM dummy_rast WHERE rid=2;
```

See Also

ST_AsGDALRaster, ST_ColorMap, ST_GDALDrivers, Section 10.4

11.11.6 ST_AsTIFF

ST_AsTIFF — Return the raster selected bands as a single TIFF image (byte array). If no band is specified or any of specified bands does not exist in the raster, then will try to use all bands.

Synopsis

bytea **ST_AsTIFF**(raster rast, text[] options='', integer srid=sameassource);
bytea **ST_AsTIFF**(raster rast, text compression='', integer srid=sameassource);
bytea **ST_AsTIFF**(raster rast, integer[] nbands, text compression='', integer srid=sameassource);
bytea **ST_AsTIFF**(raster rast, integer[] nbands, text[] options, integer srid=sameassource);

Description

Returns the selected bands of the raster as a single Tagged Image File Format (TIFF). If no band is specified, will try to use all bands. This is a wrapper around ST_AsGDALRaster. Use ST_AsGDALRaster if you need to export as less common raster types. There are many variants of the function with many options. If no spatial reference SRS text is present, the spatial reference of the raster is used. These are itemized below:

- `nbands` is an array of bands to export (note that max is 3 for PNG) and the order of the bands is RGB. e.g ARRAY[3,2,1] means map band 3 to Red, band 2 to green and band 1 to blue
- `compression` Compression expression -- JPEG90 (or some other percent), LZW, JPEG, DEFLATE9.
- `options` text Array of GDAL create options as defined for GTiff (look at create_options for GTiff of ST_GDALDrivers). or refer to GDAL Raster format options for more details.
- `srid` srid of spatial_ref_sys of the raster. This is used to populate the georeference information

Availability: 2.0.0 - requires GDAL >= 1.6.0.

Examples: Use jpeg compression 90%

Code

```
SELECT ST_AsTIFF(rast, 'JPEG90') As rasttiff
FROM dummy_rast WHERE rid=2;
```

See Also

ST_GDALDrivers, ST_AsGDALRaster, ST_SRID

11.12 Raster Processing: Map Algebra

11.12.1 ST_Clip

ST_Clip — Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If `crop` is not specified or TRUE, the output raster is cropped. If `touched` is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.

Synopsis

raster **ST_Clip**(raster rast, integer[] nband, geometry geom, double precision[] nodataval=NULL, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, integer nband, geometry geom, double precision nodataval, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, integer nband, geometry geom, boolean crop, boolean touched=FALSE);
raster **ST_Clip**(raster rast, geometry geom, double precision[] nodataval=NULL, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, geometry geom, double precision nodataval, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, geometry geom, boolean crop, boolean touched=FALSE);

Description

Returns a raster that is clipped by the input geometry `geom`. If band index is not specified, all bands are processed.

Rasters resulting from ST_Clip must have a nodata value assigned for areas clipped, one for each band. If none are provided and the input raster do not have a nodata value defined, nodata values of the resulting raster are set to ST_MinPossibleValue(ST_BandPixelTyp band)). When the number of nodata value in the array is smaller than the number of band, the last one in the array is used for the remaining bands. If the number of nodata value is greater than the number of band, the extra nodata values are ignored. All variants accepting an array of nodata values also accept a single value which will be assigned to each band.

If `crop` is not specified, true is assumed meaning the output raster is cropped to the intersection of the `geom`and `rast` extents. If `crop` is set to false, the new raster gets the same extent as `rast`. If `touched` is set to true, then all pixels in the `rast` that intersect the geometry are selected.



Note
The default behavior is touched=false, which will only select pixels where the center of the pixel is covered by the geometry.

Enhanced: 3.5.0 - touched argument added.

Availability: 2.0.0

Enhanced: 2.1.0 Rewritten in C

Examples here use Massachusetts aerial data available on MassGIS site MassGIS Aerial Orthos.

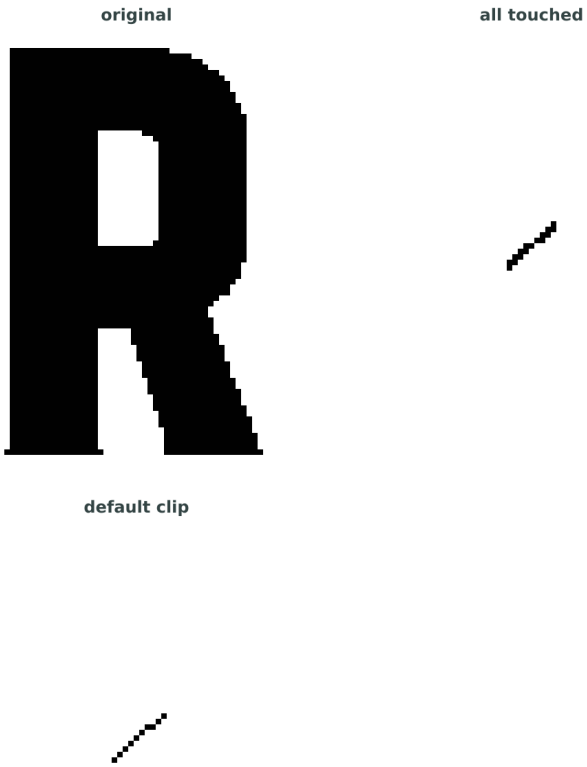
Examples

Comparing selecting all touched vs. not all touched.

Code

```
WITH source AS (
    SELECT rast, geom
    FROM ST_AsRaster(ST_Letters('R'), scalex => 1.0, scaley => -1.0) AS r(rast)
    INNER JOIN ST_GeomFromText('LINESTRING(0 1,5 6,10 10)') AS g(geom)
        ON ST_Intersects(r.rast, g.geom)
), clipped AS (
    SELECT rast,
        ST_Clip(rast, geom, touched => true) AS rast_touched,
        ST_Clip(rast, geom, touched => false) AS rast_not_touched
    FROM source
), variants AS (
    SELECT 'original' AS title, ST_Count(rast) AS pixel_count, rast AS rendered
    FROM clipped
    UNION ALL
    SELECT 'all touched', ST_Count(rast_touched), rast_touched
    FROM clipped
    UNION ALL
    SELECT 'default clip', ST_Count(rast_not_touched), rast_not_touched
    FROM clipped
)
SELECT title, pixel_count, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY CASE title
    WHEN 'original' THEN 1
    WHEN 'all touched' THEN 2
    ELSE 3
END;
```

Figure



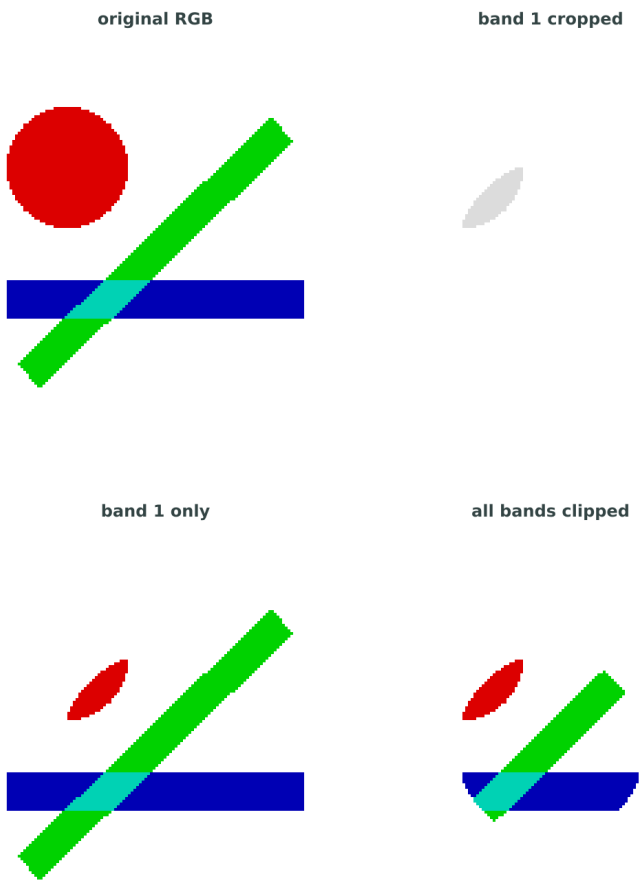
The following self-contained example uses three simple shapes as RGB bands. It compares cropping band 1, clipping band 1 without cropping and then restoring bands 2 and 3, and clipping every band without cropping.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_AddBand(
            ST_AddBand(
                ST_MakeEmptyRaster(128, 128, 0, 128, 1, -1, 0, 0, 0),
                '8BUI'::text, 0::double precision, 0::double precision
            ),
            '8BUI'::text, 0::double precision, 0::double precision
        ),
        '8BUI'::text, 0::double precision, 0::double precision
    ) AS rast
), source AS (
    SELECT ST_SetValue(
        ST_SetValue(
            ST_SetValue(rast, 1, ST_Buffer('POINT(32 96)'::geometry, 22), 220),
            2, ST_Buffer('LINESTRING(18 20,110 110)'::geometry, 6,
                         'endcap=flat join=bevel'), 210
        ),
        3, ST_Buffer('LINESTRING(10 48,118 48)'::geometry, 7,
```

```
                    'endcap=flat'), 180
    ) AS rast,
    ST_Buffer('POINT(64 64)'::geometry, 34) AS clipper
    FROM canvas
), variants AS (
    SELECT 'original RGB' AS title, rast AS rendered FROM source
    UNION ALL
    SELECT 'band 1 cropped', ST_Clip(rast, 1, clipper, true) FROM source
    UNION ALL
    SELECT
        'band 1 only',
        ST_AddBand(
            ST_Clip(rast, 1, clipper, false),
            ARRAY[ST_Band(rast, 2), ST_Band(rast, 3)]
        )
    FROM source
    UNION ALL
    SELECT 'all bands clipped', ST_Clip(rast, clipper, false) FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY CASE title
    WHEN 'original RGB' THEN 1
    WHEN 'band 1 cropped' THEN 2
    WHEN 'band 1 only' THEN 3
    ELSE 4
END;
```

Figure



Crop, single-band clipping, and all-band clipping.



Source tile



Cropped



One-band source



One-band clip



All-band source



All-band clip

See Also

ST_AddBand, ST_Count, ST_MapAlgebra (callback function version), ST_Intersection

11.12.2 ST_ColorMap

ST_ColorMap — Creates a new raster of up to four 8BUI bands (grayscale, RGB, RGBA) from the source raster and a specified band. Band 1 is assumed if not specified.

Synopsis

raster **ST_ColorMap**(raster rast, integer nband=1, text colormap=grayscale, text method=INTERPOLATE);

raster **ST_ColorMap**(raster rast, text colormap, text method=INTERPOLATE);

Description

Apply a `colormap` to the band at `nband` of `rast` resulting a new raster comprised of up to four 8BUI bands. The number of 8BUI bands in the new raster is determined by the number of color components defined in `colormap`.

If `nband` is not specified, then band 1 is assumed.

`colormap` can be a keyword of a pre-defined colormap or a set of lines defining the value and the color components.

Valid pre-defined `colormap` keyword:

- `grayscale` or `greyscale` for a one 8BUI band raster of shades of gray.
- `pseudocolor` for a four 8BUI (RGBA) band raster with colors going from blue to green to red.
- `fire` for a four 8BUI (RGBA) band raster with colors going from black to red to pale yellow.
- `bluered` for a four 8BUI (RGBA) band raster with colors going from blue to pale white to red.

Users can pass a set of entries (one per line) to `colormap` to specify custom colormaps. Each entry generally consists of five values: the pixel value and corresponding Red, Green, Blue, Alpha components (color components between 0 and 255). Percent values can be used instead of pixel values where 0% and 100% are the minimum and maximum values found in the raster band. Values can be separated with commas (','), tabs, colons (':') and/or spaces. The pixel value can be set to *nv*, *null* or *nodata* for the NODATA value. An example is provided below.

Code

```
5 0 0 0 255
4 100:50 55 255
1 150, 100 150 255
0% 255 255 255 255
nv 0 0 0 0
```

The syntax of `colormap` is similar to that of the color-relief mode of GDAL gdaldem.

Valid keywords for `method`:

- `INTERPOLATE` to use linear interpolation to smoothly blend the colors between the given pixel values
- `EXACT` to strictly match only those pixels values found in the colormap. Pixels whose value does not match a colormap entry will be set to 0 0 0 0 (RGBA)
- `NEAREST` to use the colormap entry whose value is closest to the pixel value



Note
A great reference for colormaps is ColorBrewer.



Warning
The resulting bands of new raster will have no NODATA value set. Use ST_SetBandNoDataValue to set a NODATA value if one is needed.

Availability: 2.1.0

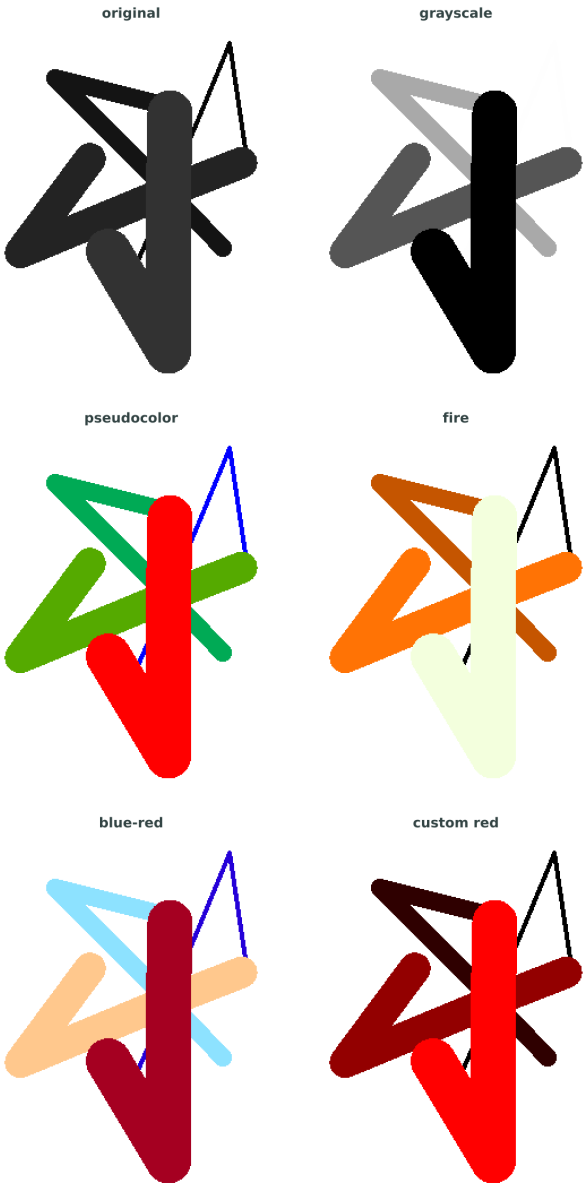
Examples

This example builds a raster from rotated buffered lines and compares the original values with predefined and custom color maps.

Code

```
WITH ref AS (
    SELECT ST_MakeEmptyRaster(200, 200, 0, 200, 1, -1, 0, 0) AS rast
), source AS (
    SELECT ST_Union(rast) AS rast
    FROM (
    SELECT
        ST_AsRaster(ST_Rotate(ST_Buffer('LINESTRING(0 2,50 50,150 150,125 50)'::geometry,
                    i*2
                ),
                pi() * i * 0.125, ST_Point(50, 50)
            ),
            ref.rast, '8BUI'::text, i * 5
        ) AS rast
    FROM ref
    CROSS JOIN generate_series(1, 10, 3) AS i
) AS shapes
)
SELECT title, ST_AsPNG(rendered) AS image
FROM source
CROSS JOIN LATERAL (VALUES
    ('original', rast),
    ('grayscale', ST_ColorMap(rast, 1, 'greyscale')),
    ('pseudocolor', ST_ColorMap(rast, 1, 'pseudocolor')),
    ('fire', ST_ColorMap(rast, 1, 'fire')),
    ('blue-red', ST_ColorMap(rast, 1, 'bluered')),
    ('custom red', ST_ColorMap(
        rast,
        1,
        E'100% 255 0 0 255\n80% 160 0 0 255\n50% 130 0 0 255\n' ||
        E'30% 30 0 0 255\n20% 60 0 0 255\n0% 0 0 0 255\n' ||
        E'nv 255 255 255 0'
    ))
) AS variants(title, rendered);
```

Figure



See Also

ST_AsPNG, ST_AsRaster ST_MapAlgebra (callback function version), ST_Grayscale ST_NumBands, ST_Reclass, ST_SetBandNoDat
ST_Union

11.12.3 ST_Grayscale

ST_Grayscale — Creates a new one-8BUI band raster from the source raster and specified bands representing Red, Green and Blue

Synopsis

(1) raster **ST_Grayscale**(raster rast, integer redband=1, integer greenband=2, integer blueband=3, text extenttype=INTERSECTION);

(2) raster **ST_Grayscale**(rastbandarg[] rastbandargset, text extenttype=INTERSECTION);

Description

Create a raster with one 8BUI band given three input bands (from one or more rasters). Any input band whose pixel type is not 8BUI will be reclassified using ST_Reclass.



Note
This function is not like ST_ColorMap with the `grayscale` keyword as ST_ColorMap operates on only one band while this function expects three bands for RGB. This function applies the following equation for converting RGB to Grayscale: 0.2989 * RED + 0.5870 * GREEN + 0.1140 * BLUE

Availability: 2.5.0

Examples

This self-contained example paints an apple with gradients and highlights into three raster bands. It compares the original RGB raster with the equivalent grayscale results produced by both function variants.

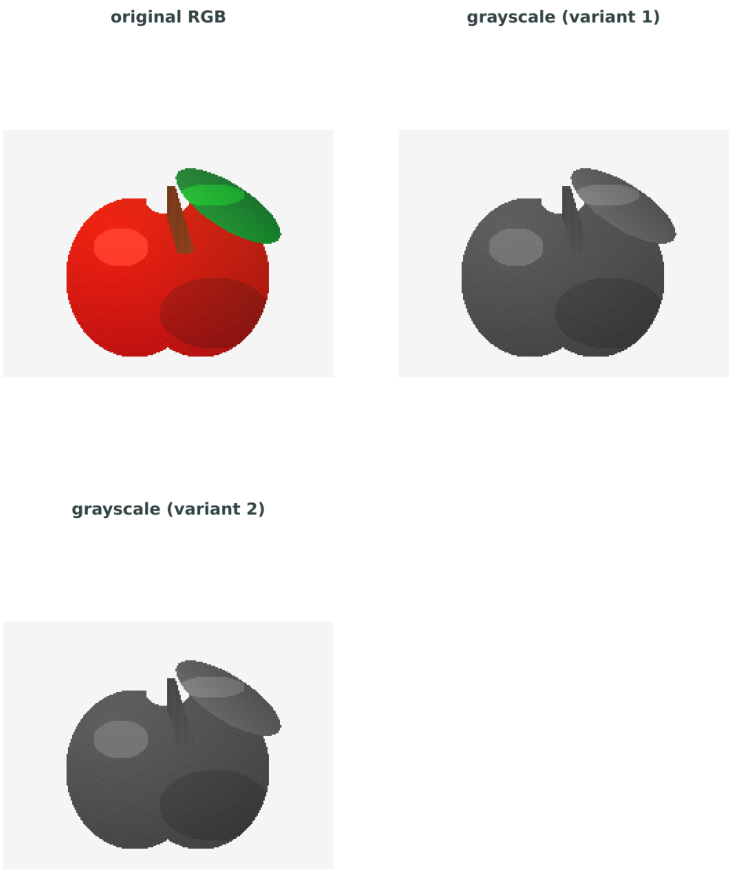
Code

```
WITH pixels AS (
    SELECT
        col,
        row,
        x,
        y,
        (((x - 48.0)^2 / (25.0^2) + (y - 54.0)^2 / (29.0^2) < 1)
         OR ((x - 72.0)^2 / (25.0^2) + (y - 54.0)^2 / (29.0^2) < 1))
         AND NOT ((x - 60.0)^2 / (8.0^2) + (y - 26.0)^2 / (5.0^2) < 1) AS fruit,
        ((x - 82.0) * 0.84 + (y - 28.0) * 0.54)^2 / (22.0^2)
          + (-(x - 82.0) * 0.54 + (y - 28.0) * 0.84)^2 / (8.0^2) < 1 AS leaf,
        abs((x - 62.0) * 0.96 - (y - 29.0) * 0.28) < 3
          AND x BETWEEN 60 AND 69 AND y BETWEEN 21 AND 45 AS stem
    FROM (
        SELECT
            col,
            row,
            col / 2.0 AS x,
            row / 2.0 AS y
        FROM generate_series(1, 180) AS row
        CROSS JOIN generate_series(1, 240) AS col
    ) AS source_pixels
), shaded AS (
    SELECT
        col,
        row,
        x,
        y,
        CASE
            WHEN stem THEN 95 + y * 0.7 + (x % 3) * 8
            WHEN leaf THEN 28 + (90 - x) * 0.5
            WHEN fruit THEN LEAST(255, GREATEST(0,
                150
                + (85 - sqrt((x - 39.0)^2 + (y - 37.0)^2)) * 1.1
                + CASE
                    WHEN (x - 43.0)^2 / (10.0^2) + (y - 43.0)^2 / (7.0^2) < 1
```

```
                    THEN 35
                    ELSE 0
                END
                - CASE
                    WHEN (x - 77.0)^2 / (20.0^2) + (y - 67.0)^2 / (13.0^2) < 1
                    THEN 35
                    ELSE 0
                END
                + ((x * 13 + y * 7) % 19 - 9)
            ))
            ELSE 245
        END AS red,
        CASE
            WHEN stem THEN 50 + y * 0.35
            WHEN leaf THEN LEAST(190, GREATEST(40,
                105 + (95 - x) * 1.2 + (y - 20) * 1.5
                + CASE
                    WHEN (x - 76.0)^2 / (12.0^2) + (y - 24.0)^2 / (4.0^2) < 1
                    THEN 45
                    ELSE 0
                END
            ))
            WHEN fruit THEN LEAST(95, GREATEST(15,
                24 + (68 - y) * 0.35
                + CASE
                    WHEN (x - 43.0)^2 / (10.0^2) + (y - 43.0)^2 / (7.0^2) < 1
                    THEN 30
                    ELSE 0
                END
                + ((x * 5 + y * 11) % 13 - 6)
            ))
            ELSE 245
        END AS green,
        CASE
            WHEN stem THEN 22 + (x % 4) * 3
            WHEN leaf THEN 42 + (100 - x) * 0.4
            WHEN fruit THEN LEAST(75, GREATEST(10,
                18
                + CASE
                    WHEN (x - 43.0)^2 / (10.0^2) + (y - 43.0)^2 / (7.0^2) < 1
                    THEN 25
                    ELSE 0
                END
                + ((x * 3 + y * 5) % 11 - 5)
            ))
            ELSE 245
        END AS blue
    FROM pixels
), rows AS (
    SELECT
        row,
        array_agg(red::double precision ORDER BY col) AS red_values,
        array_agg(green::double precision ORDER BY col) AS green_values,
        array_agg(blue::double precision ORDER BY col) AS blue_values
    FROM shaded
    GROUP BY row
), source AS (
    SELECT ST_SetValues(ST_SetValues(ST_SetValues(
        ST_AddBand(ST_AddBand(ST_AddBand(
            ST_MakeEmptyRaster(240, 180, 0, 180, 1, -1, 0, 0, 0),
            '8BUI'::text, 245::double precision, 0::double precision),
            '8BUI'::text, 245::double precision, 0::double precision),
```

```
            '8BUI'::text, 245::double precision, 0::double precision),
        1, 1, 1, array_agg(red_values ORDER BY row)::double precision[][]),
        2, 1, 1, array_agg(green_values ORDER BY row)::double precision[][]),
        3, 1, 1, array_agg(blue_values ORDER BY row)::double precision[][]) AS rast
    FROM rows
)
SELECT ST_AsPNG(rast) AS "original RGB",
    ST_AsPNG(ST_Grayscale(rast)) AS "grayscale (variant 1)",
    ST_AsPNG(ST_Grayscale(ARRAY[
        ROW(rast, 1)::rastbandarg,
        ROW(rast, 2)::rastbandarg,
        ROW(rast, 3)::rastbandarg
    ]::rastbandarg[])) AS "grayscale (variant 2)"
FROM source;
```

Figure



Original photographic illustration:



original apple



grayscale apple

See Also

ST_AsPNG, ST_Reclass, ST_ColorMap

11.12.4 ST_Intersection

ST_Intersection — Returns a raster or a set of geometry-pixelvalue pairs representing the shared portion of two rasters or the geometrical intersection of a vectorization of the raster and a geometry.

Synopsis

setof geomval **ST_Intersection**(geometry geom, raster rast, integer band_num=1);
setof geomval **ST_Intersection**(raster rast, geometry geom);
setof geomval **ST_Intersection**(raster rast, integer band, geometry geomin);
raster **ST_Intersection**(raster rast1, raster rast2, double precision[] nodataval);
raster **ST_Intersection**(raster rast1, raster rast2, text returnband, double precision[] nodataval);
raster **ST_Intersection**(raster rast1, integer band1, raster rast2, integer band2, double precision[] nodataval);
raster **ST_Intersection**(raster rast1, integer band1, raster rast2, integer band2, text returnband, double precision[] nodataval);

Description

Returns a raster or a set of geometry-pixelvalue pairs representing the shared portion of two rasters or the geometrical intersection of a vectorization of the raster and a geometry.

The first three variants, returning a setof geomval, works in vector space. The raster is first vectorized (using ST_DumpAsPolygons) into a set of geomval rows and those rows are then intersected with the geometry using the ST_Intersection (geometry, geometry) PostGIS function. Geometries intersecting only with a nodata value area of a raster returns an empty geometry. They are normally excluded from the results by the proper usage of ST_Intersects in the WHERE clause.

You can access the geometry and the value parts of the resulting set of geomval by surrounding them with parenthesis and adding '.geom' or '.val' at the end of the expression. e.g. (ST_Intersection(rast, geom)).geom

The other variants, returning a raster, works in raster space. They are using the two rasters version of ST_MapAlgebraExpr to perform the intersection.

The extent of the resulting raster corresponds to the geometrical intersection of the two raster extents. The resulting raster includes 'BAND1', 'BAND2' or 'BOTH' bands, following what is passed as the `returnband` parameter. Nodata value areas

present in any band results in nodata value areas in every bands of the result. In other words, any pixel intersecting with a nodata value pixel becomes a nodata value pixel in the result.

Rasters resulting from ST_Intersection must have a nodata value assigned for areas not intersecting. You can define or replace the nodata value for any resulting band by providing a `nodataval[]` array of one or two nodata values depending if you request 'BAND1', 'BAND2' or 'BOTH' bands. The first value in the array replace the nodata value in the first band and the second value replace the nodata value in the second band. If one input band do not have a nodata value defined and none are provided as an array, one is chosen using the ST_MinPossibleValue function. All variant accepting an array of nodata value can also accept a single value which will be assigned to each requested band.

In all variants, if no band number is specified band 1 is assumed. If you need an intersection between a raster and geometry that returns a raster, refer to ST_Clip.



Note
For more control over the result extent or NODATA handling, use the two-raster variant of ST_MapAlgebraExpr. To compute a raster-space intersection with a geometry, use ST_Clip.
Use this function together with ST_Intersects and an index on the raster or geometry column when filtering table data.

Enhanced: 2.0.0 - Intersection in the raster space was introduced. In earlier pre-2.0.0 versions, only intersection performed in vector space were supported.

Examples: Geometry, Raster -- resulting in geometry vals

Code

```
SELECT
    foo.rid,
    foo.gid,
    GeometryType((foo.geomval).geom) AS geom_type,
    (foo.geomval).val AS value
FROM (
    SELECT
        A.rid,
        g.gid,
        ST_Intersection(A.rast, g.geom) As geomval
    FROM dummy_rast AS A
    CROSS JOIN (
        VALUES
            (1, ST_Point(3427928, 5793243.85) ),
            (2, 'LINESTRING(3427927.85 5793243.75,3427927.8 5793243.75,3427927.8 5793243.8)' ↩
    ::geometry),
            (3, 'LINESTRING(1 2,3 4)'::geometry)
    ) As g(gid, geom)
    WHERE A.rid = 2
) AS foo;
```

Output

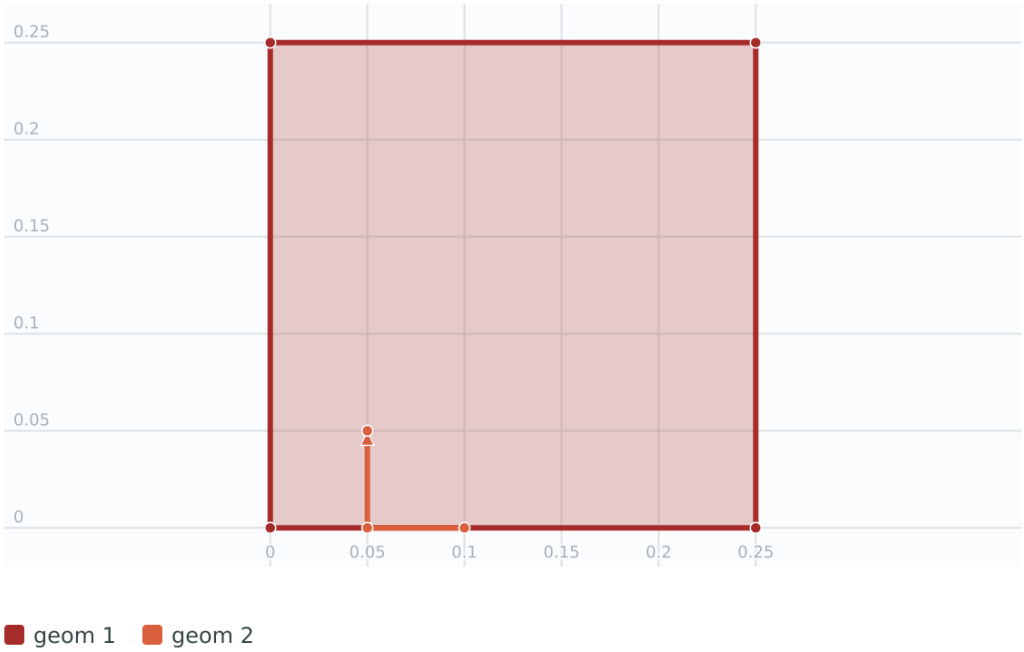
```
 rid | gid |      geom_type       | value
-----+-----+----------------------+-------
   2 |   1 | POINT                |   249
   2 |   1 | POINT                |   253
   2 |   2 | POINT                |   254
   2 |   2 | POINT                |   251
   2 |   2 | POINT                |   253
   2 |   2 | LINESTRING           |   252
   2 |   2 | MULTILINESTRING      |   250
   2 |   3 | GEOMETRYCOLLECTION   |
```

The same line example rendered in a common coordinate frame. The raster footprint and complete input line remain visible behind the returned intersection pieces.

Code

```
WITH source AS (
    SELECT
        rast,
        ST_GeomFromText(
            'LINESTRING(3427927.85 5793243.75,'
            '3427927.8 5793243.75,3427927.8 5793243.8)'
        ) AS input_geometry
    FROM dummy_rast
    WHERE rid = 2
), intersection AS (
SELECT
    rast,
    input_geometry,
    ST_Collect(gv.geom) AS result_geometry
FROM source
CROSS JOIN LATERAL ST_Intersection(rast, input_geometry) AS gv
GROUP BY rast, input_geometry
), layers AS (
    SELECT
        'raster' AS title,
        ST_Translate(ST_Envelope(rast), -3427927.75, -5793243.75) AS geom
    FROM intersection
    UNION ALL
    SELECT
        'input',
        ST_Translate(input_geometry, -3427927.75, -5793243.75)
    FROM intersection
    UNION ALL
    SELECT
        'intersection',
        ST_Translate(result_geometry, -3427927.75, -5793243.75)
    FROM intersection
)
SELECT title, geom AS geom
FROM layers
ORDER BY CASE title
    WHEN 'raster' THEN 1
    WHEN 'input' THEN 2
    ELSE 3
END;
```

Figure



See Also

geomval, ST_Intersects, ST_MapAlgebraExpr, ST_Clip, ST_AsText

11.12.5 ST_MapAlgebra (callback function version)

ST_MapAlgebra (callback function version) — Callback function version - Returns a one-band raster given one or more input rasters, band indexes and one user-specified callback function.

Synopsis

raster **ST_MapAlgebra**(rastbandarg[] rastbandargset, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=INTERSECTIO
raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast, integer[] nband, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=FIRST,
raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast, integer nband, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=FIRST,
raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast1, integer nband1, raster rast2, integer nband2, regprocedure callbackfunc, text pixeltype=NULL,
text extenttype=INTERSECTION, raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC user-
args=NULL);
raster **ST_MapAlgebra**(raster rast, integer nband, regprocedure callbackfunc, float8[] mask, boolean weighted, text pixel-
type=NULL, text extenttype=INTERSECTION, raster customextent=NULL, text[] VARIADIC userargs=NULL);

Description

Returns a one-band raster given one or more input rasters, band indexes and one user-specified callback function.

`rast,rast1,rast2, rastbandargset` Rasters on which the map algebra process is evaluated.

`rastbandargset` allows the use of a map algebra operation on many rasters and/or many bands. See example Variant 1.

nband, nband1, nband2 Band numbers of the raster to be evaluated. nband can be an integer or integer[] denoting the bands. nband1 is band on rast1 and nband2 is band on rast2 for the 2 raster/2band case.

callbackfunc The `callbackfunc` parameter must be the name and signature of an SQL or PL/pgSQL function, cast to a regprocedure. An example PL/pgSQL function example is:

Code

```
CREATE OR REPLACE FUNCTION sample_callbackfunc(value double precision[][][], pos ↩
    integer[][], VARIADIC userargs text[])
    RETURNS double precision
    AS $$
    BEGIN
        RETURN 0;
    END;
    $$ LANGUAGE 'plpgsql' IMMUTABLE;
```

The `callbackfunc` must have two or three arguments: a 3-dimension double precision array, a 2-dimension integer array and, optionally, a variadic 1-dimension text array. The first argument `value` is the set of values (as double precision) from all input rasters. The three dimensions (where indexes are 1-based) are: raster #, row y, column x. The second argument `position` is the set of pixel positions from the output raster and input rasters. The outer dimension (where indexes are 0-based) is the raster #. The position at outer dimension index 0 is the output raster's pixel position. For each outer dimension, there are two elements in the inner dimension for X and Y. The optional third argument `userargs` is for passing through any user-specified arguments.

The callback return value is used as the value of the output pixel. If the callback returns `NULL`, the output pixel is set to NODATA when the output band has a NODATA value. If the output band has no NODATA value, a `NULL` callback result raises an error.

Passing a regprocedure argument to a SQL function requires the full function signature to be passed, then cast to a regprocedure type. To pass the above example PL/pgSQL function as an argument, the SQL for the argument is:

Code

```
'sample_callbackfunc(double precision[], integer[], text[])'::regprocedure
```

Note that PostgreSQL array types do not encode dimensions in function signatures, so multidimensional callback arguments are written as double precision[] and integer[] in the regprocedure signature.

mask An n-dimensional array (matrix) of numbers used to filter what cells get passed to map algebra call-back function. 0 means a neighbor cell value should be treated as no-data and 1 means value should be treated as data. If weight is set to true, then the values, are used as multipliers to multiple the pixel value of that value in the neighborhood position.

weighted boolean (true/false) to denote if a mask value should be weighted (multiplied by original value) or not (only applies to proto that takes a mask).

pixeltype If `pixeltype` is passed in, the one band of the new raster will be of that pixeltype. If pixeltype is passed NULL or left out, the new raster band will have the same pixeltype as the specified band of the first raster (for extent types: INTERSECTION, UNION, FIRST, CUSTOM) or the specified band of the appropriate raster (for extent types: SECOND, LAST). If in doubt, always specify `pixeltype`.

The resulting pixel type of the output raster must be one listed in ST_BandPixelType or left out or set to NULL.

extenttype Possible values are INTERSECTION (default), UNION, FIRST (default for one raster variants), SECOND, LAST, CUSTOM.

customextent If `extentype` is CUSTOM, a raster must be provided for `customextent`. See example 4 of Variant 1.

distancex The distance in pixels from the reference cell in x direction. So width of resulting matrix would be `2*distancex + 1`.If not specified only the reference cell is considered (neighborhood of 0).

distancey The distance in pixels from reference cell in y direction. Height of resulting matrix would be `2*distancey + 1` .If not specified only the reference cell is considered (neighborhood of 0).

userargs If the `callbackfunc` declares a third argument, that argument is a variadic text array. All trailing text arguments are passed through to the specified `callbackfunc`, and are contained in the `userargs` argument.



Note
For more information about the VARIADIC keyword, please refer to the PostgreSQL documentation and the "SQL Functions with Variable Numbers of Arguments" section of Query Language (SQL) Functions.

Variant 1 accepts an array of `rastbandarg` allowing the use of a map algebra operation on many rasters and/or many bands. See example Variant 1.

Variants 2 and 3 operate upon one or more bands of one raster. See example Variant 2 and 3.

Variant 4 operate upon two rasters with one band per raster. See example Variant 4.

Availability: 2.2.0: Ability to add a mask

Availability: 2.1.0

Examples

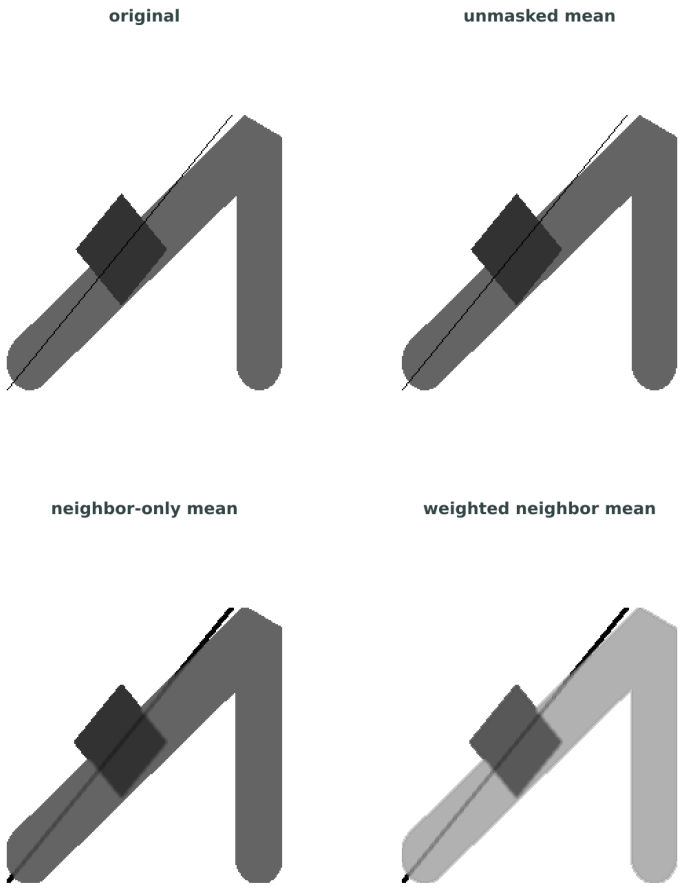
Compare an unmasked mean with neighbor-only and weighted-neighbor masks. The weighted mask multiplies neighboring values by 2 before computing the result.

Code

```
WITH foo AS (
    SELECT ST_SetBandNoDataValue(
        ST_SetValue(
            ST_SetValue(
                ST_AsRaster(
                    ST_Buffer('LINESTRING(50 50,100 90,100 50)'::geometry, 5, 'join=bevel'),
                    200, 200, ARRAY['8BUI'], ARRAY[100], ARRAY[0]
                ),
                ST_Buffer('POINT(70 70)'::geometry, 10, 'quad_segs=1'),
                50
            ),
            'LINESTRING(20 20,100 100,150 98)'::geometry,
            1
        ),
        0
    ) AS rast
), variants AS (
    SELECT 'original' AS title, rast
    FROM foo
    UNION ALL
    SELECT 'unmasked mean' AS title,
        ST_MapAlgebra(rast, 1, 'ST_Mean4ma(double precision[], int[], text[])'::regprocedure ↩
    ) AS rast
    FROM foo
    UNION ALL
    SELECT 'neighbor-only mean' AS title,
        ST_MapAlgebra(
            rast,
            1,
            'ST_Mean4ma(double precision[], int[], text[])'::regprocedure,
            '{{1,1,1}, {1,0,1}, {1,1,1}}'::double precision[],
            false
        ) AS rast
    FROM foo
```

```
    UNION ALL
    SELECT 'weighted neighbor mean' AS title,
        ST_MapAlgebra(
            rast,
            1,
            'ST_Mean4ma(double precision[], int[], text[])'::regprocedure,
            '{{2,2,2}, {2,0,2}, {2,2,2}}'::double precision[],
            true
        ) AS rast
    FROM foo
)
SELECT title, ST_AsPNG(rast) AS image
FROM variants;
```

Figure



Variant 1.

One raster, one band

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, 0, 0, 0), 1, '16BUI', ↩
    1, 0) AS rast
)
```

```
SELECT
    ST_MapAlgebra(ARRAY[ROW(rast, 1)]::rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

One raster, several bands

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1,  ↩
    0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast
)
SELECT
    ST_MapAlgebra(ARRAY[ROW(rast, 3), ROW(rast, 1), ROW(rast, 3), ROW(rast, 2)]::rastbandarg ↩
    [],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

Several rasters, several bands

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1,  ↩
    0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast UNION ALL
    SELECT 2 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 1, 1, -1,  ↩
    0, 0, 0), 1, '16BUI', 2, 0), 2, '8BUI', 20, 0), 3, '32BUI', 300, 0) AS rast
)
SELECT
    ST_MapAlgebra(ARRAY[ROW(t1.rast, 3), ROW(t2.rast, 1), ROW(t2.rast, 3), ROW(t1.rast, 2) ↩
    ]::rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 1
    AND t2.rid = 2
```

Complete example of tiles of a coverage with neighborhood.

Code

```
WITH foo AS (
    SELECT 0 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, 0, 0, 0), 1, '16BUI',  ↩
    1, 0) AS rast UNION ALL
    SELECT 1, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, 0, 1, -1, 0, 0, 0), 1, '16BUI', 2, 0)  ↩
    AS rast UNION ALL
    SELECT 2, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, 0, 1, -1, 0, 0, 0), 1, '16BUI', 3, 0)  ↩
    AS rast UNION ALL

    SELECT 3, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -2, 1, -1, 0, 0, 0), 1, '16BUI', 10, 0) ↩
     AS rast UNION ALL
    SELECT 4, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -2, 1, -1, 0, 0, 0), 1, '16BUI', 20, 0) ↩
     AS rast UNION ALL
    SELECT 5, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -2, 1, -1, 0, 0, 0), 1, '16BUI', 30, 0) ↩
     AS rast UNION ALL
```

```
    SELECT 6, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -4, 1, -1, 0, 0, 0), 1, '16BUI', 100, ↩
    0) AS rast UNION ALL
    SELECT 7, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -4, 1, -1, 0, 0, 0), 1, '16BUI', 200, ↩
    0) AS rast UNION ALL
    SELECT 8, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -4, 1, -1, 0, 0, 0), 1, '16BUI', 300, ↩
    0) AS rast
)
SELECT
    t1.rid,
    ST_MapAlgebra(ARRAY[ROW(ST_Union(t2.rast), 1)]::rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure,
        '32BUI',
        'CUSTOM', t1.rast,
        1, 1
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 4
    AND t2.rid BETWEEN 0 AND 8
    AND ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rid, t1.rast
```

Example like the prior one for tiles of a coverage with neighborhood but using only basic SQL features.

Code

```
WITH src AS (
    SELECT 0 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, 0, 0, 0), 1, '16BUI', ↩
    1, 0) AS rast UNION ALL
    SELECT 1, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, 0, 1, -1, 0, 0, 0), 1, '16BUI', 2, 0) ↩
    AS rast UNION ALL
    SELECT 2, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, 0, 1, -1, 0, 0, 0), 1, '16BUI', 3, 0) ↩
    AS rast UNION ALL

    SELECT 3, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -2, 1, -1, 0, 0, 0), 1, '16BUI', 10, 0) ↩
     AS rast UNION ALL
    SELECT 4, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -2, 1, -1, 0, 0, 0), 1, '16BUI', 20, 0) ↩
     AS rast UNION ALL
    SELECT 5, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -2, 1, -1, 0, 0, 0), 1, '16BUI', 30, 0) ↩
     AS rast UNION ALL

    SELECT 6, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -4, 1, -1, 0, 0, 0), 1, '16BUI', 100, ↩
    0) AS rast UNION ALL
    SELECT 7, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -4, 1, -1, 0, 0, 0), 1, '16BUI', 200, ↩
    0) AS rast UNION ALL
    SELECT 8, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -4, 1, -1, 0, 0, 0), 1, '16BUI', 300, ↩
    0) AS rast
), foo AS (
    SELECT
        t1.rid,
        ST_Union(t2.rast) AS rast
    FROM src t1
    JOIN src t2
        ON ST_Intersects(t1.rast, t2.rast)
        AND t2.rid BETWEEN 0 AND 8
    WHERE t1.rid = 4
    GROUP BY t1.rid
), bar AS (
    SELECT
        t1.rid,
```

```
        ST_MapAlgebra(
            ARRAY[ROW(t2.rast, 1)]::rastbandarg[],
            'sample_callbackfunc(double precision[], int[], text[])'::regprocedure,
            '32BUI',
            'CUSTOM', t1.rast,
            1, 1
        ) AS rast
    FROM src t1
    JOIN foo t2
        ON t1.rid = t2.rid
)
SELECT
    rid,
    (ST_MetaData(rast)),
    (ST_BandMetaData(rast, 1)),
    ST_Value(rast, 1, 1, 1)
FROM bar;
```

Variants 2 and 3.

One raster, several bands

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
    0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        rast,
        ARRAY[3, 1, 3, 2]::integer[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

One raster, one band

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
    0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        rast,
        2,
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

Variant 4.

Two rasters, two bands

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
    0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast UNION ALL
    SELECT 2 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 1, 1, -1, ↩
    0, 0, 0), 1, '16BUI', 2, 0), 2, '8BUI', 20, 0), 3, '32BUI', 300, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        t1.rast,
        2,
        t2.rast,
        1,
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 1
    AND t2.rid = 2
```

See Also

rastbandarg, ST_Union, ST_MapAlgebra (expression version)

11.12.6 ST_MapAlgebra (expression version)

ST_MapAlgebra (expression version) — Expression version - Returns a one-band raster given one or two input rasters, band indexes and one or more user-specified SQL expressions.

Synopsis

raster **ST_MapAlgebra**(raster rast, integer nband, text pixeltype, text expression, double precision nodataval=NULL);
raster **ST_MapAlgebra**(raster rast, text pixeltype, text expression, double precision nodataval=NULL);
raster **ST_MapAlgebra**(raster rast1, integer nband1, raster rast2, integer nband2, text expression, text pixeltype=NULL, text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);
raster **ST_MapAlgebra**(raster rast1, raster rast2, text expression, text pixeltype=NULL, text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);

Description

Expression version - Returns a one-band raster given one or two input rasters, band indexes and one or more user-specified SQL expressions.

Availability: 2.1.0

Description: Variants 1 and 2 (one raster)

Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation defined by the `expression` on the input raster (`rast`). If `nband` is not provided, band 1 is assumed. The new raster will have the same georeference, width, and height as the original raster but will only have one band.

If `pixeltype` is passed in, then the new raster will have a band of that pixeltype. If pixeltype is passed NULL, then the new raster band will have the same pixeltype as the input `rast` band.

- Keywords permitted for `expression`
  1. `[rast]` - Pixel value of the pixel of interest
  2. `[rast.val]` - Pixel value of the pixel of interest
  3. `[rast.x]` - 1-based pixel column of the pixel of interest
  4. `[rast.y]` - 1-based pixel row of the pixel of interest

Description: Variants 3 and 4 (two raster)

Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation to the two bands defined by the `expression` on the two input raster bands `rast1`, (`rast2`). If no `band1`, `band2` is specified band 1 is assumed. The resulting raster will be aligned (scale, skew and pixel corners) on the grid defined by the first raster. The resulting raster will have the extent defined by the `extenttype` parameter.

`expression` A PostgreSQL algebraic expression involving the two rasters and PostgreSQL defined functions/operators that will define the pixel value when pixels intersect. e.g. (([rast1] + [rast2])/2.0)::integer

`pixeltype` The resulting pixel type of the output raster. Must be one listed in ST_BandPixelType, left out or set to NULL. If not passed in or set to NULL, will default to the pixeltype of the first raster.

`extenttype` Controls the extent of resulting raster

1. `INTERSECTION` - The extent of the new raster is the intersection of the two rasters. This is the default.
2. `UNION` - The extent of the new raster is the union of the two rasters.
3. `FIRST` - The extent of the new raster is the same as the one of the first raster.
4. `SECOND` - The extent of the new raster is the same as the one of the second raster.

`nodata1expr` An algebraic expression involving only `rast2` or a constant that defines what to return when pixels of `rast1` are nodata values and spatially corresponding rast2 pixels have values.

`nodata2expr` An algebraic expression involving only `rast1` or a constant that defines what to return when pixels of `rast2` are nodata values and spatially corresponding rast1 pixels have values.

`nodatanodataval` A numeric constant to return when spatially corresponding rast1 and rast2 pixels are both nodata values.

- Keywords permitted in `expression`, `nodata1expr` and `nodata2expr`
  1. `[rast1]` - Pixel value of the pixel of interest from `rast1`
  2. `[rast1.val]` - Pixel value of the pixel of interest from `rast1`
  3. `[rast1.x]` - 1-based pixel column of the pixel of interest from `rast1`
  4. `[rast1.y]` - 1-based pixel row of the pixel of interest from `rast1`
  5. `[rast2]` - Pixel value of the pixel of interest from `rast2`
  6. `[rast2.val]` - Pixel value of the pixel of interest from `rast2`
  7. `[rast2.x]` - 1-based pixel column of the pixel of interest from `rast2`
  8. `[rast2.y]` - 1-based pixel row of the pixel of interest from `rast2`

Examples

Variants 1 and 2.

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, 1, 0, 0, 0), '32BF'::text, 1, -1)
    AS rast
)
SELECT
    ST_MapAlgebra(rast, 1, NULL, 'ceil([rast]*[rast.x]/[rast.y]+[rast.val])')
FROM foo;
```

Variant 3 and 4.

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1,
    0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI'::text, 100, 0) AS rast UNION
    ALL
    SELECT 2 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 1, 1, -1,
    0, 0, 0), 1, '16BUI', 2, 0), 2, '8BUI', 20, 0), 3, '32BUI'::text, 300, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        t1.rast,
        2,
        t2.rast,
        1,
        '([rast2] + [rast1.val]) / 2'
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 1
    AND t2.rid = 2;
```

See Also

rastbandarg, ST_Union, ST_MapAlgebra (callback function version)

11.12.7 ST_MapAlgebraExpr

ST_MapAlgebraExpr — 1 raster band version: Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation on the input raster band and of pixeltype provided. Band 1 is assumed if no band is specified.

Synopsis

raster **ST_MapAlgebraExpr**(raster rast, integer band, text pixeltype, text expression, double precision nodataval=NULL);
raster **ST_MapAlgebraExpr**(raster rast, text pixeltype, text expression, double precision nodataval=NULL);

Description



Warning
ST_MapAlgebraExpr is deprecated as of 2.1.0. Use ST_MapAlgebra (expression version) instead.

Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation defined by the `expression` on the input raster (`rast`). If no `band` is specified band 1 is assumed. The new raster will have the same georeference, width, and height as the original raster but will only have one band.

If `pixeltype` is passed in, then the new raster will have a band of that pixeltype. If pixeltype is passed NULL, then the new raster band will have the same pixeltype as the input `rast` band.

In the expression you can use the term `[rast]` to refer to the pixel value of the original band, `[rast.x]` to refer to the 1-based pixel column index, `[rast.y]` to refer to the 1-based pixel row index.

Availability: 2.0.0

Examples

Create a new 1 band raster from our original that is a function of modulo 2 of the original raster band.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast raster;
UPDATE dummy_rast SET map_rast = ST_MapAlgebraExpr(rast, NULL, 'mod([rast]::numeric,2)')  ↩
    WHERE rid = 2;

SELECT
    ST_Value(rast, 1, i, j) As origval,
    ST_Value(map_rast, 1, i, j) As mapval
FROM dummy_rast
CROSS JOIN generate_series(1, 3) AS i
CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
 origval | mapval
---------+--------
     253 |      1
     254 |      0
     253 |      1
     253 |      1
     254 |      0
     254 |      0
     250 |      0
     254 |      0
     254 |      0
```

Create a new 1 band raster of pixel-type 2BUI from our original that is reclassified and set the nodata value to be 0.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast2 raster;
UPDATE dummy_rast SET
    map_rast2 = ST_MapAlgebraExpr(rast, '2BUI'::text, 'CASE WHEN [rast] BETWEEN 100 and 250  ↩
    THEN 1 WHEN [rast] = 252 THEN 2 WHEN [rast] BETWEEN 253 and 254 THEN 3 ELSE 0 END'::text ↩
    , '0')
WHERE rid = 2;

SELECT DISTINCT
    ST_Value(rast, 1, i, j) As origval,
    ST_Value(map_rast2, 1, i, j) As mapval
FROM dummy_rast
CROSS JOIN generate_series(1, 5) AS i
CROSS JOIN generate_series(1, 5) AS j
WHERE rid = 2;
```

Output

```
 origval | mapval
---------+--------
     249 |      1
     250 |      1
     251 |
     252 |      2
     253 |      3
     254 |      3
```

Code

```
SELECT
    ST_BandPixelType(map_rast2) As b1pixtyp
FROM dummy_rast
WHERE rid = 2;
```

Output

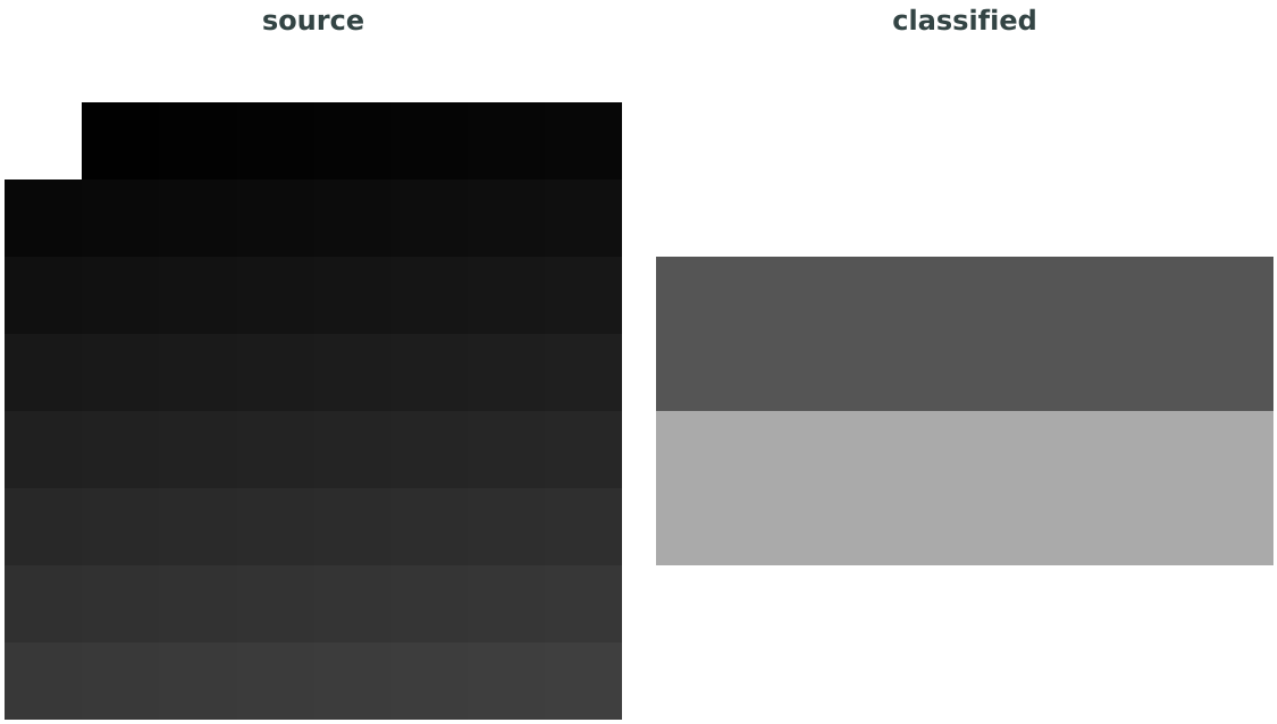
```
2BUI
```

A self-contained source raster and the raster produced by classifying its values into four display levels.

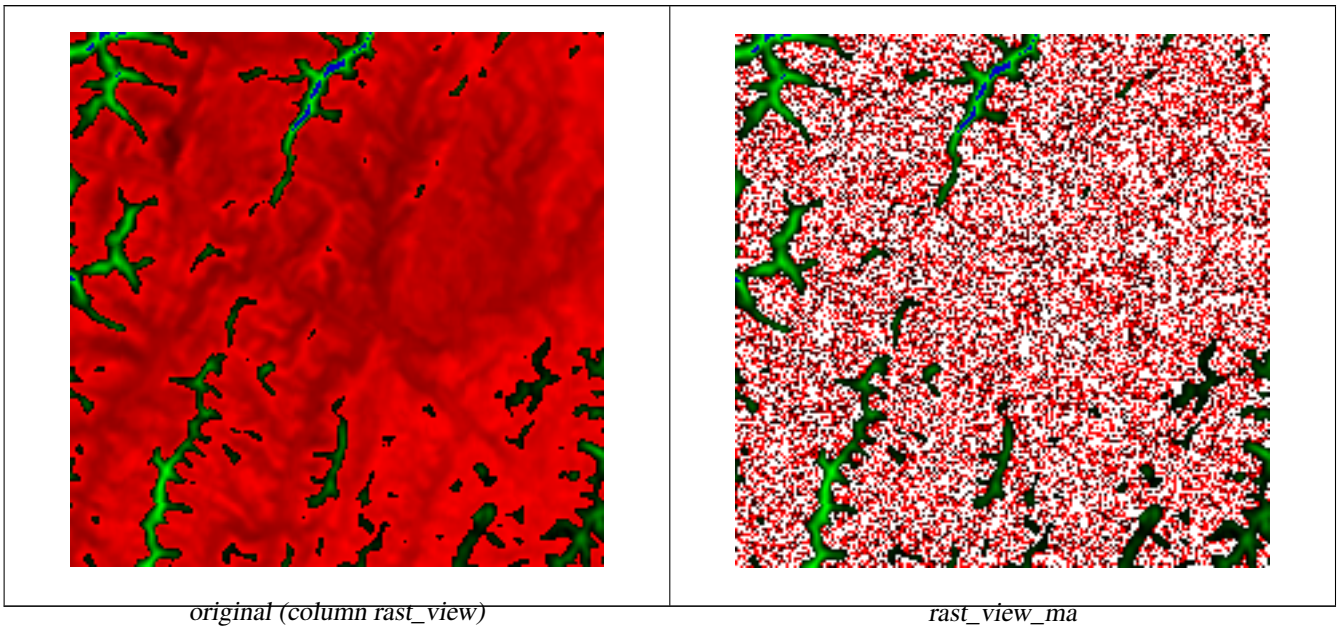
Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(8, 8, 0, 8, 1, -1, 0, 0, 0),
            1, '8BUI', 0, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 2, 3, 4, 5, 6, 7],
            [8, 9, 10, 11, 12, 13, 14, 15],
            [16, 17, 18, 19, 20, 21, 22, 23],
            [24, 25, 26, 27, 28, 29, 30, 31],
            [32, 33, 34, 35, 36, 37, 38, 39],
            [40, 41, 42, 43, 44, 45, 46, 47],
            [48, 49, 50, 51, 52, 53, 54, 55],
            [56, 57, 58, 59, 60, 61, 62, 63]
        ]::double precision[][]
    ) AS rast
), variants AS (
    SELECT 'source' AS title, rast AS rendered
    FROM source
    UNION ALL
    SELECT
        'classified',
        ST_MapAlgebraExpr(rast, '8BUI', 'floor([rast] / 16) * 85')
    FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

Figure



Map algebra altering the first band of a three-band raster.



original (column rast_view) *rast_view_ma*

Create a new 3 band raster same pixel type from our original 3 band raster with first band altered by map algebra and remaining 2 bands unaltered.

Code

```
SELECT
    ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(rast_view),
                ST_MapAlgebraExpr(rast_view, 1, NULL, 'tan([rast])*[rast]')
            ),
            ST_Band(rast_view, 2)
        ),
        ST_Band(rast_view, 3)
    )  As rast_view_ma
FROM wind
WHERE rid=167;
```

See Also

ST_MapAlgebraExpr, ST_MapAlgebraFct, ST_BandPixelType, ST_GeoReference, ST_Value

11.12.8 ST_MapAlgebraExpr

ST_MapAlgebraExpr — 2 raster band version: Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation on the two input raster bands and of pixeltype provided. band 1 of each raster is assumed if no band numbers are specified. The resulting raster will be aligned (scale, skew and pixel corners) on the grid defined by the first raster and have its extent defined by the "extenttype" parameter. Values for "extenttype" can be: INTERSECTION, UNION, FIRST, SECOND.

Synopsis

raster **ST_MapAlgebraExpr**(raster rast1, raster rast2, text expression, text pixeltype=same_as_rast1_band, text extenttype=INTERSEC
text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);
raster **ST_MapAlgebraExpr**(raster rast1, integer band1, raster rast2, integer band2, text expression, text pixeltype=same_as_rast1_band
text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);

Description



Warning
ST_MapAlgebraExpr is deprecated as of 2.1.0. Use ST_MapAlgebra (expression version) instead.

Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation to the two bands defined by the `expression` on the two input raster bands `rast1`, (`rast2`). If no `band1`, `band2` is specified band 1 is assumed. The resulting raster will be aligned (scale, skew and pixel corners) on the grid defined by the first raster. The resulting raster will have the extent defined by the `extenttype` parameter.

`expression` A PostgreSQL algebraic expression involving the two rasters and PostgreSQL defined functions/operators that will define the pixel value when pixels intersect. e.g. (([rast1] + [rast2])/2.0)::integer

`pixeltype` The resulting pixel type of the output raster. Must be one listed in ST_BandPixelType, left out or set to NULL. If not passed in or set to NULL, will default to the pixeltype of the first raster.

`extenttype` Controls the extent of resulting raster

1. `INTERSECTION` - The extent of the new raster is the intersection of the two rasters. This is the default.

2. UNION - The extent of the new raster is the union of the two rasters.
3. FIRST - The extent of the new raster is the same as the one of the first raster.
4. SECOND - The extent of the new raster is the same as the one of the second raster.

nodata1expr An algebraic expression involving only rast2 or a constant that defines what to return when pixels of rast1 are nodata values and spatially corresponding rast2 pixels have values.

nodata2expr An algebraic expression involving only rast1 or a constant that defines what to return when pixels of rast2 are nodata values and spatially corresponding rast1 pixels have values.

nodatanodataval A numeric constant to return when spatially corresponding rast1 and rast2 pixels are both nodata values.

If pixeltype is passed in, then the new raster will have a band of that pixeltype. If pixeltype is passed NULL or no pixel type specified, then the new raster band will have the same pixeltype as the input rast1 band.

Use the term [rast1.val] [rast2.val] to refer to the pixel value of the original raster bands and [rast1.x], [rast1.y] etc. to refer to the column / row positions of the pixels.

Availability: 2.0.0

Examples

2 Band Intersection and Union.

Create a new 1 band raster from our original that is a function of modulo 2 of the original raster band.

Create a cool set of rasters.

Insert some cool shapes around Boston in Massachusetts state plane meters, then map their intersection and union.

Code

```
DROP TABLE IF EXISTS fun_shapes;
CREATE TABLE fun_shapes(rid serial PRIMARY KEY, fun_name text, rast raster);

INSERT INTO fun_shapes(fun_name, rast)
VALUES ('ref', ST_AsRaster(ST_MakeEnvelope(235229, 899970, 237229, 901930, 26986), 200, 200, ↩
    '8BUI', 0, 0));

INSERT INTO fun_shapes(fun_name, rast)
WITH ref(rast) AS (SELECT rast FROM fun_shapes WHERE fun_name = 'ref' )
SELECT 'area' AS fun_name, ST_AsRaster(ST_Buffer(ST_SetSRID(ST_Point(236229, 900930), 26986) ↩
    , 1000),
            ref.rast, '8BUI', 10, 0) As rast
FROM ref
UNION ALL
SELECT 'rand bubbles',
       ST_AsRaster(
           (
               SELECT ST_Collect(geom)
               FROM (
                   SELECT ST_Buffer(
                       ST_SetSRID(ST_Point(236229 + i*random()*100, 900930 + j*random()*100) ↩
    , 26986),
                       random()*20
                   ) AS geom
                   FROM generate_series(1, 10) AS i,
                        generate_series(1, 10) AS j
               ) AS foo
           ),
           ref.rast, '8BUI', 200, 0
       )
```

```
FROM ref;

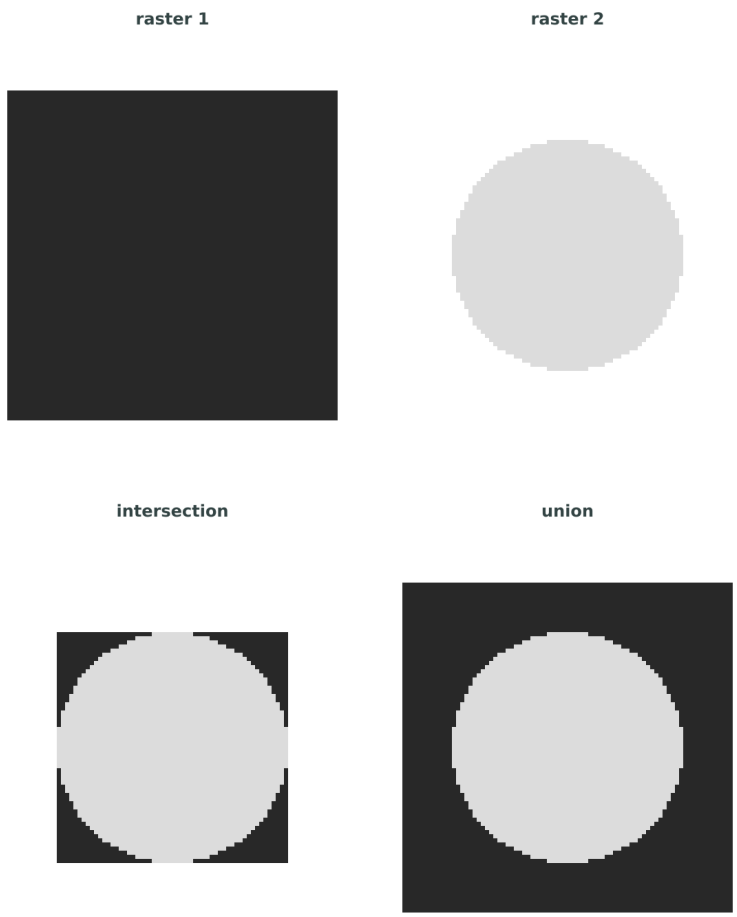
SELECT  ST_MapAlgebraExpr(area.rast, bub.rast, '[rast2.val]', '8BUI', 'INTERSECTION', '[ ↩
    rast2.val]', '[rast1.val]') As interrast,
        ST_MapAlgebraExpr(area.rast, bub.rast, '[rast2.val]', '8BUI', 'UNION', '[rast2.val]' ↩
    , '[rast1.val]') As unionrast
FROM
  (SELECT rast FROM fun_shapes WHERE fun_name = 'area') As area
CROSS JOIN
  (SELECT rast FROM fun_shapes WHERE fun_name = 'rand bubbles') As bub
```

The same intersection and union operations using two deterministic, self-contained rasters.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(80, 80, 0, 80, 1, -1, 0, 0, 0),
        1, '8BUI', 40, 0
    ) AS rast
), inputs AS (
    SELECT
        rast AS raster_1,
        ST_AsRaster(
            ST_Buffer(ST_Point(40, 40), 28),
            rast,
            '8BUI', 220, 0
        ) AS raster_2
    FROM canvas
), variants AS (
    SELECT 'raster 1' AS title, raster_1 AS rendered FROM inputs
    UNION ALL
    SELECT 'raster 2', raster_2 FROM inputs
    UNION ALL
    SELECT 'intersection', ST_MapAlgebraExpr(
        raster_1, raster_2,
        '[rast2.val]', '8BUI', 'INTERSECTION',
        '[rast2.val]', '[rast1.val]'
    ) FROM inputs
    UNION ALL
    SELECT 'union', ST_MapAlgebraExpr(
        raster_1, raster_2,
        '[rast2.val]', '8BUI', 'UNION',
        '[rast2.val]', '[rast1.val]'
    ) FROM inputs
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY CASE title
    WHEN 'raster 1' THEN 1
    WHEN 'raster 2' THEN 2
    WHEN 'intersection' THEN 3
    ELSE 4
END;
```

Figure



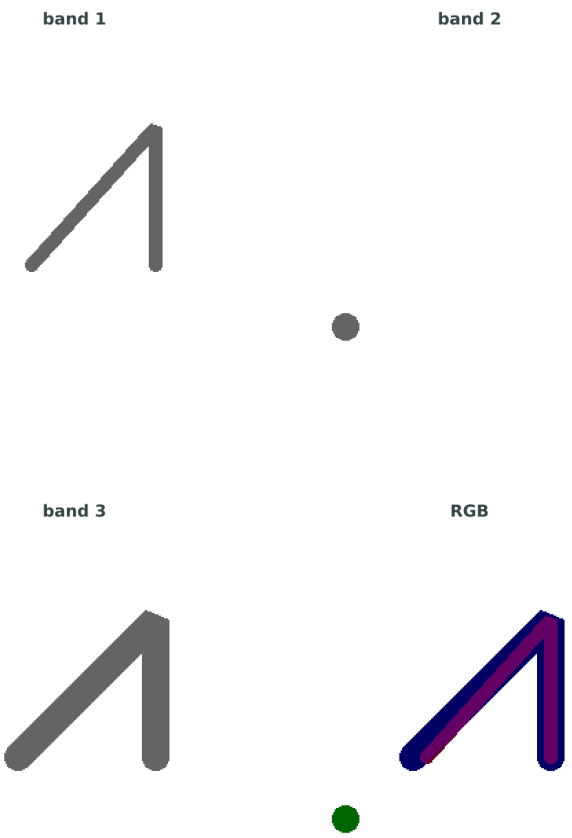
Overlaying rasters on a one-unit-per-pixel canvas as separate bands. We use ST_AsPNG to render the image so all single band ones look grey.

Code

```
WITH mygeoms AS (
    SELECT 2 AS bnum, ST_Buffer(ST_Point(1, 5), 10) AS geom
    UNION ALL
    SELECT 3,
        ST_Buffer(
            ST_GeomFromText('LINESTRING(50 50,150 150,150 50)'),
            10,
            'join=bevel'
        )
    UNION ALL
    SELECT 1,
        ST_Buffer(
            ST_GeomFromText('LINESTRING(60 50,150 150,150 50)'),
            5,
            'join=bevel'
        )
), canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(
```

```
            200,
            200,
            ST_XMin(e)::integer,
            ST_YMax(e)::integer,
            1, -1, 0, 0),
        '8BUI'::text,
        0
    ) AS rast
    FROM (
      SELECT ST_Extent(geom) AS e
      FROM mygeoms
    ) AS bounds
), rbands AS (
    SELECT ARRAY(
        SELECT ST_MapAlgebraExpr(
            canvas.rast,
            ST_AsRaster(m.geom, canvas.rast, '8BUI', 100),
            '[rast2.val]', '8BUI', 'FIRST',
            '[rast2.val]', '[rast1.val]'
        )
        FROM mygeoms AS m
        CROSS JOIN canvas
        ORDER BY m.bnum
    ) AS rasts
)
SELECT title, ST_AsPNG(rendered) AS image
FROM rbands
CROSS JOIN LATERAL (VALUES
    ('band 1', rasts[1]),
    ('band 2', rasts[2]),
    ('band 3', rasts[3]),
    ('RGB', ST_AddBand(ST_AddBand(rasts[1], rasts[2]), rasts[3]))
) AS variants(title, rendered);
```

Figure



Overlay a two-meter boundary of selected parcels over aerial imagery. The query first unions the parcel geometries, clips the source raster shards to the region, unions those smaller shards, and finally adds the first two bands plus a third-band map algebra overlay.

Create a new three-band raster composed of the first two clipped bands and the third band overlaid with the geometry. This query took 3.6 seconds on a 64-bit PostGIS Windows installation.

Code

```
WITH pr AS (
    SELECT ST_Clip(rast, ST_Expand(geom, 50)) AS rast, g.geom
    FROM aerials.o_2_boston AS r
    INNER JOIN (
        SELECT ST_Union(ST_Transform(geom, 26986)) AS geom
        FROM landparcels
        WHERE pid IN ('0303890000', '0303900000')
    ) AS g
        ON ST_Intersects(rast::geometry, ST_Expand(g.geom, 50))
), prunion AS (
    SELECT ST_AddBand(
        NULL,
        ARRAY[
            ST_Union(rast, 1),
            ST_Union(rast, 2),
```

```
            ST_Union(rast, 3)
        ]
    ) AS clipped,
    geom
    FROM pr
    GROUP BY geom
)
SELECT ST_AddBand(
    ST_Band(clipped, ARRAY[1, 2]),
    ST_MapAlgebraExpr(
     ST_Band(clipped, 3),
     ST_AsRaster(ST_Buffer(ST_Boundary(geom), 2), clipped, '8BUI', 250),
     '[rast2.val]',
     '8BUI',
     'FIRST',
     '[rast2.val]',
     '[rast1.val]') ) As rast
FROM prunion;
```

Parcel-boundary overlay.



The blue lines are the boundaries of selected parcels

The generated four-panel canvas example above shows the same band-overlay technique without requiring an external aerial-imagery or parcel table.

See Also

ST_MapAlgebraExpr, ST_AddBand, ST_AsPNG, ST_AsRaster, ST_MapAlgebraFct, ST_BandPixelType, ST_GeoReference, ST_Value, ST_Union, ST_Union

11.12.9 ST_MapAlgebraFct

ST_MapAlgebraFct — 1 band version - Creates a new one band raster formed by applying a valid PostgreSQL function on the input raster band and of pixeltype provided. Band 1 is assumed if no band is specified.

Synopsis

raster **ST_MapAlgebraFct**(raster rast, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, regprocedure onerasteruserfunc, text[] VARIADIC args);
raster **ST_MapAlgebraFct**(raster rast, text pixeltype, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, text pixeltype, regprocedure onerasteruserfunc, text[] VARIADIC args);
raster **ST_MapAlgebraFct**(raster rast, integer band, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, integer band, regprocedure onerasteruserfunc, text[] VARIADIC args);
raster **ST_MapAlgebraFct**(raster rast, integer band, text pixeltype, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, integer band, text pixeltype, regprocedure onerasteruserfunc, text[] VARIADIC args);

Description



Warning
ST_MapAlgebraFct is deprecated as of 2.1.0. Use ST_MapAlgebra (callback function version) instead.

Creates a new one band raster formed by applying a valid PostgreSQL function specified by the `onerasteruserfunc` on the input raster (`rast`). If no `band` is specified, band 1 is assumed. The new raster will have the same georeference, width, and height as the original raster but will only have one band.

If `pixeltype` is passed in, then the new raster will have a band of that pixeltype. If pixeltype is passed NULL, then the new raster band will have the same pixeltype as the input `rast` band.

The `onerasteruserfunc` parameter must be the name and signature of a SQL or PL/pgSQL function, cast to a regprocedure. A very simple and quite useless PL/pgSQL function example is:

Code

```
CREATE OR REPLACE FUNCTION simple_function(pixel FLOAT, pos INTEGER[], VARIADIC args TEXT[])
RETURNS FLOAT
AS $$ BEGIN
    RETURN 0.0;
END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
```

The `userfunction` may accept two or three arguments: a float value, an optional integer array, and a variadic text array. The first argument is the value of an individual raster cell (regardless of the raster datatype). The second argument is the position of the current processing cell in the form '{x,y}'. The third argument indicates that all remaining parameters to ST_MapAlgebraFct shall be passed through to the `userfunction`.

Passing a regprodedure argument to a SQL function requires the full function signature to be passed, then cast to a regprocedure type. To pass the above example PL/pgSQL function as an argument, the SQL for the argument is:

Code

```
'simple_function(float,integer[],text[])'::regprocedure
```

Note that the argument contains the name of the function, the types of the function arguments, quotes around the name and argument types, and a cast to a regprocedure.

The third argument to the `userfunction` is a variadic text array. All trailing text arguments to any ST_MapAlgebraFct call are passed through to the specified `userfunction`, and are contained in the `args` argument.



Note
For more information about the VARIADIC keyword, please refer to the PostgreSQL documentation and the "SQL Functions with Variable Numbers of Arguments" section of Query Language (SQL) Functions.



Note
The text[] argument to the `userfunction` is required, regardless of whether you choose to pass any arguments to your user function for processing or not.

Availability: 2.0.0

Examples

Create a new 1 band raster from our original that is a function of modulo 2 of the original raster band.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast raster;
CREATE FUNCTION mod_fct(pixel float, pos integer[], variadic args text[])
RETURNS float
AS $$
BEGIN
    RETURN pixel::integer % 2;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

UPDATE dummy_rast SET map_rast = ST_MapAlgebraFct(rast, NULL, 'mod_fct(float,integer[],text ↩
    [])'::regprocedure) WHERE rid = 2;

SELECT ST_Value(rast, 1, i, j) As origval, ST_Value(map_rast, 1, i, j) As mapval
FROM dummy_rast CROSS JOIN generate_series(1, 3) AS i CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
 origval | mapval
---------+--------
     253 |      1
     254 |      0
     253 |      1
     253 |      1
     254 |      0
     254 |      0
     250 |      0
     254 |      0
     254 |      0
```

Create a new 1 band raster of pixel-type 2BUI from our original that is reclassified and set the nodata value to a passed parameter to the user function (0).

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast2 raster;
CREATE FUNCTION classify_fct(pixel float, pos integer[], variadic args text[])
RETURNS float
AS
$$
DECLARE
    nodata float := 0;
BEGIN
    IF NOT args[1] IS NULL THEN
        nodata := args[1];
    END IF;
    IF pixel < 251 THEN
        RETURN 1;
```

```
    ELSIF pixel = 252 THEN
        RETURN 2;
    ELSIF pixel > 252 THEN
        RETURN 3;
    ELSE
        RETURN nodata;
    END IF;
END;
$$
LANGUAGE 'plpgsql';
UPDATE dummy_rast SET map_rast2 = ST_MapAlgebraFct(rast,'2BUI','classify_fct(float,integer ↩
    [],text[])'::regprocedure, '0') WHERE rid = 2;

SELECT DISTINCT ST_Value(rast,1,i,j) As origval, ST_Value(map_rast2, 1, i, j) As mapval
FROM dummy_rast CROSS JOIN generate_series(1, 5) AS i CROSS JOIN generate_series(1,5) AS j
WHERE rid = 2;
```

Output

```
 origval | mapval
---------+--------
     249 |      1
     250 |      1
     251 |
     252 |      2
     253 |      3
     254 |      3
```

Code

```
SELECT ST_BandPixelType(map_rast2) As b1pixtyp
FROM dummy_rast WHERE rid = 2;
```

Output

```
 b1pixtyp
----------
 2BUI
```

Create a new 3 band raster same pixel type from our original 3 band raster with first band altered by map algebra and remaining 2 bands unaltered.

Code

```
CREATE FUNCTION rast_plus_tan(pixel float, pos integer[], variadic args text[])
RETURNS float
AS
$$
BEGIN
    RETURN tan(pixel) * pixel;
END;
$$
LANGUAGE 'plpgsql';

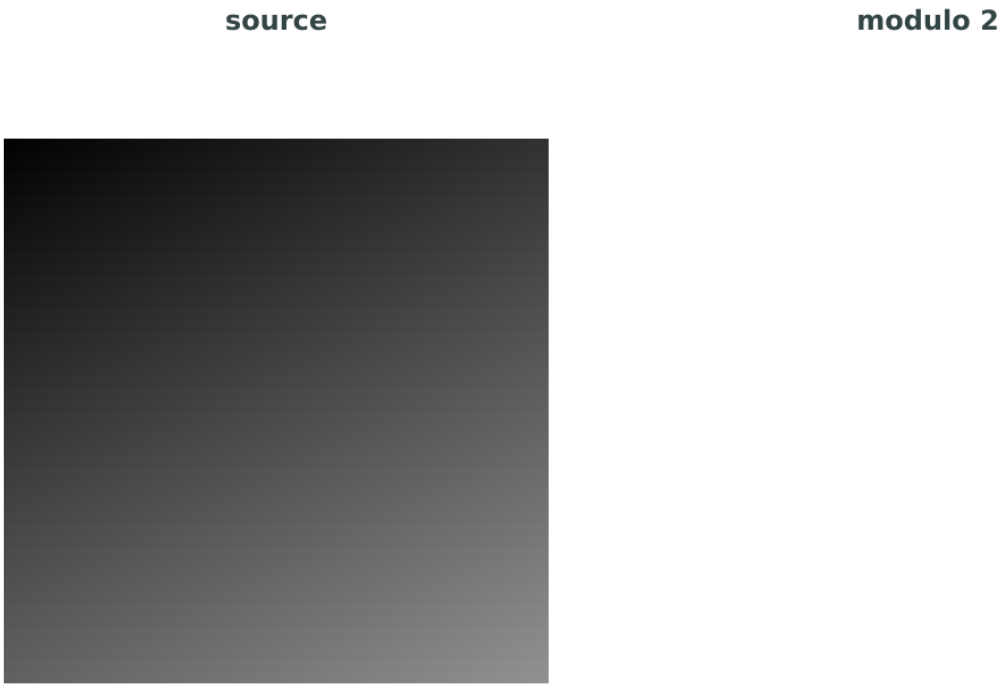
SELECT ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(rast_view),
            ST_MapAlgebraFct(rast_view, 1, NULL, 'rast_plus_tan(float,integer[],text[])':: ↩
    regprocedure)
        ),
        ST_Band(rast_view, 2)
    ),
    ST_Band(rast_view, 3) As rast_view_ma
)
FROM wind
WHERE rid=167;
```

The deprecated callback form above is equivalent to a modern ST_MapAlgebra (callback function version) expression for simple per-pixel transforms. This build-time example renders the original raster and a modulo-2 classification.

Code

```
WITH pixels AS (
    SELECT x, y, (x + y * 2)::double precision AS value
    FROM generate_series(1, 48) AS x
    CROSS JOIN generate_series(1, 48) AS y
), rows AS (
    SELECT y, array_agg(value ORDER BY x) AS values
    FROM pixels
    GROUP BY y
), source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(48, 48, 0, 48, 1, -1, 0, 0, 0),
            1, '8BUI', 0, 0
        ),
        1, 1, 1, array_agg(values ORDER BY y)::double precision[][]
    ) AS rast
    FROM rows
), variants AS (
    SELECT 'source' AS title, rast AS rendered FROM source
    UNION ALL
    SELECT 'modulo 2', ST_MapAlgebra(
        rast, 1,
        '8BUI',
        'mod([rast]::integer,2) * 255'
    )
    FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

Figure



See Also

ST_MapAlgebraExpr, ST_BandPixelType, ST_GeoReference, ST_SetValue

11.12.10 ST_MapAlgebraFct

ST_MapAlgebraFct — 2 band version - Creates a new one band raster formed by applying a valid PostgreSQL function on the 2 input raster bands and of pixeltype provided. Band 1 is assumed if no band is specified. Extent type defaults to INTERSECTION if not specified.

Synopsis

raster **ST_MapAlgebraFct**(raster rast1, raster rast2, regprocedure tworastuserfunc, text pixeltype=same_as_rast1, text extenttype=INTERSECTION, text[] VARIADIC userargs);
raster **ST_MapAlgebraFct**(raster rast1, integer band1, raster rast2, integer band2, regprocedure tworastuserfunc, text pixeltype=same_as_rast1, text extenttype=INTERSECTION, text[] VARIADIC userargs);

Description



Warning
ST_MapAlgebraFct is deprecated as of 2.1.0. Use ST_MapAlgebra (callback function version) instead.

Creates a new one band raster formed by applying a valid PostgreSQL function specified by the `tworastuserfunc` on the input raster `rast1`, `rast2`. If no `band1` or `band2` is specified, band 1 is assumed. The new raster will have the same georeference, width, and height as the original rasters but will only have one band.

If `pixeltype` is passed in, then the new raster will have a band of that pixeltype. If pixeltype is passed NULL or left out, then the new raster band will have the same pixeltype as the input `rast1` band.

The `tworastuserfunc` parameter must be the name and signature of an SQL or PL/pgSQL function, cast to a regprocedure. An example PL/pgSQL function example is:

Code

```
CREATE OR REPLACE FUNCTION simple_function_for_two_rasters(pixel1 FLOAT, pixel2 FLOAT, pos ↩
    INTEGER[], VARIADIC args TEXT[])
RETURNS FLOAT
AS $$ BEGIN
    RETURN 0.0;
END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
```

The `tworastuserfunc` may accept three or four arguments: a double precision value, a double precision value, an optional integer array, and a variadic text array. The first argument is the value of an individual raster cell in `rast1` (regardless of the raster datatype). The second argument is an individual raster cell value in `rast2`. The third argument is the position of the current processing cell in the form '{x,y}'. The fourth argument indicates that all remaining parameters to ST_MapAlgebraFct shall be passed through to the `tworastuserfunc`.

Passing a regprodedure argument to a SQL function requires the full function signature to be passed, then cast to a regprocedure type. To pass the above example PL/pgSQL function as an argument, the SQL for the argument is:

Code

```
'simple_function(double precision, double precision, integer[], text[])'::regprocedure
```

Note that the argument contains the name of the function, the types of the function arguments, quotes around the name and argument types, and a cast to a regprocedure.

The fourth argument to the `tworastuserfunc` is a variadic text array. All trailing text arguments to any ST_MapAlgebraFct call are passed through to the specified `tworastuserfunc`, and are contained in the `userargs` argument.



Note
For more information about the VARIADIC keyword, please refer to the PostgreSQL documentation and the "SQL Functions with Variable Numbers of Arguments" section of Query Language (SQL) Functions.



Note
The text[] argument to the `tworastuserfunc` is required, regardless of whether you choose to pass any arguments to your user function for processing or not.

Availability: 2.0.0

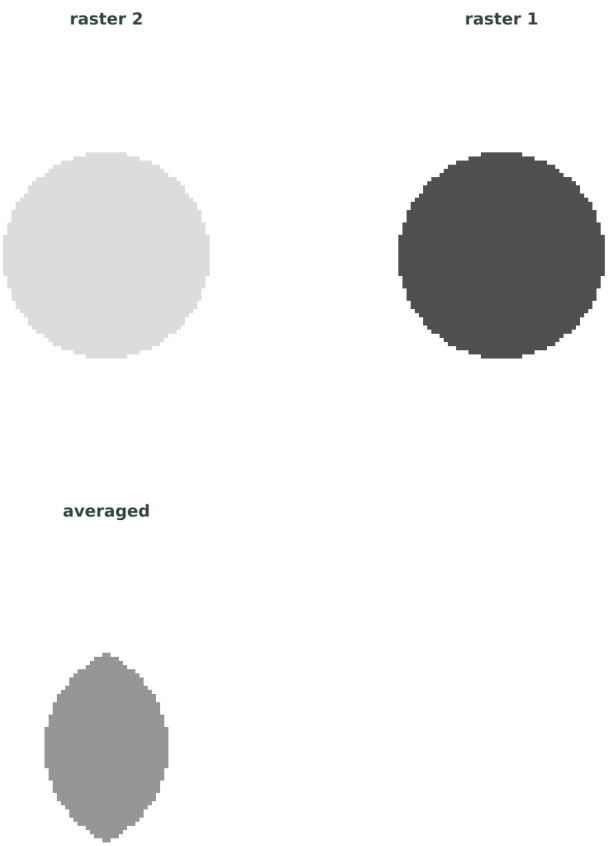
Example: Overlaying rasters on a canvas as separate bands

The deprecated two-raster callback form can often be expressed with the modern two-raster ST_MapAlgebra (callback function version) expression variant. This build-time example renders two input rasters and their averaged overlap.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(80, 80, 0, 80, 1, -1, 0, 0, 0),
        1, '8BUI', 0, 0
    ) AS rast
), inputs AS (
    SELECT
        ST_AsRaster(ST_Buffer(ST_Point(30, 40), 25), rast, '8BUI', 80, 0)
            AS raster_1,
        ST_AsRaster(ST_Buffer(ST_Point(50, 40), 25), rast, '8BUI', 220, 0)
            AS raster_2
    FROM canvas
), variants AS (
    SELECT 'raster 1' AS title, raster_1 AS rendered FROM inputs
    UNION ALL
    SELECT 'raster 2', raster_2 FROM inputs
    UNION ALL
    SELECT 'averaged', ST_MapAlgebra(
        raster_1, 1,
        raster_2, 1,
        '([rast1] + [rast2]) / 2',
        '8BUI',
        'UNION'
    ) FROM inputs
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

Figure



Define our user defined function.

Code

```
CREATE OR REPLACE FUNCTION raster_mapalgebra_union(
    rast1 double precision,
    rast2 double precision,
    pos integer[],
    VARIADIC userargs text[]
)
    RETURNS double precision
    AS $$
    DECLARE
    BEGIN
        CASE
            WHEN rast1 IS NOT NULL AND rast2 IS NOT NULL THEN
                RETURN ((rast1 + rast2)/2.);
            WHEN rast1 IS NULL AND rast2 IS NULL THEN
                RETURN NULL;
            WHEN rast1 IS NULL THEN
                RETURN rast2;
            ELSE
```

```
                RETURN rast1;
        END CASE;

        RETURN NULL;
    END;
    $$ LANGUAGE 'plpgsql' IMMUTABLE COST 1000;
```

Prepare a test table of rasters.

Code

```
DROP TABLE IF EXISTS map_shapes;
CREATE TABLE map_shapes(rid serial PRIMARY KEY, rast raster, bnum integer, descrip text);
INSERT INTO map_shapes(rast,bnum, descrip)
WITH mygeoms
    AS ( SELECT 2 As bnum, ST_Buffer(ST_Point(90,90),30) As geom, 'circle' As descrip
            UNION ALL
            SELECT 3 AS bnum,
                ST_Buffer(ST_GeomFromText('LINESTRING(50 50,150 150,150 50)'), 15) As geom,  ↩
    'big road' As descrip
            UNION ALL
            SELECT 1 As bnum,
                ST_Translate(ST_Buffer(ST_GeomFromText('LINESTRING(60 50,150 150,150 50)'),  ↩
    8,'join=bevel'), 10,-6) As geom, 'small road' As descrip
            ),
   -- define our canvas to be 1 to 1 pixel to geometry
   canvas
    AS ( SELECT ST_AddBand(
        ST_MakeEmptyRaster(
            250,
            250,
            ST_XMin(e)::integer,
            ST_YMax(e)::integer,
            1, -1, 0, 0),
        '8BUI'::text,
        0) As rast
        FROM (SELECT ST_Extent(geom) As e,
                    Max(ST_SRID(geom)) As srid
                    from mygeoms
                    ) As foo
            )
-- return our rasters aligned with our canvas
SELECT ST_AsRaster(m.geom, canvas.rast, '8BUI', 240) As rast, bnum, descrip
                FROM mygeoms AS m CROSS JOIN canvas
UNION ALL
SELECT canvas.rast, 4, 'canvas'
FROM canvas;
```

Map algebra on single-band rasters and then collect the result with ST_AddBand.

Code

```
INSERT INTO map_shapes(rast,bnum,descrip)
SELECT ST_AddBand(ST_AddBand(rasts[1], rasts[2]),rasts[3]), 4, 'map bands overlay fct union  ↩
    (canvas)'
    FROM (SELECT ARRAY(SELECT ST_MapAlgebraFct(
            m1.rast,
            m2.rast,
            'raster_mapalgebra_union(double precision, double precision, integer[], text[])' ↩
    ::regprocedure,
```

```
            '8BUI',
            'FIRST')
                FROM map_shapes As m1 CROSS JOIN map_shapes As m2
    WHERE m1.descrip = 'canvas' AND m2.descrip <> 'canvas' ORDER BY m2.bnum) As rasts) As  ↩
    foo;
```

A compact, executable callback example. The source rasters and their averaged overlap are rendered at build time.

Code

```
CREATE OR REPLACE FUNCTION pg_temp.raster_average(
    value_1 double precision,
    value_2 double precision,
    position integer[],
    VARIADIC userargs text[]
) RETURNS double precision
LANGUAGE SQL IMMUTABLE
AS 'SELECT COALESCE(($1 + $2) / 2.0, $1, $2)';

WITH canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(80, 80, 0, 80, 1, -1, 0, 0, 0),
        1, '8BUI', 0, 0
    ) AS rast
), inputs AS (
    SELECT
        ST_AsRaster(ST_Buffer(ST_Point(30, 40), 25), rast, '8BUI', 80, 0)
            AS raster_1,
        ST_AsRaster(ST_Buffer(ST_Point(50, 40), 25), rast, '8BUI', 220, 0)
            AS raster_2
    FROM canvas
), variants AS (
    SELECT 'raster 1' AS title, raster_1 AS rendered FROM inputs
    UNION ALL
    SELECT 'raster 2', raster_2 FROM inputs
    UNION ALL
    SELECT 'averaged', ST_MapAlgebraFct(
        raster_1,
        raster_2,
        'pg_temp.raster_average(double precision,double precision,integer[],text[])':: ↩
    regprocedure,
        '8BUI',
        'UNION'
    ) FROM inputs
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

User Defined function that takes extra args

Code

```
CREATE OR REPLACE FUNCTION raster_mapalgebra_userargs(
    rast1 double precision,
    rast2 double precision,
    pos integer[],
    VARIADIC userargs text[]
)
```

```
    RETURNS double precision
    AS $$
    DECLARE
    BEGIN
        CASE
            WHEN rast1 IS NOT NULL AND rast2 IS NOT NULL THEN
                RETURN least(userargs[1]::integer,(rast1 + rast2)/2.);
            WHEN rast1 IS NULL AND rast2 IS NULL THEN
                RETURN userargs[2]::integer;
            WHEN rast1 IS NULL THEN
                RETURN greatest(rast2, random()*userargs[3]::integer)::integer;
            ELSE
                RETURN greatest(rast1, random()*userargs[4]::integer)::integer;
        END CASE;

        RETURN NULL;
    END;
    $$ LANGUAGE 'plpgsql' VOLATILE COST 1000;

SELECT ST_MapAlgebraFct(
    m1.rast,
    1,
    m1.rast,
    3,
    'raster_mapalgebra_userargs(double precision, double precision, integer[], text[])':: ↩
    regprocedure,
    '8BUI',
    'INTERSECT',
    '100', '200', '200', '0')
                FROM map_shapes As m1
    WHERE m1.descrip = 'map bands overlay fct union (canvas)';
```

The generated callback figure above demonstrates the raster alignment and overlap used by this variant; the exact callback result remains available as text.

See Also

ST_MapAlgebraExpr, ST_BandPixelType, ST_GeoReference, ST_SetValue

11.12.11 ST_MapAlgebraFctNgb

ST_MapAlgebraFctNgb — 1-band version: Map Algebra Nearest Neighbor using user-defined PostgreSQL function. Return a raster which values are the result of a PLPGSQL user function involving a neighborhood of values from the input raster band.

Synopsis

raster **ST_MapAlgebraFctNgb**(raster rast, integer band, text pixeltype, integer ngbwidth, integer ngbheight, regprocedure onerastngbuserfunc, text nodatamode, text[] VARIADIC args);

Description



Warning
ST_MapAlgebraFctNgb is deprecated as of 2.1.0. Use ST_MapAlgebra (callback function version) instead.

(one raster version) Return a raster which values are the result of a PLPGSQL user function involving a neighborhood of values from the input raster band. The user function takes the neighborhood of pixel values as an array of numbers, for each pixel, returns the result from the user function, replacing pixel value of currently inspected pixel with the function result.

rast Raster on which the user function is evaluated.

band Band number of the raster to be evaluated. Default to 1.

pixeltype The resulting pixel type of the output raster. Must be one listed in ST_BandPixelType or left out or set to NULL. If not passed in or set to NULL, will default to the pixeltype of the `rast`. Results are truncated if they are larger than what is allowed for the pixeltype.

ngbwidth The width of the neighborhood, in cells.

ngbheight The height of the neighborhood, in cells.

onerastngbuserfunc PLPGSQL/psql user function to apply to neighborhood pixels of a single band of a raster. The first element is a 2-dimensional array of numbers representing the rectangular pixel neighborhood

nodatamode Defines what value to pass to the function for a neighborhood pixel that is nodata or NULL

'ignore': any NODATA values encountered in the neighborhood are ignored by the computation -- this flag must be sent to the user callback function, and the user function decides how to ignore it.

'NULL': any NODATA values encountered in the neighborhood will cause the resulting pixel to be NULL -- the user callback function is skipped in this case.

'value': any NODATA values encountered in the neighborhood are replaced by the reference pixel (the one in the center of the neighborhood). Note that if this value is NODATA, the behavior is the same as 'NULL' (for the affected neighborhood)

args Arguments to pass into the user function.

Availability: 2.0.0

Examples

Examples utilize the katrina raster loaded as a single tile described in https://gdal.org/user/drivers/raster/postgisraster.html and then prepared in the ST_Rescale examples

A simple 'callback' user function that averages up all the values in a neighborhood.

Code

```
CREATE OR REPLACE FUNCTION rast_avg(matrix float[][], nodatamode text, variadic args text[])
    RETURNS float AS
    $$
    DECLARE
        _matrix float[][];
        x1 integer;
        x2 integer;
        y1 integer;
        y2 integer;
        sum float;
    BEGIN
        _matrix := matrix;
        sum := 0;
        FOR x in array_lower(matrix, 1)..array_upper(matrix, 1) LOOP
            FOR y in array_lower(matrix, 2)..array_upper(matrix, 2) LOOP
                sum := sum + _matrix[x][y];
            END LOOP;
        END LOOP;
        RETURN (sum*1.0/(array_upper(matrix,1)*array_upper(matrix,2) ))::integer ;
    END;
```

```
    $$
LANGUAGE 'plpgsql' IMMUTABLE COST 1000;
```

Apply the raster averaging callback within two pixels of each input pixel in the X and Y directions.

Code

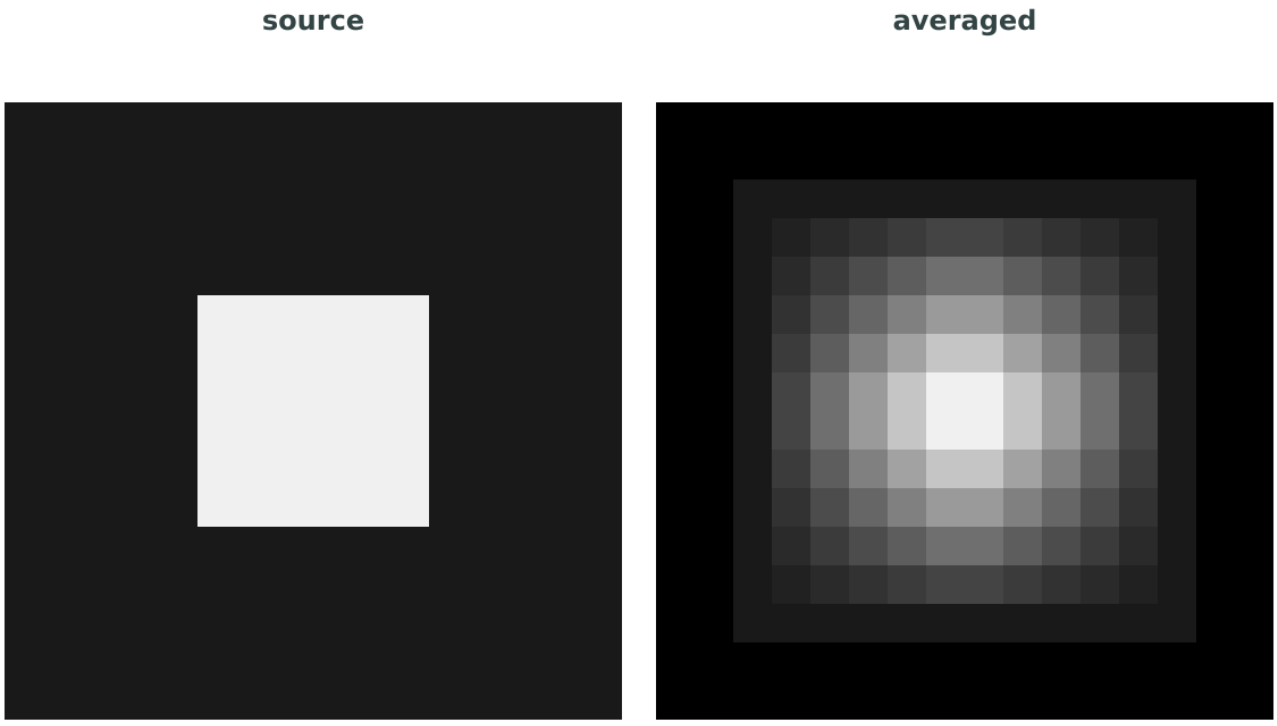
```
SELECT ST_MapAlgebraFctNgb(
        rast,
        1,
        '8BUI',
        4,
        4,
        'rast_avg(float[][], text, text[])'::regprocedure,
        'NULL',
        NULL) As nn_with_border
    FROM katrinas_rescaled
    limit 1;
```

A self-contained neighborhood average. The source raster has a sharp central square, while the result spreads those values across neighboring pixels. The source uses a non-NODATA background so `nodatamode ignore` averages background values with the square values, instead of treating the background as absent.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(16, 16, 0, 16, 1, -1, 0, 0, 0),
            1, '8BUI', 25, NULL
        ),
        1, 6, 6,
        ARRAY[
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240]
        ]::double precision[][]
    ) AS rast
), variants AS (
    SELECT 'source' AS title, rast AS rendered FROM source
    UNION ALL
    SELECT 'averaged', ST_SetBandNoDataValue(ST_MapAlgebraFctNgb(
        rast,
        1,
        '8BUI',
        2,
        2,
        'ST_Mean4ma(float[][],text,text[])'::regprocedure,
        'ignore',
        NULL
    ), NULL) FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

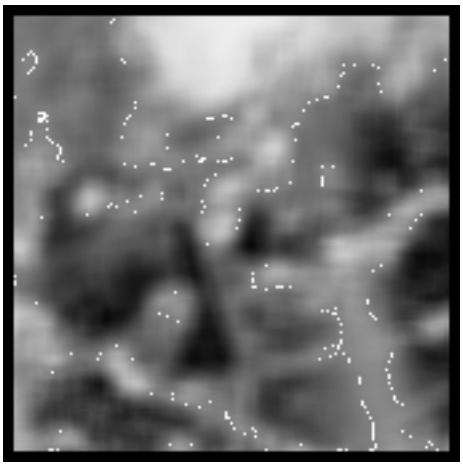
Figure



Neighborhood averaging.



First band of our raster



Averaged result

See Also

ST_MapAlgebraFct, ST_MapAlgebraExpr, ST_Rescale

11.12.12 ST_Reclass

ST_Reclass — Creates a new raster composed of band types reclassified from original. The nband is the band to be changed. If nband is not specified assumed to be 1. All other bands are returned unchanged. Use case: convert a 16BUI band to a 8BUI and so forth for simpler rendering as viewable formats.

Synopsis

raster **ST_Reclass**(raster rast, integer nband, text reclassexpr, text pixeltype, double precision nodataval=NULL);
raster **ST_Reclass**(raster rast, reclassarg[] VARIADIC reclassargset);
raster **ST_Reclass**(raster rast, text reclassexpr, text pixeltype);

Description

Creates a new raster formed by applying a reclassification operation defined by the `reclassexpr` on the input raster (`rast`). Refer to reclassarg for the description of reclassification expressions. If no `band` is specified band 1 is assumed.

The new raster will have the same georeference, width, and height as the original raster. The bands of the new raster have pixel type of `pixeltype`. If `reclassargset` is specified then each `reclassarg` defines the type of the target band. Bands not designated are returned unchanged.

Availability: 2.0.0

Examples

Basic.

Create a new raster from the original where band 2 is converted from 8BUI to 4BUI and all values from 101-254 are set to nodata value.

Code

```
ALTER TABLE dummy_rast ADD COLUMN reclass_rast raster;
UPDATE dummy_rast SET reclass_rast = ST_Reclass(rast, 2, '0-87:1-10, 88-100:11-15,  ↩
    101-254:0-0', '4BUI', 0) WHERE rid = 2;

SELECT i as col, j as row, ST_Value(rast, 2, i, j) As origval,
    ST_Value(reclass_rast, 2, i, j) As reclassval,
    ST_Value(reclass_rast, 2, i, j, false) As reclassval_include_nodata
FROM dummy_rast CROSS JOIN generate_series(1, 3) AS i CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
 col | row | origval | reclassval | reclassval_include_nodata
-----+-----+---------+------------+--------------------------
   1 |   1 |      78 |          9 |                         9
   2 |   1 |      98 |         14 |                        14
   3 |   1 |     122 |            |                         0
   1 |   2 |      96 |         14 |                        14
   2 |   2 |     118 |            |                         0
   3 |   2 |     180 |            |                         0
   1 |   3 |      99 |         15 |                        15
   2 |   3 |     112 |            |                         0
   3 |   3 |     169 |            |                         0
```

Advanced using multiple reclassargs.

Create a new raster from the original where band 1,2,3 is converted to 1BB,4BUI, 4BUI respectively and reclassified. Note this uses the variadic `reclassarg` argument which can take as input an indefinite number of reclassargs (theoretically as many bands as you have)

Code

```
UPDATE dummy_rast SET reclass_rast =
    ST_Reclass(
        rast,
        ROW(2, '0-87]:1-10, (87-100]:11-15, (101-254]:0-0', '4BUI', NULL)::reclassarg,
        ROW(1, '0-253]:1, 254:0', '1BB', NULL)::reclassarg,
        ROW(3, '0-70]:1, (70-86:2, [86-150):3, [150-255:4', '4BUI', NULL)::reclassarg
        ) WHERE rid = 2;

SELECT i as col, j as row, ST_Value(rast, 1, i, j) As ov1, ST_Value(reclass_rast, 1, i, j) ↩
    As rv1,
    ST_Value(rast, 2, i, j) As ov2, ST_Value(reclass_rast, 2, i, j) As rv2,
    ST_Value(rast, 3, i, j) As ov3, ST_Value(reclass_rast, 3, i, j) As rv3
FROM dummy_rast CROSS JOIN generate_series(1, 3) AS i CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
col | row | ov1 | rv1 | ov2 | rv2 | ov3 | rv3
----+-----+-----+-----+-----+-----+-----+-----
  1 |   1 | 253 |   1 |  78 |   9 |  70 |   1
  2 |   1 | 254 |   0 |  98 |  14 |  86 |   3
  3 |   1 | 253 |   1 | 122 |   0 | 100 |   3
  1 |   2 | 253 |   1 |  96 |  14 |  80 |   2
  2 |   2 | 254 |   0 | 118 |   0 | 108 |   3
  3 |   2 | 254 |   0 | 180 |   0 | 162 |   4
  1 |   3 | 250 |   1 |  99 |  15 |  90 |   3
  2 |   3 | 254 |   0 | 112 |   0 | 108 |   3
  3 |   3 | 254 |   0 | 169 |   0 | 175 |   4
```

Advanced Map a single band 32BF raster to multiple viewable bands.

Create a new 3 band (8BUI,8BUI,8BUI viewable raster) from a raster that has only one 32bf band

Code

```
ALTER TABLE wind ADD COLUMN rast_view raster;
UPDATE wind
    set rast_view = ST_AddBand(
        NULL,
        ARRAY[
    ST_Reclass(rast, 1, '0.1-10]:1-10,9-10]:11,(11-33:0'::text, '8BUI'::text, 0),
    ST_Reclass(rast, 1, '11-33):0-255,[0-32:0,(34-1000:0'::text, '8BUI'::text, 0),
    ST_Reclass(rast, 1, '0-32]:0,(32-100:100-255'::text, '8BUI'::text, 0)
    ]
    );
```

See Also

ST_AddBand, ST_Band, ST_BandPixelType, ST_MakeEmptyRaster, reclassarg, ST_Value

11.12.13 ST_ReclassExact

ST_ReclassExact — Creates a new raster composed of bands reclassified from original, using a 1:1 mapping from values in the original band to new values in the destination band.

Synopsis

raster **ST_ReclassExact**(raster rast, double precision[] inputvalues, double precision[] outputvalues, integer bandnumber=1, text pixeltype=32BF, double precision nodatavalue=NULL);

Description

Creates a new raster formed by applying a reclassification operation defined by the `inputvalues` and `outputvalues` arrays. Pixel values found in the input array are mapped to the corresponding value in the output array. All other pixel values are mapped to the `nodatavalue`.

The output pixel type defaults to float, but can be specified using the `pixeltype` parameter. If no `bandnumber` is specified band 1 is assumed.

The new raster will have the same georeference, width, and height as the original raster. Bands not designated are returned unchanged.

Availability: 3.6.0

Example: Basic

Create a small raster and map its pixels to new values.

Create a raster with four pixels: first row 1 2, second row 3 4.

Code

```
CREATE TABLE reclassexact (
    id integer,
    rast raster
);
INSERT INTO reclassexact (id, rast)
SELECT 1, ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(2, -- width in pixels
        2, -- height in pixels
        0, -- upper-left x-coordinate
        0, -- upper-left y-coordinate
        1, -- pixel size in x-direction
        -1, -- pixel size in y-direction (negative for north-up)
        0, -- skew in x-direction
        0, -- skew in y-direction
        4326  -- SRID (e.g., WGS 84)
      ),
      '32BUI'::text, -- pixel type (e.g., '32BF' for float, '8BUI' for unsigned 8-bit int)
      0.0, -- initial value for the band (e.g., 0.0 or a no-data value)
      -99            -- nodatavalue
    ),
    1, -- band number (usually 1 for single-band rasters)
    1, -- x origin for setting values (usually 1)
    1, -- y origin for setting values (usually 1)
    ARRAY[
      ARRAY[1, 2],
      ARRAY[3, 4]
    ]::double precision[][] -- 2D array of values
  );
```

Reclass the values to new values and dump the values of the new raster for display.

Code

```
WITH rc AS (
  SELECT ST_ReclassExact(rast, -- input raster
    ARRAY[4, 3, 2, 1], -- input map
    ARRAY[14, 13, 12, 11], -- output map
    1, -- band number to remap
    '32BUI'              -- output raster pixel type
    ) AS rast
```

```
  FROM reclassexact
  WHERE id = 1
  )
SELECT 'rce-1', (ST_DumpValues(rc.rast)).*
FROM rc;
```

See Also

ST_Reclass, ST_AddBand, ST_Band, ST_MakeEmptyRaster

11.12.14 ST_Union

ST_Union — Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.

Synopsis

raster **ST_Union**(setof raster rast);
raster **ST_Union**(setof raster rast, unionarg[] unionargset);
raster **ST_Union**(setof raster rast, integer nband);
raster **ST_Union**(setof raster rast, text uniontype);
raster **ST_Union**(setof raster rast, integer nband, text uniontype);

Description

Returns the union of a set of raster tiles into a single raster composed of at least one band. The resulting raster's extent is the extent of the whole set. In the case of intersection, the resulting value is defined by `uniontype` which is one of the following: LAST (default), FIRST, MIN, MAX, COUNT, SUM, MEAN, RANGE.



Note
In order for rasters to be unioned, they must all have the same alignment. Use ST_SameAlignment and ST_NotSameAlignmentReason for more details and help. One way to fix alignment issues is to use ST_Resample and use the same reference raster for alignment.

Availability: 2.0.0

Enhanced: 2.1.0 Improved Speed (fully C-Based).

Availability: 2.1.0 ST_Union(rast, unionarg) variant was introduced.

Enhanced: 2.1.0 ST_Union(rast) (variant 1) unions all bands of all input rasters. Prior versions of PostGIS assumed the first band.

Enhanced: 2.1.0 ST_Union(rast, uniontype) (variant 4) unions all bands of all input rasters.

Examples

Reconstitute a single band chunked raster tile.

This creates a single band from first band of raster tiles. That form the original file system tile.

Code

```
SELECT filename, ST_Union(rast, 1) As file_rast
FROM sometable WHERE filename IN('dem01', 'dem02') GROUP BY filename;
```

Return a multi-band raster that is the union of tiles intersecting geometry.

This creates a multi band raster collecting all the tiles that intersect a line. Note that In 2.0, this would have just returned a single band raster. , new union works on all bands by default. This is equivalent to unionarg: ARRAY[ROW(1, 'LAST'), ROW(2, 'LAST'), ROW(3, 'LAST')]::unionarg[].

Code

```
SELECT ST_Union(rast)
FROM aerials.boston
WHERE ST_Intersects(rast, ST_GeomFromText('LINESTRING(230486 887771,230500 88772)', 26986) ) ↩
    ;
```

Select and reorder bands while unioning tiles that intersect a geometry.

Here we use the longer syntax if we only wanted a subset of bands or we want to change order of bands

This creates a multi band raster collecting all the tiles that intersect a line.

Code

```
SELECT ST_Union(rast, ARRAY[ROW(2, 'LAST'), ROW(1, 'LAST'), ROW(3, 'LAST')]::unionarg[])
FROM aerials.boston
WHERE ST_Intersects(rast, ST_GeomFromText('LINESTRING(230486 887771,230500 88772)', 26986) ) ↩
    ;
```

See Also

unionarg, ST_Envelope, ST_ConvexHull, ST_Clip, ST_Union

11.13 Built-in Map Algebra Callback Functions

11.13.1 ST_Distinct4ma

ST_Distinct4ma — Raster processing function that calculates the number of unique pixel values in a neighborhood.

Synopsis

float8 **ST_Distinct4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Distinct4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the number of unique pixel values in a neighborhood of pixels.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Distinct4ma(float[][],text,text[]) ↩
    '::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid | st_value
-----+----------
   2 |        3
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.2 ST_InvDistWeight4ma

ST_InvDistWeight4ma — Raster processing function that interpolates a pixel's value from the pixel's neighborhood.

Synopsis

double precision **ST_InvDistWeight4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate an interpolated value for a pixel using the Inverse Distance Weighted method.

There are two optional parameters that can be passed through `userargs`. The first parameter is the power factor (variable k in the equation below) between 0 and 1 used in the Inverse Distance Weighted equation. If not specified, default value is 1. The second parameter is the weight percentage applied only when the value of the pixel of interest is included with the interpolated value from the neighborhood. If not specified and the pixel of interest has a value, that value is returned.

The basic inverse-distance weighted estimate is: z = (Σᵢ zᵢ / dᵢᵏ) / (Σᵢ 1 / dᵢ&#x1d
Here `zᵢ` is a neighboring pixel value, `dᵢ` is its distance from the pixel being estimated, and `k` is a real-valued power factor between 0 and 1.



Note
This function is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , NULL::double precision, -9999),
        1, 1, 1, ARRAY[
            [10, 10, 10],
            [10, NULL, 20],
            [20, 20, 20]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(
    ST_MapAlgebra(
        rast,
        1,
        'ST_InvDistWeight4ma(double precision[][][],integer[][],text[])'::regprocedure,
        '32BF',
        'FIRST',
        NULL,
        1,
        1
    ),
    2, 2
) AS interpolated_value
FROM foo;
```

Output

```
15
```

See Also

ST_MapAlgebra (callback function version), ST_MinDist4ma

11.13.3 ST_Max4ma

ST_Max4ma — Raster processing function that calculates the maximum pixel value in a neighborhood.

Synopsis

float8 **ST_Max4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Max4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the maximum pixel value in a neighborhood of pixels.

For Variant 2, a substitution value for NODATA pixels can be specified by passing that value to userargs.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Max4ma(float[][],text,text[])':: ↩
    regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid | st_value
-----+----------
   2 |      254
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Sum4ma, ST_Mean4ma, ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.4 ST_Mean4ma

ST_Mean4ma — Raster processing function that calculates the mean pixel value in a neighborhood.

Synopsis

float8 **ST_Mean4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Mean4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the mean pixel value in a neighborhood of pixels.

For Variant 2, a substitution value for NODATA pixels can be specified by passing that value to userargs.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Variant 1.

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, '32BF', 1, 1, 'ST_Mean4ma(float[][],text,text[])' ↩
    ::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid |     st_value
-----+------------------
   2 | 253.222229003906
(1 row)
```

Variant 2.

Code

```
SELECT
  rid,
  ST_Value(
      ST_MapAlgebra(rast, 1, 'ST_Mean4ma(double precision[][][], integer[][], text[])':: ↩
    regprocedure, '32BF', 'FIRST', NULL, 1, 1),
      2,
      2)
FROM dummy_rast
 WHERE rid = 2;
```

Output

```
 rid |     st_value
-----+------------------
   2 | 253.222229003906
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Range4ma, ST_StdDev4ma

11.13.5 ST_Min4ma

ST_Min4ma — Raster processing function that calculates the minimum pixel value in a neighborhood.

Synopsis

float8 **ST_Min4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Min4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the minimum pixel value in a neighborhood of pixels.

For Variant 2, a substitution value for NODATA pixels can be specified by passing that value to userargs.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Min4ma(float[][],text,text[])':: ↩
    regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid | st_value
-----+----------
   2 |      250
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.6 ST_MinDist4ma

ST_MinDist4ma — Raster processing function that returns the minimum distance (in number of pixels) between the pixel of interest and a neighboring pixel with value.

Synopsis

double precision **ST_MinDist4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Return the shortest distance (in number of pixels) between the pixel of interest and the closest pixel with value in the neighborhood.



Note
The intent of this function is to provide an informative data point that helps infer the usefulness of the pixel of interest's interpolated value from ST_InvDistWeight4ma. This function is particularly useful when the neighborhood is sparsely populated.



Note
This function is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , NULL::double precision, -9999),
        1, 1, 1, ARRAY[
            [10, 10, 10],
            [10, NULL, 20],
            [20, 20, 20]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(
    ST_MapAlgebra(
        rast,
        1,
        'ST_MinDist4ma(double precision[][][],integer[][],text[])'::regprocedure,
        '32BF',
        'FIRST',
        NULL,
        1,
        1
    ),
    2, 2
) AS min_distance
FROM foo;
```

Output

```
1
```

See Also

ST_MapAlgebra (callback function version), ST_InvDistWeight4ma

11.13.7 ST_Range4ma

ST_Range4ma — Raster processing function that calculates the range of pixel values in a neighborhood.

Synopsis

float8 **ST_Range4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Range4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the range of pixel values in a neighborhood of pixels.

For Variant 2, a substitution value for NODATA pixels can be specified by passing that value to userargs.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Range4ma(float[][],text,text[])':: ↩
    regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid | st_value
-----+----------
   2 |        4
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.8 ST_StdDev4ma

ST_StdDev4ma — Raster processing function that calculates the standard deviation of pixel values in a neighborhood.

Synopsis

float8 **ST_StdDev4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_StdDev4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the standard deviation of pixel values in a neighborhood of pixels.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, '32BF', 1, 1, 'ST_StdDev4ma(float[][],text,text[]) ↩
    '::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid |     st_value
-----+------------------
   2 | 1.30170822143555
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.9 ST_Sum4ma

ST_Sum4ma — Raster processing function that calculates the sum of all pixel values in a neighborhood.

Synopsis

float8 **ST_Sum4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Sum4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

Calculate the sum of all pixel values in a neighborhood of pixels.

For Variant 2, a substitution value for NODATA pixels can be specified by passing that value to userargs.



Note
Variant 1 is a specialized callback function for use as a callback parameter to ST_MapAlgebraFctNgb.



Note
Variant 2 is a specialized callback function for use as a callback parameter to ST_MapAlgebra (callback function version).



Warning
Use of Variant 1 is discouraged since ST_MapAlgebraFctNgb has been deprecated as of 2.1.0.

Availability: 2.0.0

Enhanced: 2.1.0 Addition of Variant 2

Examples

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, '32BF', 1, 1, 'ST_Sum4ma(float[][],text,text[])':: ↩
    regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
 rid | st_value
-----+----------
   2 |     2279
(1 row)
```

See Also

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Mean4ma, ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.14 Raster Processing: DEM (Elevation)

11.14.1 ST_Aspect

ST_Aspect — Returns the aspect (in degrees by default) of an elevation raster band. Useful for analyzing terrain.

Synopsis

raster **ST_Aspect**(raster rast, integer band=1, text pixeltype=32BF, text units=DEGREES, boolean interpolate_nodata=FALSE);
raster **ST_Aspect**(raster rast, integer band, raster customextent, text pixeltype=32BF, text units=DEGREES, boolean interpolate_nodata=FALSE);

Description

Returns the aspect (in degrees by default) of an elevation raster band. Utilizes map algebra and applies the aspect equation to neighboring pixels.

`units` indicates the units of the aspect. Possible values are: RADIANS, DEGREES (default).

When `units` = RADIANS, values are between 0 and 2 * pi radians measured clockwise from North.

When `units` = DEGREES, values are between 0 and 360 degrees measured clockwise from North.

If slope of pixel is zero, aspect of pixel is -1.



Note
For more information about Slope, Aspect and Hillshade, please refer to ESRI - How hillshade works and ERDAS Field Guide - Aspect Images.

Availability: 2.0.0

Enhanced: 2.1.0 Uses ST_MapAlgebra() and added optional `interpolate_nodata` function parameter

Changed: 2.1.0 In prior versions, return values were in radians. Now, return values default to degrees

Examples

Variant 1.

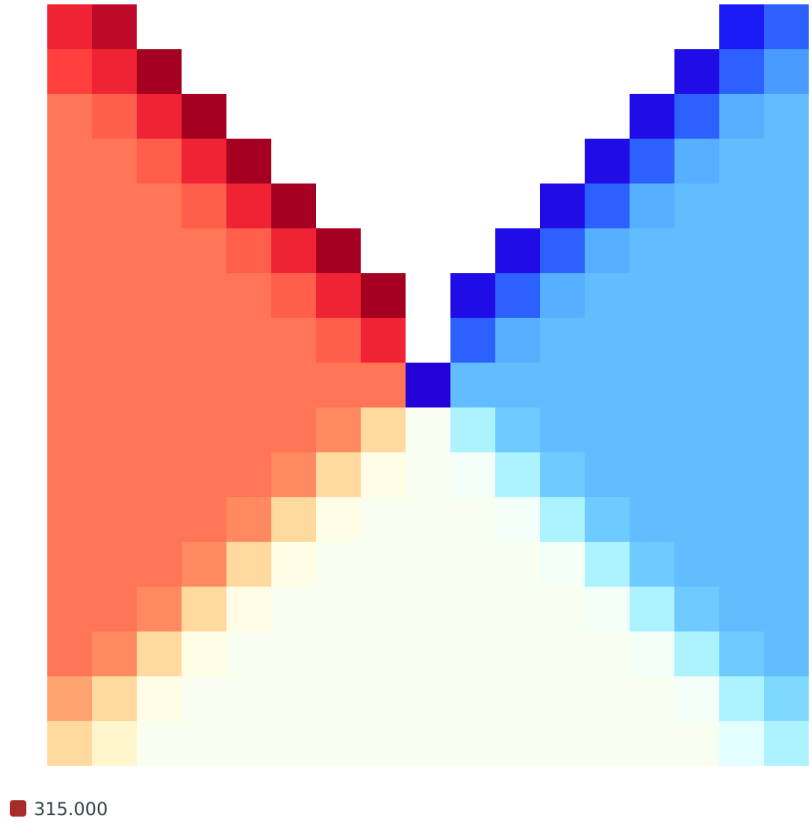
This symmetric peak highlights the expected compass directions at the corners and the `-1` aspect at the summit. The generated PNG shows the full aspect surface without relying on a manual illustration. The generated source raster is large enough for the PNG to show the directional field around the summit.

Code

```
WITH pixels AS (
    SELECT
        x,
        y,
        (20 - greatest(abs(x - 9), abs(y - 9)))::double precision AS value
    FROM generate_series(1, 17) AS y
    CROSS JOIN generate_series(1, 17) AS x
), rows AS (
    SELECT
        y,
        array_agg(value ORDER BY x) AS row_values
    FROM pixels
    GROUP BY y
), source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(17, 17, 0, 0, 1, -1, 0, 0, 0),
            1, '32BF', 0, -9999
        ),
        1, 1, 1,
```

```
        array_agg(row_values ORDER BY y)::double precision[][]
    ) AS rast
    FROM rows
), aspect AS (
    SELECT ST_Aspect(rast, 1, '32BF') AS rast
    FROM source
)
SELECT
    round(ST_Value(rast, 1, 1, 1)::numeric, 3) AS northwest_degrees,
    round(ST_Value(rast, 1, 9, 9)::numeric, 3) AS summit_degrees,
    round(ST_Value(rast, 1, 17, 17)::numeric, 3) AS southeast_degrees,
    ST_AsPNG(ST_ColorMap(rast, 1, 'bluered')) AS image
FROM aspect;
```

Figure



Variant 2.

Complete example of tiles of a coverage.

Code

```
WITH foo AS (
    SELECT ST_Tile(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(6, 6, 0, 0, 1, -1, 0, 0, 0),
                1, '32BF', 0, -9999
            ),
            1, 1, 1, ARRAY[
                [1, 1, 1, 1, 1, 1],
                [1, 1, 1, 1, 2, 1],
                [1, 2, 2, 3, 3, 1],
                [1, 1, 3, 2, 1, 1],
                [1, 2, 2, 1, 2, 1],
```

```
                [1, 1, 1, 1, 1, 1]
            ]::double precision[]
        ),
        2, 2
    ) AS rast
)
SELECT
    t1.rast,
    ST_Aspect(ST_Union(t2.rast), 1, t1.rast)
FROM foo t1
CROSS JOIN foo t2
WHERE ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rast;
```

See Also

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Roughness, ST_HillShade, ST_Slope

11.14.2 ST_HillShade

ST_HillShade — Returns the hypothetical illumination of an elevation raster band using provided azimuth, altitude, brightness and scale inputs.

Synopsis

raster **ST_HillShade**(raster rast, integer band=1, text pixeltype=32BF, double precision azimuth=315, double precision altitude=45, double precision max_bright=255, double precision scale=1.0, boolean interpolate_nodata=FALSE);
raster **ST_HillShade**(raster rast, integer band, raster customextent, text pixeltype=32BF, double precision azimuth=315, double precision altitude=45, double precision max_bright=255, double precision scale=1.0, boolean interpolate_nodata=FALSE);

Description

Returns the hypothetical illumination of an elevation raster band using the azimuth, altitude, brightness, and scale inputs. Utilizes map algebra and applies the hill shade equation to neighboring pixels. Return pixel values are between 0 and 255.

`azimuth` is a value between 0 and 360 degrees measured clockwise from North.

`altitude` is a value between 0 and 90 degrees where 0 degrees is at the horizon and 90 degrees is directly overhead.

`max_bright` is a value between 0 and 255 with 0 as no brightness and 255 as max brightness.

`scale` is the ratio of vertical units to horizontal. For Feet:LatLon use scale=370400, for Meters:LatLon use scale=111120.

If `interpolate_nodata` is TRUE, values for NODATA pixels from the input raster will be interpolated using ST_InvDistWeight4ma before computing the hillshade illumination.



Note
For more information about Hillshade, please refer to How hillshade works.

Availability: 2.0.0

Enhanced: 2.1.0 Uses ST_MapAlgebra() and added optional `interpolate_nodata` function parameter

Changed: 2.1.0 In prior versions, azimuth and altitude were expressed in radians. Now, azimuth and altitude are expressed in degrees

Examples

Variant 1.

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 1, 1, 1],
            [1, 2, 2, 2, 1],
            [1, 2, 3, 2, 1],
            [1, 2, 2, 2, 1],
            [1, 1, 1, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_DumpValues(ST_HillShade(rast, 1, '32BF'))
FROM foo

                                                                                    ↩
                               st_dumpvalues
```

Output

```
--------------------------------------------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------
 (1,"{{NULL,NULL,NULL,NULL,NULL},{NULL,251.32763671875,220.749786376953,147.224319458008,NULL},{NULL ↩
    ,220.749786376953,180.312225341797,67.7497863769531,NULL},{NULL,147.224319458008,
67.7497863769531,43.1210060119629,NULL},{NULL,NULL,NULL,NULL,NULL}}")
(1 row)
```

Variant 2.

Complete example of tiles of a coverage.

Code

```
WITH foo AS (
    SELECT ST_Tile(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(6, 6, 0, 0, 1, -1, 0, 0, 0),
                1, '32BF', 0, -9999
            ),
            1, 1, 1, ARRAY[
                [1, 1, 1, 1, 1, 1],
                [1, 1, 1, 1, 2, 1],
                [1, 2, 2, 3, 3, 1],
                [1, 1, 3, 2, 1, 1],
                [1, 2, 2, 1, 2, 1],
                [1, 1, 1, 1, 1, 1]
            ]::double precision[]
        ),
        2, 2
    ) AS rast
)
SELECT
    t1.rast,
    ST_HillShade(ST_Union(t2.rast), 1, t1.rast)
FROM foo t1
CROSS JOIN foo t2
WHERE ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rast;
```

See Also

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Roughness, ST_Aspect, ST_Slope

11.14.3 ST_Roughness

ST_Roughness — Returns a raster with the calculated "roughness" of a DEM.

Synopsis

raster **ST_Roughness**(raster rast, integer nband, raster customextent, text pixeltype="32BF" , boolean interpolate_nodata=FALSE);

Description

Calculates the "roughness" of a DEM, by subtracting the maximum from the minimum for a given area.

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 2, 3, 3],
            [1, 2, 4, 15, 3],
            [2, 4, 10, 7, 5],
            [3, 5, 7, 6, 4],
            [3, 3, 5, 4, 4]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(ST_Roughness(rast, 1, '32BF'), 3, 3) AS roughness
FROM foo;
```

Output

```
13
```

See Also

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Slope, ST_HillShade, ST_Aspect

11.14.4 ST_Slope

ST_Slope — Returns the slope (in degrees by default) of an elevation raster band. Useful for analyzing terrain.

Synopsis

raster **ST_Slope**(raster rast, integer nband=1, text pixeltype=32BF, text units=DEGREES, double precision scale=1.0, boolean interpolate_nodata=FALSE);
raster **ST_Slope**(raster rast, integer nband, raster customextent, text pixeltype=32BF, text units=DEGREES, double precision scale=1.0, boolean interpolate_nodata=FALSE);

Description

Returns the slope (in degrees by default) of an elevation raster band. Utilizes map algebra and applies the slope equation to neighboring pixels.

`units` indicates the units of the slope. Possible values are: RADIANS, DEGREES (default), PERCENT.

`scale` is the ratio of vertical units to horizontal. For Feet:LatLon use scale=370400, for Meters:LatLon use scale=111120.

If `interpolate_nodata` is TRUE, values for NODATA pixels from the input raster will be interpolated using ST_InvDistWeight4ma before computing the surface slope.



Note
For more information about Slope, Aspect and Hillshade, please refer to ESRI - How hillshade works and ERDAS Field Guide - Slope Images.

Availability: 2.0.0

Enhanced: 2.1.0 Uses ST_MapAlgebra() and added optional `units`, `scale`, `interpolate_nodata` function parameters

Changed: 2.1.0 In prior versions, return values were in radians. Now, return values default to degrees

Examples

Variant 1.

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 1, 1, 1],
            [1, 2, 2, 2, 1],
            [1, 2, 3, 2, 1],
            [1, 2, 2, 2, 1],
            [1, 1, 1, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_DumpValues(ST_Slope(rast, 1, '32BF'))
FROM foo

                            st_dumpvalues
```

Output

```
------------------------------------------------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------
 ↩
    (1,"{{10.0249881744385,21.5681285858154,26.5650520324707,21.5681285858154,10.0249881744385},{21.5681285858154,35.2643890380859,36.8698959350586,35.2643890380859,21.5681285858154},{26.5650520324707,36.8698959350586,0,36.8698959350586,26.5650520324707},{21.5681285858154,35.2643890380859,36.8698959350586,35.2643890380859,21.5681285858154},{10.0249881744385,21.5681285858154,26.5650520324707,21.5681285858154,10.0249881744385}}")
(1 row)
```

Variant 2.

Complete example of tiles of a coverage.

Code

```
WITH foo AS (
    SELECT ST_Tile(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(6, 6, 0, 0, 1, -1, 0, 0, 0),
                1, '32BF', 0, -9999
            ),
            1, 1, 1, ARRAY[
                [1, 1, 1, 1, 1, 1],
                [1, 1, 1, 1, 2, 1],
                [1, 2, 2, 3, 3, 1],
                [1, 1, 3, 2, 1, 1],
                [1, 2, 2, 1, 2, 1],
                [1, 1, 1, 1, 1, 1]
            ]::double precision[]
        ),
        2, 2
    ) AS rast
)
SELECT
    t1.rast,
    ST_Slope(ST_Union(t2.rast), 1, t1.rast)
FROM foo t1
CROSS JOIN foo t2
WHERE ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rast;
```

See Also

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Roughness, ST_HillShade, ST_Aspect

11.14.5 ST_TPI

ST_TPI — Returns a raster with the calculated Topographic Position Index.

Synopsis

raster **ST_TPI**(raster rast, integer nband, raster customextent, text pixeltype="32BF" , boolean interpolate_nodata=FALSE);

Description

Calculates the Topographic Position Index, which is defined as the focal mean with radius of one minus the center cell.



Note
This function only supports a focalmean radius of one.

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 2, 3, 3],
            [1, 2, 4, 15, 3],
            [2, 4, 10, 7, 5],
            [3, 5, 7, 6, 4],
            [3, 3, 5, 4, 4]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(ST_TPI(rast, 1, '32BF'), 3, 3) AS tpi
FROM foo;
```

Output

```
3.75
```

See Also

ST_MapAlgebra (callback function version), ST_TRI, ST_Roughness, ST_Slope, ST_HillShade, ST_Aspect

11.14.6 ST_TRI

ST_TRI — Returns a raster with the calculated Terrain Ruggedness Index.

Synopsis

raster **ST_TRI**(raster rast, integer nband, raster customextent, text pixeltype="32BF" , boolean interpolate_nodata=FALSE);

Description

Terrain Ruggedness Index is calculated by comparing a central pixel with its neighbors, taking the absolute values of the differences, and averaging the result.



Note
This function only supports a focalmean radius of one.

Availability: 2.1.0

Examples

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF' ↩
    , 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 2, 3, 3],
```

```
            [1, 2, 4, 15, 3],
            [2, 4, 10, 7, 5],
            [3, 5, 7, 6, 4],
            [3, 3, 5, 4, 4]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(ST_TRI(rast, 1, '32BF'), 3, 3) AS tri
FROM foo;
```

Output

```
5
```

See Also

ST_MapAlgebra (callback function version), ST_Roughness, ST_TPI, ST_Slope, ST_HillShade, ST_Aspect

11.14.7 ST_InterpolateRaster

ST_InterpolateRaster — Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation.

Synopsis

raster **ST_InterpolateRaster**(geometry input_points, text algorithm_options, raster template, integer template_band_num=1);

Description

Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation. There are five interpolation algorithms available: inverse distance, inverse distance nearest-neighbor, moving average, nearest neighbor, and linear interpolation. See the gdal_grid documentation for more details on the algorithms and their parameters. For more information on how interpolations are calculated, see the GDAL grid tutorial.

Input parameters are:

`input_points` The points to drive the interpolation. Any geometry with Z-values is acceptable, all points in the input will be used.

`algorithm_options` A string defining the algorithm and algorithm options, in the format used by gdal_grid. For example, for an inverse-distance interpolation with a smoothing of 2, you would use "invdist:smoothing=2.0"

`template` A raster template to drive the geometry of the output raster. The width, height, pixel size, spatial extent and pixel type will be read from this template.

`template_band_num` By default the first band in the template raster is used to drive the output raster, but that can be adjusted with this parameter.

Availability: 3.2.0

Example

Code

```
SELECT ST_InterpolateRaster('MULTIPOINT(10.5 9.5 1000,11.5 8.5 1000,10.5 8.5 500,11.5 9.5 ↩
    500)'::geometry,
    'invdist:smoothing:2.0',
    ST_AddBand(ST_MakeEmptyRaster(200, 400, 10, 10, 0.01, -0.005, 0, 0), '16BSI')
)
```

See Also

ST_Contour

11.14.8 ST_Contour

ST_Contour — Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.

Synopsis

setof record **ST_Contour**(raster rast, integer bandnumber=1, double precision level_interval=100.0, double precision level_base=0.0, double precision[] fixed_levels=ARRAY[], boolean polygonize=false);

Description

Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.

When the `fixed_levels` parameter is a non-empty array, the `level_interval` and `level_base` parameters are ignored.

Input parameters are:

`rast` The raster to generate the contour of

`bandnumber` The band to generate the contour of

`level_interval` The elevation interval between contours generated

`level_base` The "base" relative to which contour intervals are applied, this is normally zero, but could be different. To generate 10m contours at 5, 15, 25, ... the LEVEL_BASE would be 5.

`fixed_levels` The elevation interval between contours generated

`polygonize` If `true`, contour polygons will be created, rather than polygon lines.

Return values are a set of records with the following attributes:

`geom` The geometry of the contour line.

`id` A unique identifier given to the contour line by GDAL.

`value` The raster value the line represents. For an elevation DEM input, this would be the elevation of the output contour.

Availability: 3.2.0

Example

Code

```
WITH c AS (
SELECT (ST_Contour(rast, 1, fixed_levels => ARRAY[100.0, 200.0, 300.0])).*
FROM dem_grid WHERE rid = 1
)
SELECT geom, id, value
FROM c;
```

See Also

ST_InterpolateRaster

11.15 Raster Processing: Raster to Geometry

11.15.1 Box3D

Box3D — Returns the box 3d representation of the enclosing box of the raster.

Synopsis

box3d **Box3D**(raster rast);

Description

Returns the box representing the extent of the raster.

The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MAXX, MAXY))

Changed: 2.0.0 In pre-2.0 versions, there used to be a box2d instead of box3d. Since box2d is a deprecated type, this was changed to box3d.

Examples

Code

```
SELECT
    rid,
    Box3D(rast) AS rastbox
FROM dummy_rast;
```

Output

```
rid |        rastbox
----+-------------------------------------------------
1   | BOX3D(0.5 0.5 0,20.5 60.5 0)
2   | BOX3D(3427927.75 5793243.5 0,3427928 5793244 0)
```

See Also

ST_Envelope

11.15.2 ST_ConvexHull

ST_ConvexHull — Return the convex hull geometry of the raster including pixel values equal to BandNoDataValue. For regular shaped and non-skewed rasters, this gives the same result as ST_Envelope so only useful for irregularly shaped or skewed rasters.

Synopsis

geometry **ST_ConvexHull**(raster rast);

Description

Return the convex hull geometry of the raster including the NoDataBandValue band pixels. For regular shaped and non-skewed rasters, this gives more or less the same result as ST_Envelope so only useful for irregularly shaped or skewed rasters.



Note
ST_Envelope floors the coordinates and hence add a little buffer around the raster so the answer is subtly different from ST_ConvexHull which does not floor.

Examples

See `doc/development/internals/raster-affine.md` for the scale, skew, and rotation terms that make the convex hull and envelope differ for skewed rasters.

For regular non-skewed rasters, the envelope and convex hull are more or less the same.

Code

```
SELECT ST_ConvexHull(rast) As convhull, ST_Envelope(rast) As env
FROM dummy_rast WHERE rid=1;
```

Output

```
                      convhull                       |               env
-----------------------------------------------------+----------------------------------
 POLYGON((0.5 0.5,20.5 0.5,20.5 60.5,0.5 60.5,0.5 0.5)) | POLYGON((0 0,20 0,20 60,0 60,0 0))
```

Now we skew the raster. Note how the convex hull and envelope are now different.

Code

```
SELECT ST_ConvexHull(rast) As convhull, ST_Envelope(rast) As env
FROM (SELECT ST_SetRotation(rast, 0.1, 0.1) As rast
    FROM dummy_rast WHERE rid=1) As foo;
```

Output

```
                      convhull                       |               env
-----------------------------------------------------+----------------------------------
 POLYGON((0.5 0.5,20.5 1.5,22.5 61.5,2.5 60.5,0.5 0.5)) | POLYGON((0 0,22 0,22 61,0 61,0 0))
```

See Also

ST_Envelope, ST_MinConvexHull, ST_ConvexHull, ST_AsText

11.15.3 ST_DumpAsPolygons

ST_DumpAsPolygons — Returns a set of geomval (geom,val) rows, from a given raster band. If no band number is specified, band num defaults to 1.

Synopsis

setof geomval **ST_DumpAsPolygons**(raster rast, integer band_num=1, boolean exclude_nodata_value=TRUE);

Description

This is a set-returning function (SRF). It returns a set of geomval rows, formed by a geometry (geom) and a pixel band value (val). Each polygon is the union of all pixels for that band that have the same pixel value denoted by val.

ST_DumpAsPolygon is useful for polygonizing rasters. It is the reverse of a GROUP BY in that it creates new rows. For example it can be used to expand a single raster into multiple POLYGONS/MULTIPOLYGONS.

Changed 3.3.0, validation and fixing is disabled to improve performance. May result invalid geometries.

Changed 3.7.0, the polygonization honours PostgreSQL interrupts so cancellations and statement timeouts halt processing promptly.

Availability: Requires GDAL 1.7 or higher.



Note
If there is a no data value set for a band, pixels with that value will not be returned except in the case of exclude_nodata_value=false.



Note
If you only care about count of pixels with a given value in a raster, it is faster to use ST_ValueCount.



Note
This is different than ST_PixelAsPolygons where one geometry is returned for each pixel regardless of pixel value.

Examples

This syntax uses a LATERAL join.

Code

```
SELECT val,geom As geomwkt
FROM (
SELECT dp.*
FROM dummy_rast, LATERAL ST_DumpAsPolygons(rast) AS dp
WHERE rid = 2
) As foo
WHERE val BETWEEN 249 and 251
ORDER BY val;
```

Output

```
 val |                                                              geomwkt
-----+-----------------------------------------------------------------------------
 249 | POLYGON((3427927.95 5793243.95,3427927.95 5793243.85,3427928 5793243.85,
       3427928 5793243.95,3427927.95 5793243.95))
 250 | POLYGON((3427927.75 5793243.9,3427927.75 5793243.85,3427927.8 5793243.85,
       3427927.8 5793243.9,3427927.75 5793243.9))
 250 | POLYGON((3427927.8 5793243.8,3427927.8 5793243.75,3427927.85 5793243.75,
       3427927.85 5793243.8, 3427927.8 5793243.8))
 251 | POLYGON((3427927.75 5793243.85,3427927.75 5793243.8,3427927.8 5793243.8,
       3427927.8 5793243.85,3427927.75 5793243.85))
```

See Also

geomval, ST_AsRasterAgg, ST_Value, ST_Polygon, ST_ValueCount

11.15.4 ST_Envelope

ST_Envelope — Returns the polygon representation of the extent of the raster.

Synopsis

geometry **ST_Envelope**(raster rast);

Description

Returns the polygon representation of the extent of the raster in spatial coordinate units defined by srid. It is a float8 minimum bounding box represented as a polygon.

The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MINX, MAXY), (MAXX, MAXY), (MAXX, MINY), (MINX, MINY))

Examples

Code

```
SELECT rid, ST_Envelope(rast) As envgeomwkt
FROM dummy_rast;
```

Output

```
 rid |                                    envgeomwkt
-----+---------------------------------------------------------------------
   1 | POLYGON((0 0,20 0,20 60,0 60,0 0))
   2 | POLYGON((3427927 5793243,3427928 5793243,
       3427928 5793244,3427927 5793244, 3427927 5793243))
```

See Also

ST_Envelope, ST_AsText, ST_SRID

11.15.5 ST_MinConvexHull

ST_MinConvexHull — Return the convex hull geometry of the raster excluding NODATA pixels.

Synopsis

geometry **ST_MinConvexHull**(raster rast, integer nband=NULL);

Description

Return the convex hull geometry of the raster excluding NODATA pixels. If `nband` is NULL, all bands of the raster are considered.

Availability: 2.1.0

Examples

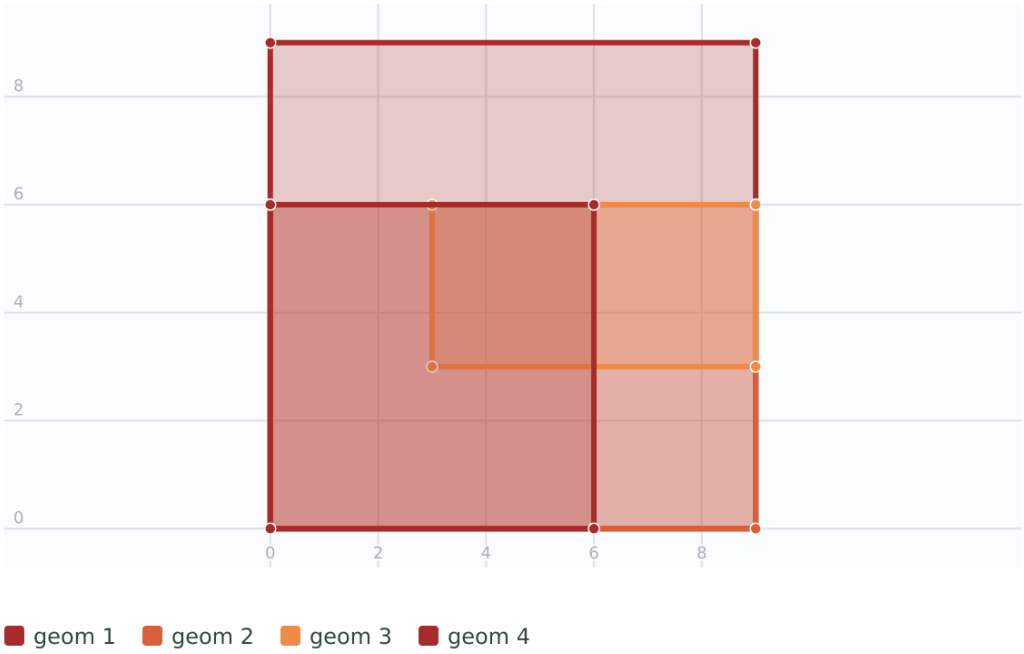
Code

```
WITH foo AS (
    SELECT
        ST_SetValues(ST_SetValues(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(9, 9, 0, 0, 1,  ↩
    -1, 0, 0, 0), 1, '8BUI', 0, 0), 2, '8BUI', 1, 0),
                1, 1, 1,
                ARRAY[
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0, 1],
                    [0, 0, 0, 1, 1, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0]
                ]::double precision[][]
            ),
            2, 1, 1,
            ARRAY[
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [1, 0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 0, 0]
            ]::double precision[][]
        ) AS rast
)
SELECT title, geom AS geom
FROM foo
CROSS JOIN LATERAL (VALUES
    ('raster hull', ST_ConvexHull(rast)),
    ('all bands', ST_MinConvexHull(rast)),
    ('band 1', ST_MinConvexHull(rast, 1)),
    ('band 2', ST_MinConvexHull(rast, 2))
) AS variants(title, geom);
```

Output

```
    title    |                 geom
-------------+--------------------------------------
 raster hull | POLYGON((0 0,9 0,9 -9,0 -9,0 0))
 all bands   | POLYGON((0 -3,9 -3,9 -9,0 -9,0 -3))
 band 1      | POLYGON((3 -3,9 -3,9 -6,3 -6,3 -3))
 band 2      | POLYGON((0 -3,6 -3,6 -9,0 -9,0 -3))
```

Figure



See Also

ST_Envelope, ST_ConvexHull, ST_ConvexHull, ST_AsText

11.15.6 ST_Polygon

ST_Polygon — Returns a multipolygon geometry formed by the union of pixels that have a pixel value that is not no data value. If no band number is specified, band num defaults to 1.

Synopsis

geometry **ST_Polygon**(raster rast, integer band_num=1);

Description

Changed 3.3.0, validation and fixing is disabled to improve performance. May result invalid geometries.

Availability: 0.1.6 Requires GDAL 1.7 or higher.

Enhanced: 2.1.0 Improved Speed (fully C-Based) and the returning multipolygon is ensured to be valid.

Changed: 2.1.0 In prior versions would sometimes return a polygon, changed to always return multipolygon.

Examples

Compare polygonization of every pixel with polygonization after zero is assigned as the NODATA value. The latter preserves the gap formed by zero-valued cells.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(4, 4, 0, 4, 1, -1, 0, 0, 0),
            1, '8BUI', 1, NULL
        ),
        1, 1, 1,
        ARRAY[
            [1, 1, 1, 1],
            [1, 0, 0, 1],
            [1, 1, 0, 1],
            [1, 1, 1, 1]
        ]::double precision[][]
    ) AS rast
), variants AS (
    SELECT 'all pixels' AS title, ST_Polygon(rast) AS geom
    FROM source
    UNION ALL
    SELECT
        'zero excluded',
        ST_Polygon(ST_SetBandNoDataValue(rast, 1, 0))
    FROM source
)
SELECT title, geom AS geom
FROM variants
ORDER BY CASE title
    WHEN 'all pixels' THEN 1
    ELSE 2
END;
```

Figure



See Also

ST_Value, ST_DumpAsPolygons

11.15.7 ST_IntersectionFractions

ST_IntersectionFractions — Calculates the fraction of each raster cell that is covered by a given geometry.

Synopsis

raster **ST_IntersectionFractions**(raster rast, geometry geom);

Description

Calculates the fraction of each raster cell that is covered by a given geometry. The first argument is a raster, which defines the grid geometry to use for the calculation. The extent and cell size are read from the raster parameter. The second argument is a geometry, which is overlaid with the grid, and each grid populated based on overlaying the geometry on the grid. For polygons, the value returned for each cell is the proportion of its area that is covered by the geometry. For linestrings, the value returned for each cell is the length contained in the cell.

Availability: 3.6.0 Requires GEOS 3.14 or higher.

Examples

This rotated square polygon covers half of each cell in the raster.

Code

```
CREATE TABLE raster_proportions_rast (
    name text,
    rast raster
);

INSERT INTO raster_proportions_rast (name, rast) VALUES (
  '2x2 raster covering 0,0 to 10,10',
  ST_MakeEmptyRaster(2, 2, -- raster width/height in pixels
    0, 10, -- upper-left corner x/y coordinates
    5, -5, -- pixel width/height in ground units
    0, 0, -- skew x/y
    0        -- SRID
  ));

SELECT name, ST_DumpValues(
    ST_IntersectionFractions(
        rast,
        'POLYGON((5 0,0 5,5 10,10 5,5 0))'::geometry),
    1)
FROM raster_proportions_rast;
```

Output

```
 2x2 raster covering 0,0 to 10,10
---------------------------------
 {{0.5,0.5},{0.5,0.5}}
```

See Also

ST_MakeEmptyRaster

11.16 Raster Operators

11.16.1 &&

&& — Returns TRUE if A's bounding box intersects B's bounding box.

Synopsis

boolean **&&**(raster A , raster B);
boolean **&&**(raster A , geometry B);
boolean **&&**(geometry B , raster A);

Description

The && operator returns TRUE if the bounding box of raster/geometr A intersects the bounding box of raster/geometr B.



Note
This operand will make use of any indexes that may be available on the rasters.

Availability: 2.0.0

Examples

Code

```
SELECT A.rid As a_rid, B.rid As b_rid, A.rast && B.rast As intersect
FROM dummy_rast AS A CROSS JOIN dummy_rast AS B LIMIT 3;
```

Output

```
 a_rid | b_rid | intersect
-------+-------+---------
     2 |     2 | t
     2 |     3 | f
     2 |     1 | f
```

11.16.2 &<

&< — Returns TRUE if A's bounding box is to the left of B's.

Synopsis

boolean **&<**(raster A , raster B);

Description

The &< operator returns TRUE if the bounding box of raster A overlaps or is to the left of the bounding box of raster B, or more accurately, overlaps or is NOT to the right of the bounding box of raster B.



Note
This operand will make use of any indexes that may be available on the rasters.

Examples

Code

```
SELECT A.rid As a_rid, B.rid As b_rid, A.rast &< B.rast As overleft
 FROM dummy_rast AS A CROSS JOIN dummy_rast AS B;
```

Output

```
a_rid | b_rid | overleft
------+-------+----------
    2 |     2 | t
    2 |     3 | f
    2 |     1 | f
    3 |     2 | t
    3 |     3 | t
    3 |     1 | f
    1 |     2 | t
    1 |     3 | t
    1 |     1 | t
```

11.16.3 &>

&> — Returns TRUE if A's bounding box is to the right of B's.

Synopsis

boolean **&>**(raster A , raster B);

Description

The &> operator returns TRUE if the bounding box of raster A overlaps or is to the right of the bounding box of raster B, or more accurately, overlaps or is NOT to the left of the bounding box of raster B.



Note
This operand will make use of any indexes that may be available on the geometries.

Examples

Code

```
SELECT A.rid As a_rid, B.rid As b_rid, A.rast &> B.rast As overright
 FROM dummy_rast AS A CROSS JOIN dummy_rast AS B;
```

Output

```
 a_rid | b_rid | overright
-------+-------+-----------
     2 |     2 | t
     2 |     3 | t
     2 |     1 | t
     3 |     2 | f
     3 |     3 | t
     3 |     1 | f
     1 |     2 | f
     1 |     3 | t
     1 |     1 | t
```

11.16.4 =

= — Returns TRUE if A's bounding box is the same as B's. Uses double precision bounding box.

Synopsis

boolean =(raster A , raster B);

Description

The = operator returns TRUE if the bounding box of raster A is the same as the bounding box of raster B. PostgreSQL uses the =, <, and > operators defined for rasters to perform internal orderings and comparison of rasters (ie. in a GROUP BY or ORDER BY clause).



Caution
This operand will NOT make use of any indexes that may be available on the rasters. Use ~= instead. This operator exists mostly so one can group by the raster column.

Availability: 2.1.0

See Also

~=

11.16.5 @

@ — Returns TRUE if A's bounding box is contained by B's. Uses double precision bounding box.

Synopsis

boolean @(raster A , raster B);
boolean @(geometry A , raster B);
boolean @(raster B , geometry A);

Description

The @ operator returns TRUE if the bounding box of raster/geometry A is contained by bounding box of raster/geometr B.



Note
This operand will use spatial indexes on the rasters.

Availability: 2.0.0 raster @ raster, raster @ geometry introduced

Availability: 2.0.5 geometry @ raster introduced

See Also

~

11.16.6 ~=

~= — Returns TRUE if A's bounding box is the same as B's.

Synopsis

boolean ~=(raster A , raster B);

Description

The ~= operator returns TRUE if the bounding box of raster A is the same as the bounding box of raster B.



Note
This operand will make use of any indexes that may be available on the rasters.

Availability: 2.0.0

Examples

Very useful usecase is for taking two sets of single band rasters that are of the same chunk but represent different themes and creating a multi-band raster

Code

```
SELECT ST_AddBand(prec.rast, alt.rast) As new_rast
FROM prec INNER JOIN alt ON (prec.rast ~= alt.rast);
```

See Also

ST_AddBand, =

11.16.7 ~

~ — Returns TRUE if A's bounding box is contains B's. Uses double precision bounding box.

Synopsis

boolean ~(raster A , raster B);
boolean ~(geometry A , raster B);
boolean ~(raster B , geometry A);

Description

The ~ operator returns TRUE if the bounding box of raster/geometry A is contains bounding box of raster/geometr B.



Note
This operand will use spatial indexes on the rasters.

Availability: 2.0.0

See Also

@

11.17 Raster and Raster Band Spatial Relationships

11.17.1 ST_Contains

ST_Contains — Return true if no points of raster rastB lie in the exterior of raster rastA and at least one point of the interior of rastB lies in the interior of rastA.

Synopsis

boolean **ST_Contains**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Contains**(raster rastA , raster rastB);

Description

Raster rastA contains rastB if and only if no points of rastB lie in the exterior of rastA and at least one point of the interior of rastB lies in the interior of rastA. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_Contains(ST_Polygon(raster), geometry) or ST_Contains(geometry, ST_Polygon(raster)).



Note
ST_Contains() is the inverse of ST_Within(). So, ST_Contains(rastA, rastB) implies ST_Within(rastB, rastA).

Availability: 2.1.0

Examples

Specified band numbers.

Code

```
SELECT r1.rid, r2.rid, ST_Contains(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 1;
```

Output

```
NOTICE:  The first raster provided has no bands
 rid | rid | st_contains
-----+-----+-------------
   1 |   1 |
   1 |   2 | f
```

No band numbers specified.

Code

```
SELECT r1.rid, r2.rid, ST_Contains(r1.rast, r2.rast) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 1;
```

Output

```
 rid | rid | st_contains
-----+-----+-------------
   1 |   1 | t
   1 |   2 | f
```

See Also

ST_Intersects, ST_Within

11.17.2 ST_ContainsProperly

ST_ContainsProperly — Return true if rastB intersects the interior of rastA but not the boundary or exterior of rastA.

Synopsis

boolean **ST_ContainsProperly**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_ContainsProperly**(raster rastA , raster rastB);

Description

Raster rastA contains properly rastB if rastB intersects the interior of rastA but not the boundary or exterior of rastA. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.

Raster rastA does not contain properly itself but does contain itself.



Note
This function will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_ContainsProperly(ST_Polygon(raster), geometry) or ST_ContainsProperly(geometry, ST_Polygon(raster)).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_ContainsProperly(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS ↩
    JOIN dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_containsproperly
-----+-----+---------------------
   2 |   1 | f
   2 |   2 | f
```

See Also

ST_Intersects, ST_Contains

11.17.3 ST_Covers

ST_Covers — Return true if no points of raster rastB lie outside raster rastA.

Synopsis

boolean **ST_Covers**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Covers**(raster rastA , raster rastB);

Description

Raster rastA covers rastB if and only if no points of rastB lie in the exterior of rastA. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_Covers(ST_Polygon(raster), geometry) or ST_Covers(geometry, ST_Polygon(raster)).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_Covers(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_covers
-----+-----+-----------
   2 |   1 | f
   2 |   2 | t
```

See Also

ST_Intersects, ST_CoveredBy

11.17.4 ST_CoveredBy

ST_CoveredBy — Return true if no points of raster rastA lie outside raster rastB.

Synopsis

boolean **ST_CoveredBy**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_CoveredBy**(raster rastA , raster rastB);

Description

Raster rastA is covered by rastB if and only if no points of rastA lie in the exterior of rastB. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_CoveredBy(ST_Polygon(raster), geometry) or ST_CoveredBy(geometry, ST_Polygon(raster)).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_CoveredBy(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_coveredby
-----+-----+--------------
   2 |   1 | f
   2 |   2 | t
```

See Also

ST_Intersects, ST_Covers

11.17.5 ST_Disjoint

ST_Disjoint — Return true if raster rastA does not spatially intersect rastB.

Synopsis

boolean **ST_Disjoint**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Disjoint**(raster rastA , raster rastB);

Description

Raster rastA and rastB are disjointed if they do not share any space together. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function does NOT use any indexes.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_Disjoint(ST_Polygon(raster), geometry).

Availability: 2.1.0

Examples

Rid = 1 has no bands, hence the NOTICE and the NULL value for st_disjoint.

Code

```
SELECT r1.rid, r2.rid, ST_Disjoint(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
NOTICE:  The second raster provided has no bands
 rid | rid | st_disjoint
-----+-----+-------------
   2 |   1 |
   2 |   2 | f
```

This time, without specifying band numbers.

Code

```
SELECT r1.rid, r2.rid, ST_Disjoint(r1.rast, r2.rast) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_disjoint
-----+-----+-------------
   2 |   1 | t
   2 |   2 | f
```

See Also

ST_Intersects

11.17.6 ST_Intersects

ST_Intersects — Return true if raster rastA spatially intersects raster rastB.

Synopsis

boolean **ST_Intersects**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Intersects**(raster rastA , raster rastB);
boolean **ST_Intersects**(raster rast , integer nband , geometry geommin);
boolean **ST_Intersects**(raster rast , geometry geommin , integer nband=NULL);
boolean **ST_Intersects**(geometry geommin , raster rast , integer nband=NULL);

Description

Return true if raster rastA spatially intersects raster rastB. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function will make use of any indexes that may be available on the rasters.

Enhanced: 2.0.0 support raster/raster intersects was introduced.



Warning
Changed: 2.1.0 The behavior of the ST_Intersects(raster, geometry) variants changed to match that of ST_Intersects(geometry, raster).

Examples

Different bands of same raster.

Code

```
SELECT ST_Intersects(rast, 2, rast, 3) FROM dummy_rast WHERE rid = 2;
```

Output

```
t
```

See Also

ST_Intersection, ST_Disjoint

11.17.7 ST_Overlaps

ST_Overlaps — Return true if raster rastA and rastB intersect but one does not completely contain the other.

Synopsis

boolean **ST_Overlaps**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Overlaps**(raster rastA , raster rastB);

Description

Return true if raster rastA spatially overlaps raster rastB. This means that rastA and rastB intersect but one does not completely contain the other. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_Overlaps(ST_Polygon(raster), geometry).

Availability: 2.1.0

Examples

Comparing different bands of same raster.

Code

```
SELECT ST_Overlaps(rast, 1, rast, 2) FROM dummy_rast WHERE rid = 2;
```

Output

```
f
```

See Also

ST_Intersects

11.17.8 ST_Touches

ST_Touches — Return true if raster rastA and rastB have at least one point in common but their interiors do not intersect.

Synopsis

boolean **ST_Touches**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Touches**(raster rastA , raster rastB);

Description

Return true if raster rastA spatially touches raster rastB. This means that rastA and rastB have at least one point in common but their interiors do not intersect. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This function will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_Touches(ST_Polygon(raster), geometry).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_Touches(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_touches
-----+-----+------------
   2 |   1 | f
   2 |   2 | f
```

See Also

ST_Intersects

11.17.9 ST_SameAlignment

ST_SameAlignment — Returns true if rasters have same skew, scale, spatial ref, and offset (pixels can be put on same grid without cutting into pixels) and false if they don't with notice detailing issue.

Synopsis

boolean **ST_SameAlignment**(raster rastA , raster rastB);
boolean **ST_SameAlignment**(double precision ulx1 , double precision uly1 , double precision scalex1 , double precision scaley1 , double precision skewx1 , double precision skewy1 , double precision ulx2 , double precision uly2 , double precision scalex2 , double precision scaley2 , double precision skewx2 , double precision skewy2);
boolean **ST_SameAlignment**(raster set rastfield);

Description

Non-Aggregate version (Variants 1 and 2): Returns true if the two rasters (either provided directly or made using the values for upperleft, scale, skew and srid) have the same scale, skew, srid and at least one of any of the four corners of any pixel of one raster falls on any corner of the grid of the other raster. Returns false if they don't and a NOTICE detailing the alignment issue.

Aggregate version (Variant 3): From a set of rasters, returns true if all rasters in the set are aligned. The ST_SameAlignment() function is an "aggregate" function in the terminology of PostgreSQL. That means that it operates on rows of data, in the same way the SUM() and AVG() functions do.

Availability: 2.0.0

Enhanced: 2.1.0 addition of Aggregate variant

Examples: Rasters

Code

```
SELECT ST_SameAlignment(ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0),
    ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0)
) as sm;
```

Output

```
t
```

Code

```
SELECT ST_SameAlignment(A.rast, b.rast)
FROM dummy_rast AS A CROSS JOIN dummy_rast AS B;
```

Output

```
NOTICE:  The two rasters provided have different SRIDs
NOTICE:  The two rasters provided have different SRIDs
 t
 f
 f
 f
```

See Also

Section 10.1, ST_NotSameAlignmentReason, ST_MakeEmptyRaster

11.17.10 ST_NotSameAlignmentReason

ST_NotSameAlignmentReason — Returns text stating if rasters are aligned and if not aligned, a reason why.

Synopsis

text **ST_NotSameAlignmentReason**(raster rastA, raster rastB);

Description

Returns text stating if rasters are aligned and if not aligned, a reason why.



Note
If there are several reasons why the rasters are not aligned, only one reason (the first test to fail) will be returned.

Availability: 2.1.0

Examples

Code

```
SELECT
    ST_SameAlignment(ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0),
        ST_MakeEmptyRaster(1, 1, 0, 0, 1.1, 1.1, 0, 0)
    ),
    ST_NotSameAlignmentReason(ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0),
        ST_MakeEmptyRaster(1, 1, 0, 0, 1.1, 1.1, 0, 0)
    )
;
```

Output

```
 st_samealignment |            st_notsamealignmentreason
------------------+-------------------------------------------------
 f                | The rasters have different scales on the X axis
(1 row)
```

See Also

Section 10.1, ST_SameAlignment

11.17.11 ST_Within

ST_Within — Return true if no points of raster rastA lie in the exterior of raster rastB and at least one point of the interior of rastA lies in the interior of rastB.

Synopsis

boolean **ST_Within**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Within**(raster rastA , raster rastB);

Description

Raster rastA is within rastB if and only if no points of rastA lie in the exterior of rastB and at least one point of the interior of rastA lies in the interior of rastB. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.



Note
This operand will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_Within(ST_Polygon(raster), geometry) or ST_Within(geometry, ST_Polygon(raster)).



Note
ST_Within() is the inverse of ST_Contains(). So, ST_Within(rastA, rastB) implies ST_Contains(rastB, rastA).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_Within(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_within
-----+-----+-----------
   2 |   1 | f
   2 |   2 | t
```

See Also

ST_Intersects, ST_Contains, ST_DWithin, ST_DFullyWithin

11.17.12 ST_DWithin

ST_DWithin — Return true if rasters rastA and rastB are within the specified distance of each other.

Synopsis

boolean **ST_DWithin**(raster rastA , integer nbandA , raster rastB , integer nbandB , double precision distance_of_srid);
boolean **ST_DWithin**(raster rastA , raster rastB , double precision distance_of_srid);

Description

Return true if rasters rastA and rastB are within the specified distance of each other. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.

The distance is specified in units defined by the spatial reference system of the rasters. For this function to make sense, the source rasters must both be of the same coordinate projection, having the same SRID.



Note
This operand will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_DWithin(ST_Polygon(raster), geometry).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_DWithin(r1.rast, 1, r2.rast, 1, 3.14) FROM dummy_rast r1 CROSS ↩
    JOIN dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_dwithin
-----+-----+------------
   2 |   1 | f
   2 |   2 | t
```

See Also

ST_Within, ST_DFullyWithin

11.17.13 ST_DFullyWithin

ST_DFullyWithin — Return true if rasters rastA and rastB are fully within the specified distance of each other.

Synopsis

boolean **ST_DFullyWithin**(raster rastA , integer nbandA , raster rastB , integer nbandB , double precision distance_of_srid);
boolean **ST_DFullyWithin**(raster rastA , raster rastB , double precision distance_of_srid);

Description

Return true if rasters rastA and rastB are fully within the specified distance of each other. If the band number is not provided (or set to NULL), only the convex hull of the raster is considered in the test. If the band number is provided, only those pixels with value (not NODATA) are considered in the test.

The distance is specified in units defined by the spatial reference system of the rasters. For this function to make sense, the source rasters must both be of the same coordinate projection, having the same SRID.



Note
This operand will make use of any indexes that may be available on the rasters.



Note
To test the spatial relationship of a raster and a geometry, use ST_Polygon on the raster, e.g. ST_DFullyWithin(ST_Polygon(raster), geometry).

Availability: 2.1.0

Examples

Code

```
SELECT r1.rid, r2.rid, ST_DFullyWithin(r1.rast, 1, r2.rast, 1, 3.14) FROM dummy_rast r1 ↩
    CROSS JOIN dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
 rid | rid | st_dfullywithin
-----+-----+-----------------
   2 |   1 | f
   2 |   2 | t
```

See Also

ST_Within, ST_DWithin

11.18 Raster Tips

11.18.1 Out-DB Rasters

11.18.1.1 Directory containing many files

When GDAL opens a file, GDAL eagerly scans the directory of that file to build a catalog of other files. If this directory contains many files (e.g. thousands, millions), opening that file becomes extremely slow (especially if that file happens to be on a network drive such as NFS).

To control this behavior, GDAL provides the following environment variable: GDAL_DISABLE_READDIR_ON_OPEN. Set `GDAL_DISABLE_READDIR_ON_OPEN` to `TRUE` to disable directory scanning.

In Ubuntu (and assuming you are using PostgreSQL's packages for Ubuntu), `GDAL_DISABLE_READDIR_ON_OPEN` can be set in */etc/postgresql/POSTGRESQL_VERSION/CLUSTER_NAME/environment* (where POSTGRESQL_VERSION is the version of PostgreSQL, e.g. 9.6 and CLUSTER_NAME is the name of the cluster, e.g. maindb). You can also set PostGIS environment variables here as well.

Code

```
# environment variables for postmaster process
# This file has the same syntax as postgresql.conf:
#  VARIABLE = simple_value
#  VARIABLE2 = 'any value!'
# I. e. you need to enclose any value which does not only consist of letters,
# numbers, and '-', '_', '.' in single quotes. Shell commands are not
# evaluated.
POSTGIS_GDAL_ENABLED_DRIVERS = 'ENABLE_ALL'

POSTGIS_ENABLE_OUTDB_RASTERS = 1

GDAL_DISABLE_READDIR_ON_OPEN = 'TRUE'
```

11.18.1.2 Maximum Number of Open Files

The maximum number of open files permitted by Linux and PostgreSQL are typically conservative (typically 1024 open files per process) given the assumption that the system is consumed by human users. For Out-DB Rasters, a single valid query can easily exceed this limit (e.g. a dataset of 10 year's worth of rasters with one raster for each day containing minimum and maximum temperatures and we want to know the absolute min and max value for a pixel in that dataset).

The easiest change to make is the following PostgreSQL setting: max_files_per_process. The default is set to 1000, which is far too low for Out-DB Rasters. A safe starting value could be 65536 but this really depends on your datasets and the queries run against those datasets. This setting can only be made on server start and probably only in the PostgreSQL configuration file (e.g. */etc/postgresql/POSTGRESQL_VERSION/CLUSTER_NAME/postgresql.conf* in Ubuntu environments).

Code

```
...
# - Kernel Resource Usage -

max_files_per_process = 65536           # min 25
                                        # (change requires restart)
...
```

The major change to make is the Linux kernel's open files limits. There are two parts to this:

- Maximum number of open files for the entire system
- Maximum number of open files per process

11.18.1.2.1 Maximum number of open files for the entire system

You can inspect the current maximum number of open files for the entire system with the following example:

Code

```
$ sysctl -a | grep fs.file-max
fs.file-max = 131072
```

If the value returned is not large enough, add a file to */etc/sysctl.d/* as per the following example:

Code

```
$ echo "fs.file-max = 6145324" >> /etc/sysctl.d/fs.conf

$ cat /etc/sysctl.d/fs.conf
fs.file-max = 6145324

$ sysctl -p --system
* Applying /etc/sysctl.d/fs.conf ...
fs.file-max = 2097152
* Applying /etc/sysctl.conf ...

$ sysctl -a | grep fs.file-max
fs.file-max = 6145324
```

11.18.1.2.2 Maximum number of open files per process

We need to increase the maximum number of open files per process for the PostgreSQL server processes.

To see what the current PostgreSQL service processes are using for maximum number of open files, do as per the following example (make sure to have PostgreSQL running):

Code

```
$ ps aux | grep postgres
```

Output

```
postgres 31713  0.0  0.4 179012 17564 pts/0    S    Dec26   0:03 /home/dustymugs/devel/postgresql/sandbox/10/usr/local/ ↩
     bin/postgres -D /home/dustymugs/devel/postgresql/sandbox/10/pgdata
postgres 31716  0.0  0.8 179776 33632 ?        Ss   Dec26   0:01 postgres: checkpointer process
postgres 31717  0.0  0.2 179144  9416 ?        Ss   Dec26   0:05 postgres: writer process
postgres 31718  0.0  0.2 179012  8708 ?        Ss   Dec26   0:06 postgres: wal writer process
postgres 31719  0.0  0.1 179568  7252 ?        Ss   Dec26   0:03 postgres: autovacuum launcher process
postgres 31720  0.0  0.1  34228  4124 ?        Ss   Dec26   0:09 postgres: stats collector process
postgres 31721  0.0  0.1 179308  6052 ?        Ss   Dec26   0:00 postgres: bgworker: logical replication launcher
```

Code

```
$ cat /proc/31718/limits
```

Output

```
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             15738                15738                processes
Max open files            1024                 4096                 files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       15738                15738                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

In the example above, we inspected the open files limit for Process 31718. It doesn't matter which PostgreSQL process, any of them will do. The response we are interested in is *Max open files*.

We want to increase *Soft Limit* and *Hard Limit* of *Max open files* to be greater than the value we specified for the PostgreSQL setting `max_files_per_process`. In our example, we set `max_files_per_process` to 65536.

In Ubuntu (and assuming you are using PostgreSQL's packages for Ubuntu), the easiest way to change the *Soft Limit* and *Hard Limit* is to edit **/etc/init.d/postgresql** (SysV) or **/lib/systemd/system/postgresql*.service** (systemd).

Let's first address the SysV Ubuntu case where we add **ulimit -H -n 262144** and **ulimit -n 131072** to **/etc/init.d/postgresql**.

Code

```
...
case "$1" in
    start|stop|restart|reload)
        if [ "$1" = "start" ]; then
            create_socket_directory
        fi
    if [ -z "`pg_lsclusters -h`" ]; then
        log_warning_msg 'No PostgreSQL clusters exist; see "man pg_createcluster"'
        exit 0
    fi

    ulimit -H -n 262144
    ulimit -n 131072

    for v in $versions; do
        $1 $v || EXIT=$?
    done
    exit ${EXIT:-0}
        ;;
    status)
...
```

Now to address the systemd Ubuntu case. We will add **LimitNOFILE=131072** to every **/lib/systemd/system/postgresql*.service** file in the **[Service]** section.

Code

```
...
[Service]

LimitNOFILE=131072

...

[Install]
WantedBy=multi-user.target
...
```

After making the necessary systemd changes, make sure to reload the daemon

Code

```
systemctl daemon-reload
```

Chapter 12

PostGIS Extras

This chapter documents features found in the extras folder of the PostGIS source tarballs and source repository. These are not always packaged with PostGIS binary releases, but are usually PL/pgSQL based or standard shell scripts that can be run as is.

Chapter 13

PostGIS Special Functions Index

13.1 PostGIS Aggregate Functions

The functions below are spatial aggregate functions that are used in the same way as SQL aggregate function such as `sum` and `average`.

- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_AsFlatGeobuf - Return a FlatGeobuf representation of a set of rows.
- ST_AsGeobuf - Return a Geobuf representation of a set of rows.
- ST_AsMVT - Aggregate function returning a MVT representation of a set of rows.
- ST_ClusterDBSCAN - Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterIntersecting - Aggregate function that clusters input geometries into connected sets.
- ST_ClusterIntersectingWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.
- ST_ClusterKMeans - Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_ClusterRelateWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.
- ST_ClusterWithin - Aggregate function that clusters geometries by separation distance.
- ST_ClusterWithinWin - Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_Collect - Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_CoverageClean - Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.
- ST_CoverageInvalidEdges - Window function that finds locations where polygons fail to form a valid coverage.
- ST_CoverageSimplify - Window function that simplifies the edges of a polygonal coverage.
- ST_CoverageUnion - Computes the union of a set of polygons forming a coverage by removing shared edges.
- ST_Extent - Aggregate function that returns the bounding box of geometries.
- ST_MakeLine - Creates a LineString from Point, MultiPoint, or LineString geometries.

- ST_MemUnion - Aggregate function which unions geometries in a memory-efficent but slower way
- ST_MinimumSpanningTree - Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.
- ST_Polygonize - Computes a collection of polygons formed from the linework of a set of geometries.
- ST_SameAlignment - Returns true if rasters have same skew, scale, spatial ref, and offset (pixels can be put on same grid without cutting into pixels) and false if they don't with notice detailing issue.
- ST_Union - Computes a geometry representing the point-set union of the input geometries.
- ST_Union - Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- TopoElementArray_Agg - Returns a topoelementarray for a set of element_id, type arrays (topoelements).

13.2 PostGIS Window Functions

The functions below are spatial window functions that are used in the same way as SQL window functions such as `row_number()`, `lead()`, and `lag()`. They must be followed by an `OVER()` clause.

- ST_ClusterDBSCAN - Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterIntersectingWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.
- ST_ClusterKMeans - Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_ClusterRelateWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.
- ST_ClusterWithinWin - Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_CoverageClean - Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.
- ST_CoverageInvalidEdges - Window function that finds locations where polygons fail to form a valid coverage.
- ST_CoverageSimplify - Window function that simplifies the edges of a polygonal coverage.
- ST_MinimumSpanningTree - Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.

13.3 PostGIS SQL-MM Compliant Functions

The functions given below are PostGIS functions that conform to the SQL/MM 3 standard

- CG_3DArea - Computes area of 3D surface geometries. Will return 0 for solids.
- CG_3DDifference - Perform 3D difference
- CG_3DIntersection - Perform 3D intersection
- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- CG_Volume - Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- ST_3DDWithin - Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.

- ST_3DIntersects - Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_3DLength - Returns the 3D length of a linear geometry.
- ST_3DPerimeter - Returns the 3D perimeter of a polygonal geometry.
- ST_AddEdgeModFace - Add a new edge and, if in doing so it splits a face, modify the original face and add a new face.
- ST_AddEdgeNewFaces - Add a new edge and, if in doing so it splits a face, delete the original face and replace it with two new faces.
- ST_AddIsoEdge - Adds an isolated edge defined by geometry alinestring to a topology connecting two existing isolated nodes anode and anothernode and returns the edge id of the new edge.
- ST_AddIsoNode - Adds an isolated node to a face in a topology and returns the nodeid of the new node. If face is null, the node is still created.
- ST_Area - Returns the area of a polygonal geometry.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsGML - Return the geometry as a GML version 2 or 3 element.
- ST_AsText - Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.
- ST_Boundary - Returns the boundary of a geometry.
- ST_Buffer - Computes a geometry covering all points within a given distance from a geometry.
- ST_Centroid - Returns the geometric center of a geometry.
- ST_ChangeEdgeGeom - Changes the shape of an edge without affecting the topology structure.
- ST_Contains - Tests if every point of B lies in A, and their interiors have a point in common
- ST_ConvexHull - Computes the convex hull of a geometry.
- ST_CoordDim - Return the coordinate dimension of a geometry.
- ST_CreateTopoGeo - Adds a collection of geometries to a given empty topology and returns a message detailing success.
- ST_Crosses - Tests if two geometries have some, but not all, interior points in common
- ST_CurveN - Returns the Nth component curve geometry of a CompoundCurve.
- ST_CurveToLine - Converts a geometry containing curves to a linear geometry.
- ST_Difference - Computes a geometry representing the part of geometry A that does not intersect geometry B.
- ST_Dimension - Returns the topological dimension of a geometry.
- ST_Disjoint - Tests if two geometries have no points in common
- ST_Distance - Returns the distance between two geometry or geography values.
- ST_EndPoint - Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_Envelope - Returns a geometry representing the bounding box of a geometry.
- ST_Equals - Tests if two geometries include the same set of points
- ST_ExteriorRing - Returns a LineString representing the exterior ring of a Polygon.
- ST_GMLToSQL - Returns a geometry from GML. Alias for ST_GeomFromGML.

- ST_GeomCollFromText - Makes a collection Geometry from collection WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_GeomFromText - Return a specified ST_Geometry value from Well-Known Text representation (WKT).
- ST_GeomFromWKB - Creates a geometry instance from a Well-Known Binary geometry representation (WKB) and optional SRID.
- ST_GeometryFromText - Returns a geometry from Well-Known Text (WKT). Alias for ST_GeomFromText.
- ST_GeometryN - Return an element of a geometry collection.
- ST_GeometryType - Returns the SQL-MM type of a geometry as text.
- ST_GetFaceEdges - Returns a set of ordered edges that bound aface.
- ST_GetFaceGeometry - Returns the polygon in the given topology with the specified face id.
- ST_InitTopoGeo - Creates a new topology schema and registers it in the topology.topology table.
- ST_InteriorRingN - Returns the Nth interior ring (hole) of a Polygon.
- ST_Intersection - Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Tests if two geometries intersect (they have at least one point in common)
- ST_IsClosed - Tests if a LineStrings's start and end points are coincident. For a PolyhedralSurface tests if it is closed (volumetric).
- ST_IsEmpty - Tests if a geometry is empty.
- ST_IsRing - Tests if a LineString is closed and simple.
- ST_IsSimple - Tests if a geometry has no points of self-intersection or self-tangency.
- ST_IsValid - Tests if a geometry is well-formed in 2D.
- ST_Length - Returns the 2D length of a linear geometry.
- ST_LineFromText - Makes a Geometry from WKT representation with the given SRID. If SRID is not given, it defaults to 0.
- ST_LineFromWKB - Creates a LINESTRING from a WKB representation, optionally using a given SRID.
- ST_LinestringFromWKB - Creates a LINESTRING from a WKB representation, optionally using a given SRID.
- ST_LocateAlong - Returns the point(s) on a geometry that match a measure value.
- ST_LocateBetween - Returns the portions of a geometry that match a measure range.
- ST_M - Returns the M coordinate of a Point.
- ST_MLineFromText - Return a specified ST_MultiLineString value from WKT representation.
- ST_MPointFromText - Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_MPolyFromText - Makes a MultiPolygon Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_ModEdgeHeal - Heals two edges by deleting the node connecting them, modifying the first edgeand deleting the second edge. Returns the id of the deleted node.
- ST_ModEdgeSplit - Split an edge by creating a new node along an existing edge, modifying the original edge and adding a new edge.
- ST_MoveIsoNode - Moves an isolated node in a topology from one point to another. If new apoint geometry exists as a node an error is thrown. Returns description of move.
- ST_NewEdgeHeal - Heals two edges by deleting the node connecting them, deleting both edges,and replacing them with an edge whose direction is the same as the firstedge provided.

- ST_NewEdgesSplit - Split an edge by creating a new node along an existing edge, deleting the original edge and replacing it with two new edges. Returns the id of the new node created that joins the new edges.
- ST_NumCurves - Return the number of component curves in a CompoundCurve.
- ST_NumGeometries - Returns the number of elements in a geometry collection.
- ST_NumInteriorRings - Returns the number of interior rings (holes) of a Polygon.
- ST_NumPatches - Returns the number of faces in a PolyhedralSurface or TIN.
- ST_NumPoints - Returns the number of points in a LineString or CircularString.
- ST_OrderingEquals - Tests if two geometries represent the same geometry and have points in the same directional order
- ST_Overlaps - Tests if two geometries have the same dimension and intersect, but each has at least one point not in the other
- ST_PatchN - Returns the Nth face of a PolyhedralSurface or TIN.
- ST_Perimeter - Returns the length of the boundary of a polygonal geometry or geography.
- ST_Point - Creates a Point with X, Y and SRID values.
- ST_PointFromText - Makes a point Geometry from WKT with the given SRID. If SRID is not given, it defaults to unknown.
- ST_PointFromWKB - Makes a geometry from WKB with the given SRID
- ST_PointN - Returns the Nth point in the first LineString or circular LineString in a geometry.
- ST_PointOnSurface - Computes a point guaranteed to lie in a polygon, or on a geometry.
- ST_Polygon - Creates a Polygon from a LineString with a specified SRID.
- ST_PolygonFromText - Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_Relate - Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix
- ST_RemEdgeModFace - Removes an edge, and if the edge separates two facesdeletes one face and modifies the other face to cover the space of both.
- ST_RemEdgeNewFace - Removes an edge and, if the removed edge separated two faces,delete the original faces and replace them with a new face.
- ST_RemoveIsoEdge - Removes an isolated edge and returns description of action. If the edge is not isolated, then an exception is thrown.
- ST_RemoveIsoNode - Removes an isolated node and returns description of action. If the node is not isolated (is start or end of an edge), then an exception is thrown.
- ST_SRID - Returns the spatial reference identifier for a geometry.
- ST_StartPoint - Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_SymDifference - Computes a geometry representing the portions of geometries A and B that do not intersect.
- ST_Touches - Tests if two geometries have at least one point in common, but their interiors do not intersect
- ST_Transform - Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Union - Computes a geometry representing the point-set union of the input geometries.
- ST_WKBToSQL - Returns a geometry from Well-Known Binary (WKB). SQL/MM alias for ST_GeomFromWKB without an SRID argument.
- ST_WKTToSQL - Returns a geometry from Well-Known Text (WKT). SQL/MM alias for ST_GeomFromText.

- ST_Within - Tests if every point of A lies in B, and their interiors have a point in common
- ST_X - Returns the X coordinate of a Point.
- ST_Y - Returns the Y coordinate of a Point.
- ST_Z - Returns the Z coordinate of a Point.

13.4 PostGIS Geography Support Functions

The functions and operators given below are PostGIS functions/operators that take as input or return as output a geography data type object.



Note
Functions with a (T) are not native geodetic functions, and use a ST_Transform call to and from geometry to do the operation. As a result, they may not behave as expected when going over dateline, poles, and for large geometries or geometry pairs that cover more than one UTM zone. Basic transform - (favoring UTM, Lambert Azimuthal (North/South), and falling back on mercator in worst case scenario)

- ST_Area - Returns the area of a polygonal geometry.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKT - Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.
- ST_AsGML - Return the geometry as a GML version 2 or 3 element.
- ST_AsGeoJSON - Return a geometry or feature in GeoJSON format.
- ST_AsKML - Return the geometry as a KML element.
- ST_AsSVG - Returns SVG path data for a geometry.
- ST_AsText - Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.
- ST_Azimuth - Returns the north-based azimuth of a line between two points.
- ST_Buffer - Computes a geometry covering all points within a given distance from a geometry.
- ST_Centroid - Returns the geometric center of a geometry.
- ST_ClosestPoint - Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.
- ST_CoveredBy - Tests if every point of A lies in B
- ST_Covers - Tests if every point of B lies in A
- ST_DWithin - Tests if two geometries are within a given distance
- ST_Distance - Returns the distance between two geometry or geography values.
- ST_GeogFromText - Return a specified geography value from Well-Known Text representation or extended (WKT).
- ST_GeogFromWKB - Creates a geography instance from a Well-Known Binary geometry representation (WKB) or extended Well Known Binary (EWKB).
- ST_GeographyFromText - Return a specified geography value from Well-Known Text representation or extended (WKT).
- = - Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.

- ST_Intersection - Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Tests if two geometries intersect (they have at least one point in common)
- ST_Length - Returns the 2D length of a linear geometry.
- ST_LineInterpolatePoint - Returns a point interpolated along a line at a fractional location.
- ST_LineInterpolatePoints - Returns points interpolated along a line at a fractional interval.
- ST_LineLocatePoint - Returns the fractional location of the closest point on a line to a point.
- ST_LineSubstring - Returns the part of a line between two fractional locations.
- ST_Perimeter - Returns the length of the boundary of a polygonal geometry or geography.
- ST_Project - Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Segmentize - Returns a modified geometry/geography having no segment longer than a given distance.
- ST_ShortestLine - Returns the 2D shortest line between two geometries
- ST_Summary - Returns a text summary of the contents of a geometry.
- <-> - Returns the 2D distance between A and B.
- && - Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.

13.5 PostGIS Raster Support Functions

The functions and operators given below are PostGIS functions/operators that take as input or return as output a raster data type object. Listed in alphabetical order.

- Box3D - Returns the box 3d representation of the enclosing box of the raster.
- @ - Returns TRUE if A's bounding box is contained by B's. Uses double precision bounding box.
- ~ - Returns TRUE if A's bounding box is contains B's. Uses double precision bounding box.
- = - Returns TRUE if A's bounding box is the same as B's. Uses double precision bounding box.
- && - Returns TRUE if A's bounding box intersects B's bounding box.
- &< - Returns TRUE if A's bounding box is to the left of B's.
- &> - Returns TRUE if A's bounding box is to the right of B's.
- ~= - Returns TRUE if A's bounding box is the same as B's.
- ST_Retile - Return a set of configured tiles from an arbitrarily tiled raster coverage.
- ST_AddBand - Returns a raster with the new band(s) of given type added with given initial value in the given index location. If no index is specified, the band is added to the end.
- ST_AsBinary/ST_AsWKB - Return the Well-Known Binary (WKB) representation of the raster.
- ST_AsGDALRaster - Return the raster tile in the designated GDAL Raster format. Raster formats are one of those supported by your compiled library. Use ST_GDALDrivers() to get a list of formats supported by your library.
- ST_AsHexWKB - Return the Well-Known Binary (WKB) in Hex representation of the raster.
- ST_AsJPEG - Return the raster tile selected bands as a single Joint Photographic Exports Group (JPEG) image (byte array). If no band is specified and 1 or more than 3 bands, then only the first band is used. If only 3 bands then all 3 bands are used and mapped to RGB.

- ST_AsPNG - Return the raster tile selected bands as a single portable network graphics (PNG) image (byte array). If 1, 3, or 4 bands in raster and no bands are specified, then all bands are used. If more 2 or more than 4 bands and no bands specified, then only band 1 is used. Bands are mapped to RGB or RGBA space.
- ST_AsRaster - Converts a PostGIS geometry to a PostGIS raster.
- ST_AsRasterAgg - Aggregate. Renders PostGIS geometries into a new raster.
- ST_AsTIFF - Return the raster selected bands as a single TIFF image (byte array). If no band is specified or any of specified bands does not exist in the raster, then will try to use all bands.
- ST_Aspect - Returns the aspect (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_Band - Returns one or more bands of an existing raster as a new raster. Useful for building new rasters from existing rasters.
- ST_BandFileSize - Returns the file size of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandFileTimestamp - Returns the file timestamp of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandIsNoData - Returns true if the band is filled with only nodata values.
- ST_BandMetaData - Returns basic meta data for a specific raster band. band num 1 is assumed if none-specified.
- ST_BandNoDataValue - Returns the value in a given band that represents no data. If no band num 1 is assumed.
- ST_BandPath - Returns system file path to a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandPixelType - Returns the type of pixel for given band. If no bandnum specified, 1 is assumed.
- ST_Clip - Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.
- ST_ColorMap - Creates a new raster of up to four 8BUI bands (grayscale, RGB, RGBA) from the source raster and a specified band. Band 1 is assumed if not specified.
- ST_Contains - Return true if no points of raster rastB lie in the exterior of raster rastA and at least one point of the interior of rastB lies in the interior of rastA.
- ST_ContainsProperly - Return true if rastB intersects the interior of rastA but not the boundary or exterior of rastA.
- ST_Contour - Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.
- ST_ConvexHull - Return the convex hull geometry of the raster including pixel values equal to BandNoDataValue. For regular shaped and non-skewed rasters, this gives the same result as ST_Envelope so only useful for irregularly shaped or skewed rasters.
- ST_Count - Returns the number of pixels in a given band of a raster or raster coverage. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the nodata value.
- ST_CountAgg - Aggregate. Returns the number of pixels in a given band of a set of rasters. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the NODATA value.
- ST_CoveredBy - Return true if no points of raster rastA lie outside raster rastB.
- ST_Covers - Return true if no points of raster rastB lie outside raster rastA.
- ST_DFullyWithin - Return true if rasters rastA and rastB are fully within the specified distance of each other.
- ST_DWithin - Return true if rasters rastA and rastB are within the specified distance of each other.
- ST_Disjoint - Return true if raster rastA does not spatially intersect rastB.
- ST_DumpAsPolygons - Returns a set of geomval (geom,val) rows, from a given raster band. If no band number is specified, band num defaults to 1.

- ST_DumpValues - Get the values of the specified band as a 2-dimension array.
- ST_Envelope - Returns the polygon representation of the extent of the raster.
- ST_FromGDALRaster - Returns a raster from a supported GDAL raster file.
- ST_GeoReference - Returns the georeference meta data in GDAL or ESRI format as commonly seen in a world file. Default is GDAL.
- ST_Grayscale - Creates a new one-8BUI band raster from the source raster and specified bands representing Red, Green and Blue
- ST_HasNoBand - Returns true if there is no band with given band number. If no band number is specified, then band number 1 is assumed.
- ST_Height - Returns the height of the raster in pixels.
- ST_HillShade - Returns the hypothetical illumination of an elevation raster band using provided azimuth, altitude, brightness and scale inputs.
- ST_Histogram - Returns a set of record summarizing a raster or raster coverage data distribution separate bin ranges. Number of bins are autocomputed if not specified.
- ST_InterpolateRaster - Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation.
- ST_Intersection - Returns a raster or a set of geometry-pixelvalue pairs representing the shared portion of two rasters or the geometrical intersection of a vectorization of the raster and a geometry.
- ST_IntersectionFractions - Calculates the fraction of each raster cell that is covered by a given geometry.
- ST_Intersects - Return true if raster rastA spatially intersects raster rastB.
- ST_IsEmpty - Returns true if the raster is empty (width = 0 and height = 0). Otherwise, returns false.
- ST_MakeEmptyCoverage - Cover georeferenced area with a grid of empty raster tiles.
- ST_MakeEmptyRaster - Returns an empty raster (having no bands) of given dimensions (width & height), upperleft X and Y, pixel size and rotation (scalex, scaley, skewx & skewy) and reference system (srid). If a raster is passed in, returns a new raster with the same size, alignment and SRID. If srid is left out, the spatial ref is set to unknown (0).
- ST_MapAlgebra (callback function version) - Callback function version - Returns a one-band raster given one or more input rasters, band indexes and one user-specified callback function.
- ST_MapAlgebraExpr - 1 raster band version: Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation on the input raster band and of pixeltype provided. Band 1 is assumed if no band is specified.
- ST_MapAlgebraExpr - 2 raster band version: Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation on the two input raster bands and of pixeltype provided. band 1 of each raster is assumed if no band numbers are specified. The resulting raster will be aligned (scale, skew and pixel corners) on the grid defined by the first raster and have its extent defined by the "extenttype" parameter. Values for "extenttype" can be: INTERSECTION, UNION, FIRST, SECOND.
- ST_MapAlgebraFct - 1 band version - Creates a new one band raster formed by applying a valid PostgreSQL function on the input raster band and of pixeltype provided. Band 1 is assumed if no band is specified.
- ST_MapAlgebraFct - 2 band version - Creates a new one band raster formed by applying a valid PostgreSQL function on the 2 input raster bands and of pixeltype provided. Band 1 is assumed if no band is specified. Extent type defaults to INTERSECTION if not specified.
- ST_MapAlgebraFctNgb - 1-band version: Map Algebra Nearest Neighbor using user-defined PostgreSQL function. Return a raster which values are the result of a PLPGSQL user function involving a neighborhood of values from the input raster band.
- ST_MapAlgebra (expression version) - Expression version - Returns a one-band raster given one or two input rasters, band indexes and one or more user-specified SQL expressions.

- ST_MemSize - Returns the amount of space (in bytes) the raster takes.
- ST_MetaData - Returns basic meta data about a raster object such as pixel size, rotation (skew), upper, lower left, etc.
- ST_MinConvexHull - Return the convex hull geometry of the raster excluding NODATA pixels.
- ST_NearestValue - Returns the nearest non-NODATA value of a given band's pixel specified by a columnx and rowy or a geometric point expressed in the same spatial reference coordinate system as the raster.
- ST_Neighborhood - Returns a 2-D double precision array of the non-NODATA values around a given band's pixel specified by either a columnX and rowY or a geometric point expressed in the same spatial reference coordinate system as the raster.
- ST_NotSameAlignmentReason - Returns text stating if rasters are aligned and if not aligned, a reason why.
- ST_NumBands - Returns the number of bands in the raster object.
- ST_Overlaps - Return true if raster rastA and rastB intersect but one does not completely contain the other.
- ST_PixelAsCentroid - Returns the centroid (point geometry) of the area represented by a pixel.
- ST_PixelAsCentroids - Returns the centroid (point geometry) for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The point geometry is the centroid of the area represented by a pixel.
- ST_PixelAsPoint - Returns a point geometry of the pixel's upper-left corner.
- ST_PixelAsPoints - Returns a point geometry for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The coordinates of the point geometry are of the pixel's upper-left corner.
- ST_PixelAsPolygon - Returns the polygon geometry that bounds the pixel for a particular row and column.
- ST_PixelAsPolygons - Returns the polygon geometry that bounds every pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel.
- ST_PixelHeight - Returns the pixel height in geometric units of the spatial reference system.
- ST_PixelOfValue - Get the columnx, rowy coordinates of the pixel whose value equals the search value.
- ST_PixelWidth - Returns the pixel width in geometric units of the spatial reference system.
- ST_Polygon - Returns a multipolygon geometry formed by the union of pixels that have a pixel value that is not no data value. If no band number is specified, band num defaults to 1.
- ST_Quantile - Compute quantiles for a raster or raster table coverage in the context of the sample or population. Thus, a value could be examined to be at the raster's 25%, 50%, 75% percentile.
- ST_RastFromHexWKB - Return a raster value from a Hex representation of Well-Known Binary (WKB) raster.
- ST_RastFromWKB - Return a raster value from a Well-Known Binary (WKB) raster.
- ST_RasterToWorldCoord - Returns the raster's upper left corner as geometric X and Y (longitude and latitude) given a column and row. Column and row starts at 1.
- ST_RasterToWorldCoordX - Returns the geometric X coordinate upper left of a raster, column and row. Numbering of columns and rows starts at 1.
- ST_RasterToWorldCoordY - Returns the geometric Y coordinate upper left corner of a raster, column and row. Numbering of columns and rows starts at 1.
- ST_Reclass - Creates a new raster composed of band types reclassified from original. The nband is the band to be changed. If nband is not specified assumed to be 1. All other bands are returned unchanged. Use case: convert a 16BUI band to a 8BUI and so forth for simpler rendering as viewable formats.
- ST_ReclassExact - Creates a new raster composed of bands reclassified from original, using a 1:1 mapping from values in the original band to new values in the destination band.

- ST_Resample - Resample a raster using a specified resampling algorithm, new dimensions, an arbitrary grid corner and a set of raster georeferencing attributes defined or borrowed from another raster.
- ST_Rescale - Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the Nearest-Neighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_Resize - Resize a raster to a new width/height
- ST_Reskew - Resample a raster by adjusting only its skew (or rotation parameters). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.
- ST_Rotation - Returns the rotation of the raster in radian.
- ST_Roughness - Returns a raster with the calculated "roughness" of a DEM.
- ST_SRID - Returns the spatial reference identifier of the raster as defined in spatial_ref_sys table.
- ST_SameAlignment - Returns true if rasters have same skew, scale, spatial ref, and offset (pixels can be put on same grid without cutting into pixels) and false if they don't with notice detailing issue.
- ST_ScaleX - Returns the X component of the pixel width in units of coordinate reference system.
- ST_ScaleY - Returns the Y component of the pixel height in units of coordinate reference system.
- ST_SetBandIndex - Update the external band number of an out-db band
- ST_SetBandIsNoData - Sets the isnodata flag of the band to TRUE.
- ST_SetBandNoDataValue - Sets the value for the given band that represents no data. Band 1 is assumed if no band is specified. To mark a band as having no nodata value, set the nodata value = NULL.
- ST_SetBandPath - Update the external path and band number of an out-db band
- ST_SetGeoReference - Set Georeference 6 georeference parameters in a single call. Numbers should be separated by white space. Accepts inputs in GDAL or ESRI format. Default is GDAL.
- ST_SetM - Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimension using the requested resample algorithm.
- ST_SetRotation - Set the rotation of the raster in radian.
- ST_SetSRID - Sets the SRID of a raster to a particular integer srid defined in the spatial_ref_sys table.
- ST_SetScale - Sets the X and Y size of pixels in units of coordinate reference system. Number units/pixel width/height.
- ST_SetSkew - Sets the georeference X and Y skew (or rotation parameter). If only one is passed in, sets X and Y to the same value.
- ST_SetUpperLeft - Sets the value of the upper left corner of the pixel of the raster to projected X and Y coordinates.
- ST_SetValue - Returns modified raster resulting from setting the value of a given band in a given columnx, rowy pixel or the pixels that intersect a particular geometry. Band numbers start at 1 and assumed to be 1 if not specified.
- ST_SetValues - Returns modified raster resulting from setting the values of a given band.
- ST_SetZ - Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimension using the requested resample algorithm.
- ST_SkewX - Returns the georeference X skew (or rotation parameter).
- ST_SkewY - Returns the georeference Y skew (or rotation parameter).
- ST_Slope - Returns the slope (in degrees by default) of an elevation raster band. Useful for analyzing terrain.

- ST_SnapToGrid - Resample a raster by snapping it to a grid. New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.
- ST_Summary - Returns a text summary of the contents of the raster.
- ST_SummaryStats - Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.
- ST_SummaryStatsAgg - Aggregate. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a set of raster. Band 1 is assumed if no band is specified.
- ST_TPI - Returns a raster with the calculated Topographic Position Index.
- ST_TRI - Returns a raster with the calculated Terrain Ruggedness Index.
- ST_Tile - Returns a set of rasters resulting from the split of the input raster based upon the desired dimensions of the output rasters.
- ST_Touches - Return true if raster rastA and rastB have at least one point in common but their interiors do not intersect.
- ST_Transform - Reprojects a raster in a known spatial reference system to another known spatial reference system using specified resampling algorithm. Options are NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos defaulting to NearestNeighbor.
- ST_Union - Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- ST_UpperLeftX - Returns the upper left X coordinate of raster in projected spatial ref.
- ST_UpperLeftY - Returns the upper left Y coordinate of raster in projected spatial ref.
- ST_Value - Returns the value of a given band in a given columnx, rowy pixel or at a particular geometric point. Band numbers start at 1 and assumed to be 1 if not specified. If exclude_nodata_value is set to false, then all pixels include nodata pixels are considered to intersect and return value. If exclude_nodata_value is not passed in then reads it from metadata of raster.
- ST_ValueCount - Returns a set of records containing a pixel band value and count of the number of pixels in a given band of a raster (or a raster coverage) that have a given set of values. If no band is specified defaults to band 1. By default nodata value pixels are not counted. and all other values in the pixel are output and pixel band values are rounded to the nearest integer.
- ST_Width - Returns the width of the raster in pixels.
- ST_Within - Return true if no points of raster rastA lie in the exterior of raster rastB and at least one point of the interior of rastA lies in the interior of rastB.
- ST_WorldToRasterCoord - Returns the upper left corner as column and row given geometric X and Y (longitude and latitude) or a point geometry expressed in the spatial reference coordinate system of the raster.
- ST_WorldToRasterCoordX - Returns the column in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw) represented in world spatial reference system of raster.
- ST_WorldToRasterCoordY - Returns the row in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw) represented in world spatial reference system of raster.
- UpdateRasterSRID - Change the SRID of all rasters in the user-specified column and table.

13.6 PostGIS Geometry / Geography / Raster Dump Functions

The functions given below are PostGIS functions that take as input or return as output a set of or single geometry_dump or geomval data type object.

- ST_DumpAsPolygons - Returns a set of geomval (geom,val) rows, from a given raster band. If no band number is specified, band num defaults to 1.

- ST_Intersection - Returns a raster or a set of geometry-pixelvalue pairs representing the shared portion of two rasters or the geometrical intersection of a vectorization of the raster and a geometry.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_DumpRings - Returns a set of geometry_dump rows for the exterior and interior rings of a Polygon.
- ST_DumpSegments - Returns a set of geometry_dump rows for the segments in a geometry.

13.7 PostGIS Box Functions

The functions given below are PostGIS functions that take as input or return as output the box* family of PostGIS spatial types. The box family of types consists of box2d, and box3d

- Box2D - Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- MakeTopologyPrecise - Snap topology vertices to precision grid.
- Box3D - Returns the box 3d representation of the enclosing box of the raster.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_3DMakeBox - Creates a BOX3D defined by two 3D point geometries.
- ST_AsMVTGeom - Transforms a geometry into the coordinate space of a MVT tile.
- ST_AsTWKB - Returns the geometry as TWKB, aka "Tiny Well-Known Binary"
- ST_Box2dFromGeoHash - Return a BOX2D from a GeoHash string.
- ST_ClipByBox2D - Computes the portion of a geometry falling within a rectangle.
- ST_EstimatedExtent - Returns the estimated extent of a spatial table.
- ST_Expand - Returns a bounding box expanded from another bounding box or a geometry.
- ST_Extent - Aggregate function that returns the bounding box of geometries.
- ST_MakeBox2D - Creates a BOX2D defined by two 2D point geometries.
- ST_RemoveIrrelevantPointsForView - Removes points that are irrelevant for rendering a specific rectangular view of a geometry.
- ST_XMax - Returns the X maxima of a 2D or 3D bounding box or a geometry.
- ST_XMin - Returns the X minima of a 2D or 3D bounding box or a geometry.
- ST_XSize - Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_YMax - Returns the Y maxima of a 2D or 3D bounding box or a geometry.
- ST_YMin - Returns the Y minima of a 2D or 3D bounding box or a geometry.
- ST_YSize - Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_ZMax - Returns the Z maxima of a 2D or 3D bounding box or a geometry.
- ST_ZMin - Returns the Z minima of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Returns the Z size of a 2D or 3D bounding box or a geometry.

- RemoveUnusedPrimitives - Removes topology primitives which not needed to define existing TopoGeometry objects.
- ValidateTopology - Returns a set of validatetopology_returntype objects detailing issues with topology.
- ValidateTopologyPrecision - Returns non-precise vertices in the topology.
- ~(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).
- @(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- &&(box2df,box2df) - Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).

13.8 PostGIS Functions that support 3D

The functions given below are PostGIS functions that do not throw away the Z-Index.

- AddGeometryColumn - Adds a geometry column to an existing table.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- CG_3DAlphaWrapping - Computes a 3D Alpha-wrapping strictly enclosing a geometry.
- CG_3DArea - Computes area of 3D surface geometries. Will return 0 for solids.
- CG_3DConvexHull - Computes the 3D convex hull of a geometry.
- CG_3DDifference - Perform 3D difference
- CG_3DIntersection - Perform 3D intersection
- CG_3DRotate - Rotates a geometry in 3D space around an axis vector.
- CG_3DScale - Scales a geometry by separate factors along X, Y, and Z axes.
- CG_3DScaleAroundCenter - Scales a geometry in 3D space around a specified center point.
- CG_3DTranslate - Translates (moves) a geometry by given offsets in 3D space.
- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- CG_ConstrainedDelaunayTriangles - Return a constrained Delaunay triangulation around the given input geometry.
- CG_Extrude - Extrude a surface to a related volume
- CG_ForceLHR - Force LHR orientation

- CG_IsPlanar - Check if a surface is or not planar
- CG_IsSolid - Test if the geometry is a solid. No validity check is performed.
- CG_MakeSolid - Cast the geometry into a solid. No check is performed. To obtain a valid solid, the input geometry must be a closed Polyhedral Surface or a closed TIN.
- CG_NurbsCurveApproximate - Creates an approximating NURBS curve fitting data points within a tolerance
- CG_NurbsCurveDerivative - Computes the derivative (tangent, curvature) of a NURBS curve at a given parameter
- CG_NurbsCurveInterpolate - Creates an interpolating NURBS curve passing through all given data points
- CG_Orientation - Determine surface orientation
- CG_RotateX - Rotates a geometry around the X-axis by a given angle.
- CG_RotateY - Rotates a geometry around the Y-axis by a given angle.
- CG_RotateZ - Rotates a geometry around the Z-axis by a given angle.
- CG_Simplify - Reduces the complexity of a geometry while preserving essential features and Z/M values.
- CG_Tesselate - Perform surface Tessellation of a polygon or polyhedralsurface and returns as a TIN or collection of TINS
- CG_Visibility - Compute a visibility polygon from a point or a segment in a polygon geometry
- CG_Volume - Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- DropGeometryColumn - Removes a geometry column from a spatial table.
- GeometryType - Returns the type of a geometry as text.
- ST_3DClosestPoint - Returns the 3D point on g1 that is closest to g2. This is the first point of the 3D shortest line.
- ST_3DDFullyWithin - Tests if two 3D geometries are entirely within a given 3D distance
- ST_3DDWithin - Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_3DInterpolatePoint - Returns the interpolated measure of a geometry closest to a point in 3D.
- ST_3DIntersects - Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_3DLength - Returns the 3D length of a linear geometry.
- ST_3DLineInterpolatePoint - Returns a point interpolated along a 3D line at a fractional location.
- ST_3DLongestLine - Returns the 3D longest line between two geometries
- ST_3DMaxDistance - Returns the 3D cartesian maximum distance (based on spatial ref) between two geometries in projected units.
- ST_3DPerimeter - Returns the 3D perimeter of a polygonal geometry.
- ST_3DShortestLine - Returns the 3D shortest line between two geometries
- ST_AddMeasure - Interpolates measures along a linear geometry.
- ST_AddPoint - Add a point to a LineString.
- ST_Affine - Apply a 3D affine transformation to a geometry.

- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.
- ST_AsGML - Return the geometry as a GML version 2 or 3 element.
- ST_AsGeoJSON - Return a geometry or feature in GeoJSON format.
- ST_AsHEXEWKB - Returns a Geometry in HEXEWKB format (as text) using either little-endian (NDR) or big-endian (XDR) encoding.
- ST_AsKML - Return the geometry as a KML element.
- ST_AsX3D - Returns a Geometry in X3D xml node element format: ISO-IEC-19776-1.2-X3DEncodings-XML
- ST_Boundary - Returns the boundary of a geometry.
- ST_BoundingDiagonal - Returns the diagonal of a geometry's bounding box.
- ST_CPAWithin - Tests if the closest point of approach of two trajectoriesis within the specified distance.
- ST_CatmullRomSmoothing - Returns a smoothed version of a geometry, using the Catmull-Rom spline algorithm
- ST_ChaikinSmoothing - Returns a smoothed version of a geometry, using the Chaikin algorithm
- ST_ClosestPointOfApproach - Returns a measure at the closest point of approach of two trajectories.
- ST_Collect - Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_ConvexHull - Computes the convex hull of a geometry.
- ST_CoordDim - Return the coordinate dimension of a geometry.
- ST_CurveN - Returns the Nth component curve geometry of a CompoundCurve.
- ST_CurveToLine - Converts a geometry containing curves to a linear geometry.
- ST_DelaunayTriangles - Returns the Delaunay triangulation of the vertices of a geometry.
- ST_Difference - Computes a geometry representing the part of geometry A that does not intersect geometry B.
- ST_DistanceCPA - Returns the distance between the closest point of approach of two trajectories.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_DumpRings - Returns a set of geometry_dump rows for the exterior and interior rings of a Polygon.
- ST_DumpSegments - Returns a set of geometry_dump rows for the segments in a geometry.
- ST_EndPoint - Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_ExteriorRing - Returns a LineString representing the exterior ring of a Polygon.
- ST_FlipCoordinates - Returns a version of a geometry with X and Y axis flipped.
- ST_Force2D - Force the geometries into a "2-dimensional mode".
- ST_ForceCurve - Upcast a geometry into its curved type, if applicable.
- ST_ForcePolygonCCW - Orients all exterior rings counter-clockwise and all interior rings clockwise.
- ST_ForcePolygonCW - Orients all exterior rings clockwise and all interior rings counter-clockwise.
- ST_ForceRHR - Force the orientation of the vertices in a polygon to follow the Right-Hand-Rule.

- ST_ForceSFS - Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_Force3D - Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.
- ST_Force3DZ - Force the geometries into XYZ mode.
- ST_Force4D - Force the geometries into XYZM mode.
- ST_ForceCollection - Convert the geometry into a GEOMETRYCOLLECTION.
- ST_GeomFromEWKB - Return a specified ST_Geometry value from Extended Well-Known Binary representation (EWKB).
- ST_GeomFromEWKT - Return a specified ST_Geometry value from Extended Well-Known Text representation (EWKT).
- ST_GeomFromGML - Takes as input GML representation of geometry and outputs a PostGIS geometry object
- ST_GeomFromGeoJSON - Takes as input a geojson representation of a geometry and outputs a PostGIS geometry object
- ST_GeomFromKML - Takes as input KML representation of geometry and outputs a PostGIS geometry object
- ST_GeometricMedian - Returns the geometric median of a MultiPoint.
- ST_GeometryN - Return an element of a geometry collection.
- ST_GeometryType - Returns the SQL-MM type of a geometry as text.
- ST_HasArc - Tests if a geometry contains a circular arc
- ST_HasM - Checks if a geometry has an M (measure) dimension.
- ST_HasZ - Checks if a geometry has a Z dimension.
- ST_InteriorRingN - Returns the Nth interior ring (hole) of a Polygon.
- ST_InterpolatePoint - Returns the interpolated measure of a geometry closest to a point.
- ST_Intersection - Computes a geometry representing the shared portion of geometries A and B.
- ST_IsClosed - Tests if a LineStrings's start and end points are coincident. For a PolyhedralSurface tests if it is closed (volumetric).
- ST_IsCollection - Tests if a geometry is a geometry collection type.
- ST_IsPolygonCCW - Tests if Polygons have exterior rings oriented counter-clockwise and interior rings oriented clockwise.
- ST_IsPolygonCW - Tests if Polygons have exterior rings oriented clockwise and interior rings oriented counter-clockwise.
- ST_IsSimple - Tests if a geometry has no points of self-intersection or self-tangency.
- ST_IsValidTrajectory - Tests if the geometry is a valid trajectory.
- ST_LengthSpheroid - Returns the 2D or 3D length/perimeter of a lon/lat geometry on a spheroid.
- ST_LineFromMultiPoint - Creates a LineString from a MultiPoint geometry.
- ST_LineInterpolatePoint - Returns a point interpolated along a line at a fractional location.
- ST_LineInterpolatePoints - Returns points interpolated along a line at a fractional interval.
- ST_LineSubstring - Returns the part of a line between two fractional locations.
- ST_LineToCurve - Converts a linear geometry to a curved geometry.
- ST_LocateBetweenElevations - Returns the portions of a geometry that lie in an elevation (Z) range.
- ST_M - Returns the M coordinate of a Point.
- ST_MMax - Returns the M maxima of a geometry.

- ST_MMin - Returns the M minima of a geometry.
- ST_MSize - Returns the M size of a geometry.
- ST_MakeLine - Creates a LineString from Point, MultiPoint, or LineString geometries.
- ST_MakePoint - Creates a 2D, 3DZ or 4D Point.
- ST_MakePolygon - Creates a Polygon or CurvePolygon from a shell and optional list of holes.
- ST_MakeValid - Attempts to make an invalid geometry valid without losing vertices.
- ST_MemSize - Returns the amount of memory space a geometry takes.
- ST_MemUnion - Aggregate function which unions geometries in a memory-efficent but slower way
- ST_NDims - Returns the coordinate dimension of a geometry.
- ST_NPoints - Returns the number of points (vertices) in a geometry.
- ST_NRings - Returns the number of rings in a polygonal geometry.
- ST_Node - Nodes a collection of lines.
- ST_NumCurves - Return the number of component curves in a CompoundCurve.
- ST_NumGeometries - Returns the number of elements in a geometry collection.
- ST_NumPatches - Returns the number of faces in a PolyhedralSurface or TIN.
- ST_PatchN - Returns the Nth face of a PolyhedralSurface or TIN.
- ST_PointFromWKB - Makes a geometry from WKB with the given SRID
- ST_PointN - Returns the Nth point in the first LineString or circular LineString in a geometry.
- ST_PointOnSurface - Computes a point guaranteed to lie in a polygon, or on a geometry.
- ST_Points - Returns a MultiPoint containing the coordinates of a geometry.
- ST_Polygon - Creates a Polygon from a LineString with a specified SRID.
- ST_RemovePoint - Remove a point from a linestring.
- ST_RemoveRepeatedPoints - Returns a version of a geometry with duplicate points removed.
- ST_Reverse - Return the geometry with vertex order reversed.
- ST_Rotate - Rotates a geometry about an origin point.
- ST_RotateX - Rotates a geometry about the X axis.
- ST_RotateY - Rotates a geometry about the Y axis.
- ST_RotateZ - Rotates a geometry about the Z axis.
- ST_Scale - Scales a geometry by given factors.
- ST_Scroll - Change start point of a closed LineString.
- ST_SetPoint - Replace point of a linestring with a given point.
- ST_ShiftLongitude - Shifts the longitude coordinates of a geometry between -180..180 and 0..360.
- ST_SnapToGrid - Snap all points of the input geometry to a regular grid.
- ST_StartPoint - Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_SwapOrdinates - Returns a version of the given geometry with given ordinate values swapped.

- ST_SymDifference - Computes a geometry representing the portions of geometries A and B that do not intersect.
- ST_TransScale - Translates and scales a geometry by given offsets and factors.
- ST_Translate - Translates a geometry by given offsets.
- ST_UnaryUnion - Computes the union of the components of a single geometry.
- ST_Union - Computes a geometry representing the point-set union of the input geometries.
- ST_WrapX - Wrap a geometry around an X value.
- ST_X - Returns the X coordinate of a Point.
- ST_XMax - Returns the X maxima of a 2D or 3D bounding box or a geometry.
- ST_XMin - Returns the X minima of a 2D or 3D bounding box or a geometry.
- ST_XSize - Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_Y - Returns the Y coordinate of a Point.
- ST_YMax - Returns the Y maxima of a 2D or 3D bounding box or a geometry.
- ST_YMin - Returns the Y minima of a 2D or 3D bounding box or a geometry.
- ST_YSize - Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_Z - Returns the Z coordinate of a Point.
- ST_ZMFlag - Returns a code indicating the ZM coordinate dimension of a geometry.
- ST_ZMax - Returns the Z maxima of a 2D or 3D bounding box or a geometry.
- ST_ZMin - Returns the Z minima of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Returns the Z size of a 2D or 3D bounding box or a geometry.
- Equals - Returns true if two topogeometries are composed of the same topology primitives.
- Intersects - Returns true if any pair of primitives from the two topogeometries intersect.
- UpdateGeometrySRID - Updates the SRID of all features in a geometry column, and the table metadata.
- &&& - Returns TRUE if A's n-D bounding box intersects B's n-D bounding box.
- &&&(geometry,gidx) - Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

13.9 PostGIS Curved Geometry Support Functions

The functions given below are PostGIS functions that can use CIRCULARSTRING, CURVEPOLYGON, and other curved geometry types

- AddGeometryColumn - Adds a geometry column to an existing table.
- Box2D - Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.

- DropGeometryColumn - Removes a geometry column from a spatial table.
- GeometryType - Returns the type of a geometry as text.
- PostGIS_AddBBox - Add bounding box to the geometry.
- PostGIS_DropBBox - Drop the bounding box cache from the geometry.
- PostGIS_HasBBox - Returns TRUE if the bbox of this geometry is cached, FALSE otherwise.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_Affine - Apply a 3D affine transformation to a geometry.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.
- ST_AsHEXEWKB - Returns a Geometry in HEXEWKB format (as text) using either little-endian (NDR) or big-endian (XDR) encoding.
- ST_AsSVG - Returns SVG path data for a geometry.
- ST_AsText - Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.
- ST_ClusterDBSCAN - Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterWithin - Aggregate function that clusters geometries by separation distance.
- ST_ClusterWithinWin - Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_Collect - Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_CoordDim - Return the coordinate dimension of a geometry.
- ST_CurveToLine - Converts a geometry containing curves to a linear geometry.
- ST_Distance - Returns the distance between two geometry or geography values.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_EndM - Returns the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_EndPoint - Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_EstimatedExtent - Returns the estimated extent of a spatial table.
- ST_FlipCoordinates - Returns a version of a geometry with X and Y axis flipped.
- ST_Force2D - Force the geometries into a "2-dimensional mode".
- ST_ForceCurve - Upcast a geometry into its curved type, if applicable.
- ST_ForceSFS - Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_Force3D - Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.
- ST_Force3DM - Force the geometries into XYM mode.
- ST_Force3DZ - Force the geometries into XYZ mode.
- ST_Force4D - Force the geometries into XYZM mode.
- ST_ForceCollection - Convert the geometry into a GEOMETRYCOLLECTION.

- ST_GeoHash - Return a GeoHash representation of the geometry.
- ST_GeogFromWKB - Creates a geography instance from a Well-Known Binary geometry representation (WKB) or extended Well Known Binary (EWKB).
- ST_GeomFromEWKB - Return a specified ST_Geometry value from Extended Well-Known Binary representation (EWKB).
- ST_GeomFromEWKT - Return a specified ST_Geometry value from Extended Well-Known Text representation (EWKT).
- ST_GeomFromText - Return a specified ST_Geometry value from Well-Known Text representation (WKT).
- ST_GeomFromWKB - Creates a geometry instance from a Well-Known Binary geometry representation (WKB) and optional SRID.
- ST_GeometryN - Return an element of a geometry collection.
- = - Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.
- &<| - Returns TRUE if A's bounding box overlaps or is below B's.
- ST_HasArc - Tests if a geometry contains a circular arc
- ST_Intersects - Tests if two geometries intersect (they have at least one point in common)
- ST_IsClosed - Tests if a LineStrings's start and end points are coincident. For a PolyhedralSurface tests if it is closed (volumetric).
- ST_IsCollection - Tests if a geometry is a geometry collection type.
- ST_IsEmpty - Tests if a geometry is empty.
- ST_LineToCurve - Converts a linear geometry to a curved geometry.
- ST_MMax - Returns the M maxima of a geometry.
- ST_MMin - Returns the M minima of a geometry.
- ST_MSize - Returns the M size of a geometry.
- ST_MemSize - Returns the amount of memory space a geometry takes.
- ST_NPoints - Returns the number of points (vertices) in a geometry.
- ST_NRings - Returns the number of rings in a polygonal geometry.
- ST_PointFromWKB - Makes a geometry from WKB with the given SRID
- ST_PointN - Returns the Nth point in the first LineString or circular LineString in a geometry.
- ST_Points - Returns a MultiPoint containing the coordinates of a geometry.
- ST_Rotate - Rotates a geometry about an origin point.
- ST_RotateZ - Rotates a geometry about the Z axis.
- ST_SRID - Returns the spatial reference identifier for a geometry.
- ST_Scale - Scales a geometry by given factors.
- ST_SetEndM - Sets the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_SetSRID - Set the SRID on a geometry.
- ST_SetStartM - Sets the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_StartM - Returns the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.

- ST_StartPoint - Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_Summary - Returns a text summary of the contents of a geometry.
- ST_SwapOrdinates - Returns a version of the given geometry with given ordinate values swapped.
- ST_TransScale - Translates and scales a geometry by given offsets and factors.
- ST_Transform - Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Translate - Translates a geometry by given offsets.
- ST_XMax - Returns the X maxima of a 2D or 3D bounding box or a geometry.
- ST_XMin - Returns the X minima of a 2D or 3D bounding box or a geometry.
- ST_XSize - Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_YMax - Returns the Y maxima of a 2D or 3D bounding box or a geometry.
- ST_YMin - Returns the Y minima of a 2D or 3D bounding box or a geometry.
- ST_YSize - Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_ZMFlag - Returns a code indicating the ZM coordinate dimension of a geometry.
- ST_ZMax - Returns the Z maxima of a 2D or 3D bounding box or a geometry.
- ST_ZMin - Returns the Z minima of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Returns the Z size of a 2D or 3D bounding box or a geometry.
- UpdateGeometrySRID - Updates the SRID of all features in a geometry column, and the table metadata.
- ~(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).
- && - Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.
- &&& - Returns TRUE if A's n-D bounding box intersects B's n-D bounding box.
- @(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- &&(box2df,box2df) - Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).
- &&&(geometry,gidx) - Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

13.10 PostGIS Polyhedral Surface Support Functions

The functions given below are PostGIS functions that can use POLYHEDRALSURFACE, POLYHEDRALSURFACEM geometries

- Box2D - Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- CG_3DArea - Computes area of 3D surface geometries. Will return 0 for solids.
- CG_3DConvexHull - Computes the 3D convex hull of a geometry.
- CG_3DDifference - Perform 3D difference
- CG_3DIntersection - Perform 3D intersection
- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- CG_ApproximateMedialAxis - Compute the approximate medial axis of an areal geometry.
- CG_Extrude - Extrude a surface to a related volume
- CG_ForceLHR - Force LHR orientation
- CG_GenerateFlatRoof - Generate a flat (box) roof from a footprint polygon.
- CG_GenerateGableRoof - Generate a gable roof from a footprint polygon.
- CG_GenerateHippedRoof - Generate a hipped roof from a footprint polygon.
- CG_GenerateRoof - Generate a roof of the requested type from a footprint polygon.
- CG_GenerateSkillionRoof - Generate a skillion (single-slope) roof from a footprint polygon.
- CG_IsPlanar - Check if a surface is or not planar
- CG_IsSolid - Test if the geometry is a solid. No validity check is performed.
- CG_MakeSolid - Cast the geometry into a solid. No check is performed. To obtain a valid solid, the input geometry must be a closed Polyhedral Surface or a closed TIN.
- CG_PolygonRepair - Repair an invalid polygon or multipolygon.
- CG_StraightSkeleton - Compute a straight skeleton from a geometry
- CG_Tesselate - Perform surface Tessellation of a polygon or polyhedralsurface and returns as a TIN or collection of TINS
- CG_Visibility - Compute a visibility polygon from a point or a segment in a polygon geometry
- CG_Volume - Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- GeometryType - Returns the type of a geometry as text.
- ST_3DClosestPoint - Returns the 3D point on g1 that is closest to g2. This is the first point of the 3D shortest line.
- ST_3DDFullyWithin - Tests if two 3D geometries are entirely within a given 3D distance
- ST_3DDWithin - Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_3DIntersects - Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)

- ST_3DLongestLine - Returns the 3D longest line between two geometries
- ST_3DMaxDistance - Returns the 3D cartesian maximum distance (based on spatial ref) between two geometries in projected units.
- ST_3DShortestLine - Returns the 3D shortest line between two geometries
- ST_Affine - Apply a 3D affine transformation to a geometry.
- ST_Area - Returns the area of a polygonal geometry.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.
- ST_AsGML - Return the geometry as a GML version 2 or 3 element.
- ST_AsX3D - Returns a Geometry in X3D xml node element format: ISO-IEC-19776-1.2-X3DEncodings-XML
- ST_CoordDim - Return the coordinate dimension of a geometry.
- ST_Dimension - Returns the topological dimension of a geometry.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_Expand - Returns a bounding box expanded from another bounding box or a geometry.
- ST_Extent - Aggregate function that returns the bounding box of geometries.
- ST_FlipCoordinates - Returns a version of a geometry with X and Y axis flipped.
- ST_Force2D - Force the geometries into a "2-dimensional mode".
- ST_ForceRHR - Force the orientation of the vertices in a polygon to follow the Right-Hand-Rule.
- ST_ForceSFS - Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_Force3D - Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.
- ST_Force3DZ - Force the geometries into XYZ mode.
- ST_ForceCollection - Convert the geometry into a GEOMETRYCOLLECTION.
- ST_GeomFromEWKB - Return a specified ST_Geometry value from Extended Well-Known Binary representation (EWKB).
- ST_GeomFromEWKT - Return a specified ST_Geometry value from Extended Well-Known Text representation (EWKT).
- ST_GeomFromGML - Takes as input GML representation of geometry and outputs a PostGIS geometry object
- ST_GeometryN - Return an element of a geometry collection.
- ST_GeometryType - Returns the SQL-MM type of a geometry as text.
- = - Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.
- &<| - Returns TRUE if A's bounding box overlaps or is below B's.
- ~= - Returns TRUE if A's bounding box is the same as B's.
- ST_IsClosed - Tests if a LineStrings's start and end points are coincident. For a PolyhedralSurface tests if it is closed (volumetric).

- ST_MemSize - Returns the amount of memory space a geometry takes.
- ST_NPoints - Returns the number of points (vertices) in a geometry.
- ST_NumGeometries - Returns the number of elements in a geometry collection.
- ST_NumPatches - Returns the number of faces in a PolyhedralSurface or TIN.
- ST_PatchN - Returns the Nth face of a PolyhedralSurface or TIN.
- ST_RemoveRepeatedPoints - Returns a version of a geometry with duplicate points removed.
- ST_Reverse - Return the geometry with vertex order reversed.
- ST_Rotate - Rotates a geometry about an origin point.
- ST_RotateX - Rotates a geometry about the X axis.
- ST_RotateY - Rotates a geometry about the Y axis.
- ST_RotateZ - Rotates a geometry about the Z axis.
- ST_Scale - Scales a geometry by given factors.
- ST_ShiftLongitude - Shifts the longitude coordinates of a geometry between -180..180 and 0..360.
- ST_Summary - Returns a text summary of the contents of a geometry.
- ST_SwapOrdinates - Returns a version of the given geometry with given ordinate values swapped.
- ST_Transform - Return a new geometry with coordinates transformed to a different spatial reference system.
- ~(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).
- && - Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.
- &&& - Returns TRUE if A's n-D bounding box intersects B's n-D bounding box.
- @(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- &&(box2df,box2df) - Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).
- &&&(geometry,gidx) - Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

13.11 PostGIS Function Support Matrix

Below is an alphabetical listing of spatial specific functions in PostGIS and the kinds of spatial types they work with or OGC/SQL compliance they try to conform to.

- A ✓ means the function works with the type or subtype natively.
- A  means it works but with a transform cast built-in using cast to geometry, transform to a "best srid" spatial ref and then cast back. Results may not be as expected for large areas or areas at poles and may accumulate floating point junk.
- A  means the function works with the type because of a auto-cast to another such as to box3d rather than direct type support.
- A  means the function only available if PostGIS compiled with SFCGAL support.
- geom - Basic 2D geometry support (x,y).
- geog - Basic 2D geography support (x,y).
- 2.5D - basic 2D geometries in 3 D/4D space (has Z or M coord).
- PS - Polyhedral surfaces
- T - Triangles and Triangulated Irregular Network surfaces (TIN)

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Collect	✓		✓	✓			
ST_LineFromMulti	✓		✓				
ST_MakeEnvelope	✓						
ST_MakeLine	✓		✓				
ST_MakePoint	✓		✓				
ST_MakePointM	✓						
ST_MakePolygon	✓		✓				
ST_Point	✓				✓		
ST_PointZ	✓						
ST_PointM	✓						
ST_PointZM	✓						
ST_Polygon	✓		✓		✓		
ST_TileEnvelope	✓						
ST_HexagonGrid	✓						
ST_Hexagon	✓						
ST_SquareGrid	✓						
ST_Square	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Letters	✓						
ST_MakeNurbsCur	✓						
GeometryType	✓		✓	✓		✓	✓
ST_Boundary	✓		✓		✓		
ST_BoundingDiago	✓		✓				
ST_CoordDim	✓		✓	✓	✓	✓	✓
ST_Dimension	✓				✓	✓	✓
ST_Dump	✓		✓	✓		✓	✓
ST_DumpPoints	✓		✓	✓		✓	✓
ST_DumpSegments	✓		✓				✓
ST_DumpRings	✓		✓				
ST_EndPoint	✓		✓	✓	✓		
ST_Envelope	✓				✓		
ST_ExteriorRing	✓		✓		✓		
ST_GeometryN	✓		✓	✓	✓	✓	✓
ST_GeometryType	✓		✓		✓	✓	
ST_HasArc	✓		✓	✓			
ST_InteriorRingN	✓		✓		✓		
ST_NumCurves	✓		✓		✓		
ST_CurveN	✓		✓		✓		
ST_IsClosed	✓		✓	✓	✓	✓	
ST_IsCollection	✓		✓	✓			
ST_IsEmpty	✓			✓	✓		
ST_IsPolygonCCW	✓		✓				
ST_IsPolygonCW	✓		✓				
ST_IsRing	✓				✓		
ST_IsSimple	✓		✓		✓		
ST_M	✓		✓		✓		
ST_MemSize	✓		✓	✓		✓	✓
ST_NDims	✓		✓				
ST_NPoints	✓		✓	✓		✓	
ST_NRings	✓		✓	✓			
ST_NumGeometrie	✓		✓		✓	✓	✓
ST_NumInteriorRin	✓				✓		

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_NumInteriorRin	✓						
ST_NumPatches	✓		✓		✓	✓	
ST_NumPoints	✓				✓		
ST_PatchN	✓		✓		✓	✓	✓
ST_PointN	✓		✓	✓	✓		
ST_Points	✓		✓	✓			
ST_StartPoint	✓		✓	✓	✓		
ST_StartM	✓			✓			
ST_EndM	✓			✓			
ST_SetStartM	✓			✓			
ST_SetEndM	✓			✓			
ST_Summary	✓	✓		✓		✓	✓
ST_X	✓		✓		✓		
ST_Y	✓		✓		✓		
ST_Z	✓		✓		✓		
ST_ZMFlag	✓		✓	✓			
ST_HasZ	✓		✓				
ST_HasM	✓		✓				
ST_ControlPoints	✓						
ST_Degree	✓						
ST_Weights	✓						
ST_Knots	✓						
ST_NumControlPoi	✓						
ST_NurbsCurveIsR	✓ ıal						
ST_IsNurbsCurve	✓						
ST_Evaluate	✓						
ST_NurbsToLineStr	✓						
ST_AddPoint	✓		✓				
ST_CollectionExtra	✓						
ST_CollectionHom	✓ ıze						
ST_CurveToLine	✓		✓	✓	✓		
ST_Scroll	✓		✓				
ST_FlipCoordinates	✓		✓	✓		✓	✓
ST_Force2D	✓		✓	✓		✓	

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Force3D	✓		✓	✓		✓	
ST_Force3DZ	✓		✓	✓		✓	
ST_Force3DM	✓			✓			
ST_Force4D	✓		✓	✓			
ST_ForceCollection	✓		✓	✓		✓	
ST_ForceCurve	✓		✓	✓			
ST_ForcePolygonC	✓		✓				
ST_ForcePolygonC	✓		✓				
ST_ForceSFS	✓		✓	✓		✓	✓
ST_ForceRHR	✓		✓			✓	
ST_LineExtend	✓						
ST_LineToCurve	✓		✓	✓			
ST_Multi	✓						
ST_Normalize	✓						
ST_Project	✓	✓					
ST_QuantizeCoordi	✓ s						
ST_RemovePoint	✓		✓				
ST_RemoveRepeate	✓ ints		✓			✓	
ST_RemoveIrreleva	✓ intsForView						
ST_RemoveSmallP	✓						
ST_Reverse	✓		✓			✓	
ST_Segmentize	✓	✓					
ST_SetPoint	✓		✓				
ST_ShiftLongitude	✓		✓			✓	✓
ST_WrapX	✓		✓				
ST_SnapToGrid	✓		✓				
ST_Snap	✓						
ST_SwapOrdinates	✓		✓	✓		✓	✓
ST_IsValid	✓				✓		
ST_IsValidDetail	✓						
ST_IsValidReason	✓						
ST_MakeValid	✓		✓				
ST_InverseTransfor	✓ peline						
ST_SetSRID	✓			✓			

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_SRID	✓			✓	✓		
ST_Transform	✓			✓	✓	✓	
ST_TransformPipel	✓						
postgis_srs_codes							
postgis_srs							
postgis_srs_all							
postgis_srs_search	✓						
ST_BdPolyFromTe	✓						
ST_BdMPolyFromT	✓						
ST_GeogFromText		✓					
ST_GeographyFromText		✓					
ST_GeomCollFrom	✓				✓		
ST_GeomFromEWI	✓		✓	✓		✓	✓
ST_GeomFromMA	✓ 1						
ST_GeometryFrom	✓				✓		
ST_GeomFromText	✓			✓	✓		
ST_LineFromText	✓				✓		
ST_MLineFromTex	✓				✓		
ST_MPointFromTe	✓				✓		
ST_MPolyFromTex	✓				✓		
ST_PointFromText	✓				✓		
ST_PolygonFromTe	✓				✓		
ST_WKTToSQL	✓				✓		
ST_GeogFromWKB		✓		✓			
ST_GeomFromEWI	✓		✓	✓		✓	✓
ST_GeomFromWK	✓			✓	✓		
ST_LineFromWKB	✓				✓		
ST_LinestringFrom	✓ B				✓		
ST_PointFromWKB	✓		✓	✓	✓		
ST_WKBToSQL	✓				✓		
ST_Box2dFromGeo	✓ h						
ST_GeomFromGeo	✓ n						
ST_GeomFromGM	✓		✓			✓	✓
ST_GeomFromGeo	✓ N		✓				
ST_GeomFromKM	✓		✓				

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_GeomFromTWKB	✓						
ST_GMLToSQL	✓				✓		
ST_LineFromEncodedPolyline	✓						
ST_PointFromGeoHash							
ST_FromFlatGeobufToTable							
ST_FromFlatGeobuf							
ST_AsEWKT	✓	✓	✓	✓		✓	✓
ST_AsText	✓	✓		✓	✓		
ST_AsBinary	✓	✓	✓	✓	✓	✓	✓
ST_AsEWKB	✓		✓	✓		✓	✓
ST_AsHEXEWKB	✓		✓	✓			
ST_AsEncodedPolyline	✓						
ST_AsFlatGeobuf	✗						
ST_AsGeobuf	✗						
ST_AsGeoJSON	✓	✓	✓				
ST_AsGML	✓	✓	✓		✓	✓	✓
ST_AsKML	✓	✓	✓				
ST_AsLatLonText	✓						
ST_AsMARC21	✓						
ST_AsMVTGeom	✓						
ST_AsMVT	✗						
ST_AsSVG	✓	✓		✓			
ST_AsTWKB	✓						
ST_AsX3D	✓		✓			✓	✓
ST_GeoHash	✓			✓			
&&	✓	✓		✓		✓	
&&(geometry,box2df)	✓			✓		✓	
&&(box2df,geometry)	✓			✓		✓	
&&(box2df,box2df)	✗			✓		✓	
&&&	✓		✓	✓		✓	✓
&&&(geometry,gidx)	✓		✓	✓		✓	✓
&&&(gidx,geometry)	✓		✓	✓		✓	✓
&&&(gidx,gidx)			✓	✓		✓	✓
&<	✓						
&<|	✓			✓		✓	

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
&>	✓						
<<	✓						
<<|	✓						
=	✓	✓		✓		✓	
>>	✓						
@	✓						
@(geometry,box2df	✓			✓		✓	
@(box2df,geometry	✓			✓		✓	
@(box2df,box2df)	✓			✓		✓	
|&>	✓						
|>>	✓						
~	✓						
~(geometry,box2df)	✓			✓		✓	
~(box2df,geometry)	✓			✓		✓	
~(box2df,box2df)	✓			✓		✓	
~=	✓					✓	
<->	✓	✓					
|=|	✓						
<#>	✓						
<<->>	✓						
ST_3DIntersects	✓		✓		✓	✓	✓
ST_Contains	✓				✓		
ST_ContainsProper	✓						
ST_CoveredBy	✓	✓					
ST_Covers	✓	✓					
ST_Crosses	✓				✓		
ST_Disjoint	✓				✓		
ST_Equals	✓				✓		
ST_Intersects	✓	✓		✓	✓		✓
ST_LineCrossingDi ion	✓						
ST_OrderingEquals	✓				✓		
ST_Overlaps	✓				✓		
ST_Relate	✓				✓		
ST_RelateMatch							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Touches	✓				✓		
ST_Within	✓				✓		
ST_3DDWithin	✓		✓		✓	✓	
ST_3DDFullyWithi	✓		✓			✓	
ST_DFullyWithin	✓						
ST_DWithin	✓	✓					
ST_PointInsideCirc	✓						
ST_Area	✓	✓			✓	✓	
ST_Azimuth	✓	✓					
ST_Angle	✓						
ST_ClosestPoint	✓	✓					
ST_3DClosestPoint	✓		✓			✓	
ST_Distance	✓	✓		✓	✓		
ST_3DDistance	✓		✓		✓	✓	
ST_DistanceSphere	✓						
ST_DistanceSphero	✓						
ST_FrechetDistance	✓						
ST_HausdorffDistar	✓						
ST_Length	✓	✓			✓		
ST_Length2D	✓						
ST_3DLength	✓		✓		✓		
ST_LengthSpheroid	✓		✓				
ST_LongestLine	✓						
ST_3DLongestLine	✓		✓			✓	
ST_MaxDistance	✓						
ST_3DMaxDistance	✓		✓			✓	
ST_MinimumClear	✓						
ST_MinimumClear	✓ Line						
ST_Perimeter	✓	✓			✓		
ST_Perimeter2D	✓						
ST_3DPerimeter	✓		✓		✓		
ST_ShortestLine	✓	✓					
ST_3DShortestLine	✓		✓			✓	
ST_ClipByBox2D	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Difference	✓		✓		✓		
ST_Intersection	✓		✓		✓		
ST_MemUnion	✓		✓				
ST_Node	✓		✓				
ST_Split	✓						
ST_Subdivide	✓						
ST_SymDifference	✓		✓		✓		
ST_UnaryUnion	✓		✓				
ST_Union	✓		✓		✓		
ST_Buffer	✓				✓		
ST_BuildArea	✓						
ST_Centroid	✓	✓			✓		
ST_ChaikinSmooth	✓		✓				
ST_CatmullRomSm	✓ ing		✓				
ST_ConcaveHull	✓						
ST_ConvexHull	✓		✓		✓		
ST_DelaunayTriang	✓		✓				✓
ST_FilterByM	✓						
ST_GeneratePoints	✓						
ST_GeometricMedi	✓		✓				
ST_LineMerge	✓						
ST_MaximumInscri	✓ Circle						
ST_LargestEmptyC	✓						
ST_MinimumBoun	✓ Circle						
ST_MinimumBoun	✓ Radius						
ST_OrientedEnvelo	✓						
ST_OffsetCurve	✓						
ST_PointOnSurface	✓		✓		✓		
ST_Polygonize	✓						
ST_ReducePrecisio	✓						
ST_SharedPaths	✓						
ST_Simplify	✓						
ST_SimplifyPreserv	✓ pology						
ST_SimplifyPolygo	✓ ll						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_SimplifyVW	✓						
ST_SetEffectiveAre	✓						
ST_TriangulatePoly	✓						
ST_VoronoiLines	✓						
ST_VoronoiPolygor	✓						
ST_CoverageInvalic	✓	ges					
ST_CoverageClean	✓						
ST_CoverageSimpli	✓						
ST_CoverageEdges	✓						
ST_CoverageUnion	✓						
ST_Affine	✓		✓	✓		✓	✓
ST_Rotate	✓		✓	✓		✓	✓
ST_RotateX	✓		✓			✓	✓
ST_RotateY	✓		✓			✓	✓
ST_RotateZ	✓		✓	✓		✓	✓
ST_Scale	✓		✓	✓		✓	✓
ST_Translate	✓		✓	✓			
ST_TransScale	✓		✓	✓			
ST_ClusterDBSCA	✓			✓			
ST_ClusterIntersect	✓						
ST_ClusterIntersect	✓	Vin					
ST_ClusterRelateW	✓						
ST_MinimumSpanr	✓	Tree					
ST_ClusterKMeans	✓						
ST_ClusterWithin	✓			✓			
ST_ClusterWithinW	✓			✓			
Box2D	✓			✓		✓	✓
Box3D	✓		✓	✓		✓	✓
ST_EstimatedExten	✓			✓			
ST_Expand	✓					✓	✓
ST_Extent	✓					✓	✓
ST_3DExtent	✓		✓	✓		✓	✓
ST_MakeBox2D	✓						
ST_3DMakeBox	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_XMax	✓		✓	✓			
ST_XMin	✓		✓	✓			
ST_YMax	✓		✓	✓			
ST_YMin	✓		✓	✓			
ST_ZMax	✓		✓	✓			
ST_ZMin	✓		✓	✓			
ST_XSize	✓		✓	✓			
ST_YSize	✓		✓	✓			
ST_ZSize	✓		✓	✓			
ST_MMin	✓		✓	✓			
ST_MMax	✓		✓	✓			
ST_MSize	✓		✓	✓			
ST_LineInterpolatel	✓ t	✓	✓				
ST_3DLineInterpol	✓ oint		✓				
ST_LineInterpolatel	✓ ts	✓	✓				
ST_LineLocatePoin	✓	✓					
ST_LineSubstring	✓	✓	✓				
ST_LocateAlong	✓				✓		
ST_LocateBetween	✓				✓		
ST_LocateBetweenI	✓ ations		✓				
ST_InterpolatePoint	✓		✓				
ST_3DInterpolatePo	✓		✓				
ST_AddMeasure	✓		✓				
ST_IsValidTrajecto	✓		✓				
ST_ClosestPointOf	✓ oach		✓				
ST_DistanceCPA	✓		✓				
ST_CPAWithin	✓		✓				
postgis.gdal_datapath							
postgis.gdal_enabled_drivers							
postgis.enable_outdb_rasters							
postgis.gdal_vsi_options							
postgis.gdal_cpl_debug							
PostGIS_AddBBox	✓			✓			
PostGIS_DropBBox	✓			✓			
PostGIS_HasBBox	✓			✓			
postgis_sfcgal_version							
postgis_sfcgal_full_version							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
CG_ForceLHR							
CG_IsPlanar							
CG_IsSolid							
CG_MakeSolid							
CG_Orientation							
CG_Area							
CG_3DArea							
CG_Volume							
CG_Intersection							
CG_Intersects							
CG_3DIntersects							
CG_Difference							
CG_3DDifference							
CG_Distance							
CG_3DDistance							
CG_3DConvexHul							
CG_3DIntersectior							
CG_Union							
CG_3DUnion							
CG_AlphaShape							
CG_ApproxConve	ion						
CG_ApproximateM	Axis						
CG_GenerateFlatR							
CG_GenerateHipp	f						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
CG_GenerateGabl							
CG_GenerateSkilli	 of						
CG_GenerateRoof							
CG_PolygonRepai							
CG_ConstrainedD	 yTriangles						
CG_Extrude							
CG_ExtrudeStraigl	 eton						
CG_GreeneApprox	 exPartition						
CG_MinkowskiSu							
CG_OptimalAlpha							
CG_OptimalConve	 tion						
CG_StraightSkelet							
CG_Tesselate							
CG_Triangulate							
CG_Visibility							
CG_YMonotonePa							
CG_StraightSkelet	 ition						
CG_3DBuffer							
CG_Rotate							
CG_2DRotate							
CG_3DRotate							
CG_RotateX							
CG_RotateY							
CG_RotateZ							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
CG_Scale							
CG_3DScale							
CG_3DScaleAroun ter							
CG_Translate							
CG_3DTranslate							
CG_Simplify							
CG_3DAlphaWrap							
CG_NurbsCurveIn ate							
CG_NurbsCurveA mate							
CG_NurbsCurveD ve							
getfaceedges_returntype							
TopoGeometry							
validatetopology_returntype							
TopoElement							
TopoElementArray							
AddTopoGeometryColumn							
RenameTopoGeometryColumn							
DropTopology							
RenameTopology							
DropTopoGeometryColumn							
FixCorruptTopoGeometryColumn							
Populate_Topology_Layer							
TopologySummary							
ValidateTopology	✓						
ValidateTopologyRelation							
ValidateTopologyPr ion	✓						
MakeTopologyPreci	✓						
FindTopology	✓						
FindLayer	✓						
TotalTopologySize							
UpgradeTopology							
CreateTopology							
CopyTopology							
ST_InitTopoGeo					✓		
ST_CreateTopoGeo	✓				✓		
TopoGeo_AddPoint	✓						
TopoGeo_AddLine g	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
TopoGeo_AddPolyg	✓						
TopoGeo_LoadGeo	✓ y						
ST_AddIsoNode	✓				✓		
ST_AddIsoEdge	✓				✓		
ST_AddEdgeNewFa	✓				✓		
ST_AddEdgeModFa	✓				✓		
ST_RemEdgeNewFace					✓		
ST_RemEdgeModFace					✓		
ST_ChangeEdgeGe	✓				✓		
ST_ModEdgeSplit	✓				✓		
ST_ModEdgeHeal					✓		
ST_NewEdgeHeal					✓		
ST_MoveIsoNode	✓				✓		
ST_NewEdgesSplit	✓				✓		
ST_RemoveIsoNode					✓		
ST_RemoveIsoEdge					✓		
GetEdgeByPoint	✓						
GetFaceByPoint	✓						
GetFaceContainingI	✓ t						
GetNodeByPoint	✓						
GetTopologyID							
GetTopologySRID							
GetTopologyName							
ST_GetFaceEdges					✓		
ST_GetFaceGeomet	✓				✓		
GetRingEdges							
GetNodeEdges							
FindVertexSegmentPairsBelowDistance							
Polygonize							
AddNode	✓						
AddEdge	✓						
AddFace	✓						
ST_Simplify	✓						
RemoveUnusedPrim	✓ es						
CreateTopoGeom	✓						
toTopoGeom	✓						
TopoElementArray_Agg							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
TopoElement	✓						
clearTopoGeom	✓						
TopoGeom_addEle	✓						
TopoGeom_remEle	✓ t						
TopoGeom_addTop	✓ om						
toTopoGeom							
GetTopoGeomElementArray							
GetTopoGeomElements							
ST_SRID	✓						
AsGML	✓						
AsTopoJSON	✓						
Equals	✓		✓				
Intersects	✓		✓				
geomval							
addbandarg							
rastbandarg							
raster							
reclassarg							
summarystats							
unionarg							
AddRasterConstraints							
DropRasterConstraints							
AddOverviewConstraints							
DropOverviewConstraints							
PostGIS_GDAL_Version							
PostGIS_Raster_Lib_Build_Date							
PostGIS_Raster_Lib_Version							
ST_GDALDrivers							
UpdateRasterSRID							
ST_CreateOverview							
ST_AddBand							
ST_AsRaster	✓						
ST_AsRasterAgg	✓						
ST_Band							
ST_MakeEmptyCoverage							
ST_MakeEmptyRaster							
ST_Tile							
ST_Retile	✓						
ST_FromGDALRaster							
ST_GeoReference							
ST_Height							
ST_IsEmpty							
ST_MemSize							
ST_MetaData							
ST_NumBands							
ST_PixelHeight							
ST_PixelWidth							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_ScaleX							
ST_ScaleY							
ST_RasterToWorldCoord							
ST_RasterToWorldCoordX							
ST_RasterToWorldCoordY							
ST_Rotation							
ST_SkewX							
ST_SkewY							
ST_SRID							
ST_Summary							
ST_UpperLeftX							
ST_UpperLeftY							
ST_Width							
ST_WorldToRasterCoord	✓						
ST_WorldToRasterCoordX	✓						
ST_WorldToRasterCoordY	✓						
ST_BandMetaData							
ST_BandNoDataValue							
ST_BandIsNoData							
ST_BandPath							
ST_BandFileSize							
ST_BandFileTimestamp							
ST_BandPixelType							
ST_MinPossibleValue							
ST_HasNoBand							
ST_PixelAsPolygon	✓						
ST_PixelAsPolygons							
ST_PixelAsPoint	✓						
ST_PixelAsPoints							
ST_PixelAsCentroid	✓						
ST_PixelAsCentroids							
ST_Value	✓						
ST_NearestValue	✓						
ST_SetZ	✓						
ST_SetM	✓						
ST_Neighborhood	✓						
ST_SetValue	✓						
ST_SetValues							
ST_DumpValues							
ST_PixelOfValue							
ST_SetGeoReference							
ST_SetRotation							
ST_SetScale							
ST_SetSkew							
ST_SetSRID							
ST_SetUpperLeft							
ST_Resample							
ST_Rescale							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Reskew							
ST_SnapToGrid							
ST_Resize							
ST_Transform							
ST_SetBandNoDataValue							
ST_SetBandIsNoData							
ST_SetBandPath							
ST_SetBandIndex							
ST_Count							
ST_CountAgg							
ST_Histogram							
ST_Quantile							
ST_SummaryStats							
ST_SummaryStatsAgg							
ST_ValueCount							
ST_RastFromWKB							
ST_RastFromHexWKB							
ST_AsBinary/ST_AsWKB							
ST_AsHexWKB							
ST_AsGDALRaster							
ST_AsJPEG							
ST_AsPNG							
ST_AsTIFF							
ST_Clip	✓						
ST_ColorMap							
ST_Grayscale							
ST_Intersection	✓						
ST_MapAlgebra (callback function version)							
ST_MapAlgebra (expression version)							
ST_MapAlgebraExpr							
ST_MapAlgebraExpr							
ST_MapAlgebraFct							
ST_MapAlgebraFct							
ST_MapAlgebraFctNgb							
ST_Reclass							
ST_ReclassExact							
ST_Union							
ST_Distinct4ma							
ST_InvDistWeight4ma							
ST_Max4ma							
ST_Mean4ma							
ST_Min4ma							
ST_MinDist4ma							
ST_Range4ma							
ST_StdDev4ma							
ST_Sum4ma							
ST_Aspect							
ST_HillShade							
ST_Roughness							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Slope							
ST_TPI							
ST_TRI							
ST_InterpolateRaste	✓						
ST_Contour							
Box3D	☑						
ST_ConvexHull	✓						
ST_DumpAsPolygons							
ST_Envelope	✓						
ST_MinConvexHull	✓						
ST_Polygon	✓						
ST_IntersectionFrac	✓ s						
&&	✓						
&<							
&>							
=							
@	✓						
~=							
~	✓						
ST_Contains							
ST_ContainsProperly							
ST_Covers							
ST_CoveredBy							
ST_Disjoint							
ST_Intersects	✓						
ST_Overlaps							
ST_Touches							
ST_SameAlignment							
ST_NotSameAlignmentReason							
ST_Within							
ST_DWithin							
ST_DFullyWithin							

13.12 New, Enhanced or changed PostGIS Functions

13.12.1 PostGIS Functions new or enhanced in 3.7

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.7

- CG_ApproximateMedialAxis - Availability: 3.7.0 - projected parameter. Requires SFCGAL >= 2.3.0 for projected result; falls back to non-projected with a notice on older versions. Compute the approximate medial axis of an areal geometry.
- CG_GenerateFlatRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a flat (box) roof from a footprint polygon.
- CG_GenerateGableRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a gable roof from a footprint polygon.

- CG_GenerateHippedRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a hipped roof from a footprint polygon.
- CG_GenerateRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a roof of the requested type from a footprint polygon.
- CG_GenerateSkillionRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a skillion (single-slope) roof from a footprint polygon.
- CG_NurbsCurveApproximate - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Creates an approximating NURBS curve fitting data points within a tolerance
- CG_NurbsCurveDerivative - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Computes the derivative (tangent, curvature) of a NURBS curve at a given parameter
- CG_NurbsCurveInterpolate - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Creates an interpolating NURBS curve passing through all given data points
- CG_PolygonRepair - Availability: 3.7.0 - requires SFCGAL >= 2.3.0 and CGAL >= 6.0. Repair an invalid polygon or multi-polygon.
- FindVertexSegmentPairsBelowDistance - Availability: 3.7 Find pairs of topology vertex/segment that are closer than tolerated distance
- ST_3DInterpolatePoint - Availability: 3.7.0 Returns the interpolated measure of a geometry closest to a point in 3D.
- ST_CatmullRomSmoothing - Availability: 3.7.0 Returns a smoothed version of a geometry, using the Catmull-Rom spline algorithm
- ST_ClusterRelateWin - Availability: 3.7.0 Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.
- ST_ControlPoints - Availability: 3.7.0 Returns the control points of a NURBS curve as a MULTIPOINT geometry.
- ST_CoverageEdges - Availability: 3.7.0 Computes the unique edges of a polygonal coverage.
- ST_Degree - Availability: 3.7.0 Returns the polynomial degree of a NURBS curve.
- ST_Evaluate - Availability: 3.7.0 Evaluates a NURBS curve at a specific parameter value and returns the resulting point.
- ST_IsNurbsCurve - Availability: 3.7.0 Checks if a geometry is a NURBS curve.
- ST_Knots - Availability: 3.7.0 Returns the knot vector of a NURBS curve.
- ST_MSize - Availability: 3.7.0 Returns the M size of a geometry.
- ST_MakeLine - Availability: 3.7.0 - Support for MultiLineString input elements was introduced Creates a LineString from Point, MultiPoint, or LineString geometries.
- ST_MakeNurbsCurve - Availability: 3.7.0 Creates a NURBS (Non-Uniform Rational B-Spline) curve from control points, optional weights, and optional knot vector.
- ST_MinimumSpanningTree - Availability: 3.7.0 Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.
- ST_NumControlPoints - Availability: 3.7.0 Returns the number of control points in a NURBS curve.
- ST_NurbsCurveIsRational - Availability: 3.7.0 Checks if a NURBS curve is rational (has weights).
- ST_NurbsToLineString - Availability: 3.7.0 Converts a NURBS curve to a LINESTRING by uniform sampling.
- ST_Weights - Availability: 3.7.0 Returns the weight array of a rational NURBS curve.
- ST_XSize - Availability: 3.7.0 Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_YSize - Availability: 3.7.0 Returns the Y size of a 2D or 3D bounding box or a geometry.

- ST_ZSize - Availability: 3.7.0 Returns the Z size of a 2D or 3D bounding box or a geometry.

Functions enhanced in PostGIS 3.7

- ST_AsX3D - Enhanced: 3.7.0: Support for curve geometries. NURBSCURVE is emitted as X3D NurbsCurve; other curve types are linearized before output. Returns a Geometry in X3D xml node element format: ISO-IEC-19776-1.2-X3DEncodings-XML
- ST_GMLToSQL - Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced. Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GeomFromGML - Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced. Takes as input GML representation of geometry and outputs a PostGIS geometry object
- ST_LongestLine - Enhanced: 3.7.0 - support for a single geometry input. Returns the 2D longest line between two geometries.
- ST_MakePolygon - Enhanced: 3.7.0 - Support for curved input rings was introduced. Creates a Polygon or CurvePolygon from a shell and optional list of holes.
- ST_MaxDistance - Enhanced: 3.7.0 - support for a single geometry input. Returns the 2D largest distance between two geometries in projected units.
- ST_Value - Enhanced: 3.7.0 resample accepts nearest-neighbor boundary options "nearest-UL", "nearest-UR", "nearest-LL", and "nearest-LR". Returns the value of a given band in a given columnx, rowy pixel or at a particular geometric point. Band numbers start at 1 and assumed to be 1 if not specified. If exclude_nodata_value is set to false, then all pixels include nodata pixels are considered to intersect and return value. If exclude_nodata_value is not passed in then reads it from metadata of raster.
- TopoGeo_AddLineString - Enhanced: 3.7.0 added support for limiting the number of new edges created in the topology. Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.

Functions changed in PostGIS 3.7

- ST_AsGeoJSON - Changed: 3.7.0 added warning about duplicate keys Return a geometry or feature in GeoJSON format.
- toTopoGeom - Changed: 3.7.0 changes meaning of tolerance 0 to really mean 0 and -1 (new default) to request using the topology precision. Converts a simple Geometry into a topo geometry.

13.12.2 PostGIS Functions new or enhanced in 3.6

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.6

- CG_2DRotate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry by a given angle around a specified point in 2D.
- CG_3DAlphaWrapping - Availability: 3.6.0 - requires SFCGAL >= 2.1.0 Computes a 3D Alpha-wrapping strictly enclosing a geometry.
- CG_3DBuffer - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Computes a 3D buffer around a geometry.
- CG_3DRotate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry in 3D space around an axis vector.
- CG_3DScale - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Scales a geometry by separate factors along X, Y, and Z axes.
- CG_3DScaleAroundCenter - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Scales a geometry in 3D space around a specified center point.
- CG_3DTranslate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Translates (moves) a geometry by given offsets in 3D space.

- CG_Rotate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry by a given angle around the origin (0,0).
- CG_RotateX - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry around the X-axis by a given angle.
- CG_RotateY - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry around the Y-axis by a given angle.
- CG_RotateZ - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry around the Z-axis by a given angle.
- CG_Scale - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Scales a geometry uniformly in all dimensions by a given factor.
- CG_Simplify - Availability: 3.6.0 - requires SFCGAL >= 2.1.0 Reduces the complexity of a geometry while preserving essential features and Z/M values.
- CG_StraightSkeletonPartition - Availability: 3.6.0 - requires SFCGAL >= 2.0.0. Computes the straight skeleton partition of a polygon.
- CG_Translate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Translates (moves) a geometry by given offsets in 2D space.
- FixCorruptTopoGeometryColumn - Availability: 3.6.1 Fixes topogeometry corruption caused by upgrade to postgis_topology 3.6.0 and higher
- MakeTopologyPrecise - Availability: 3.6.0 Snap topology vertices to precision grid.
- ST_AsRasterAgg - Availability: 3.6.0 Aggregate. Renders PostGIS geometries into a new raster.
- ST_CoverageClean - Availability: 3.6.0 - requires GEOS >= 3.14.0 Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.
- ST_IntersectionFractions - Availability: 3.6.0 Requires GEOS 3.14 or higher. Calculates the fraction of each raster cell that is covered by a given geometry.
- ST_ReclassExact - Availability: 3.6.0 Creates a new raster composed of bands reclassified from original, using a 1:1 mapping from values in the original band to new values in the destination band.
- TotalTopologySize - Availability: 3.6.0 Total disk space used by the specified topology, including all indexes and TOAST data.
- UpgradeTopology - Availability: 3.6.0 Upgrades the specified topology to support large ids (int8) for topology and primitive ids.
- ValidateTopologyPrecision - Availability: 3.6.0 Returns non-precise vertices in the topology.
- postgis.gdal_cpl_debug - Availability: 3.6.0 A boolean configuration to turn logging of GDAL debug messages on and off.

Functions enhanced in PostGIS 3.6

- CreateTopology - Enhanced: 3.6.0 added topoid and usesLargeIDs parameters. Creates a new topology schema and registers it in the topology.topology table.

Functions changed in PostGIS 3.6

- ST_GeometryN - Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and are not decomposed into patches; use ST_PatchN to access faces. Return an element of a geometry collection.
- ST_NumGeometries - Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and return 1; use ST_NumPatches to count faces. Returns the number of elements in a geometry collection.

13.12.3 PostGIS Functions new or enhanced in 3.5

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.5

- CG_3DArea - Availability: 3.5.0 Computes area of 3D surface geometries. Will return 0 for solids.
- CG_3DConvexHull - Availability: 3.5.0 Computes the 3D convex hull of a geometry.
- CG_3DDifference - Availability: 3.5.0 Perform 3D difference
- CG_3DDistance - Availability: 3.5.0 Computes the minimum 3D distance between two geometries
- CG_3DIntersection - Availability: 3.5.0 Perform 3D intersection
- CG_3DIntersects - Availability: 3.5.0 Tests if two 3D geometries intersect
- CG_3DUnion - Availability: 3.5.0 Perform 3D union using postgis_sfcgal.
- CG_AlphaShape - Availability: 3.5.0 - requires SFCGAL >= 1.4.1. Computes an Alpha-shape enclosing a geometry
- CG_ApproxConvexPartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes approximal convex partition of the polygon geometry
- CG_ApproximateMedialAxis - Availability: 3.5.0 Compute the approximate medial axis of an areal geometry.
- CG_Area - Availability: 3.5.0 Calculates the area of a geometry
- CG_Difference - Availability: 3.5.0 Computes the geometric difference between two geometries
- CG_Distance - Availability: 3.5.0 Computes the minimum distance between two geometries
- CG_Extrude - Availability: 3.5.0 Extrude a surface to a related volume
- CG_ExtrudeStraightSkeleton - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Straight Skeleton Extrusion
- CG_ForceLHR - Availability: 3.5.0 Force LHR orientation
- CG_GreeneApproxConvexPartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes approximal convex partition of the polygon geometry
- CG_Intersection - Availability: 3.5.0 Computes the intersection of two geometries
- CG_Intersects - Availability: 3.5.0 Tests if two geometries intersect (they have at least one point in common)
- CG_IsPlanar - Availability: 3.5.0 Check if a surface is or not planar
- CG_IsSolid - Availability: 3.5.0 Test if the geometry is a solid. No validity check is performed.
- CG_MakeSolid - Availability: 3.5.0 Cast the geometry into a solid. No check is performed. To obtain a valid solid, the input geometry must be a closed Polyhedral Surface or a closed TIN.
- CG_MinkowskiSum - Availability: 3.5.0 Performs Minkowski sum
- CG_OptimalAlphaShape - Availability: 3.5.0 - requires SFCGAL >= 1.4.1. Computes an Alpha-shape enclosing a geometry using an "optimal" alpha value.
- CG_OptimalConvexPartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes an optimal convex partition of the polygon geometry
- CG_Orientation - Availability: 3.5.0 Determine surface orientation
- CG_StraightSkeleton - Availability: 3.5.0 Compute a straight skeleton from a geometry
- CG_Tesselate - Availability: 3.5.0 Perform surface Tessellation of a polygon or polyhedralsurface and returns as a TIN or collection of TINS

- CG_Triangulate - Availability: 3.5.0 Triangulates a polygonal geometry
- CG_Union - Availability: 3.5.0 Computes the union of two geometries
- CG_Visibility - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Compute a visibility polygon from a point or a segment in a polygon geometry
- CG_Volume - Availability: 3.5.0 Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- CG_YMonotonePartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes y-monotone partition of the polygon geometry
- ST_HasM - Availability: 3.5.0 Checks if a geometry has an M (measure) dimension.
- ST_HasZ - Availability: 3.5.0 Checks if a geometry has a Z dimension.
- ST_RemoveIrrelevantPointsForView - Availability: 3.5.0 Removes points that are irrelevant for rendering a specific rectangular view of a geometry.
- ST_RemoveSmallParts - Availability: 3.5.0 Removes small parts (polygon rings or linestrings) of a geometry.
- TopoGeo_LoadGeometry - Availability: 3.5.0 Load a geometry into an existing topology, snapping and splitting as needed.

Functions enhanced in PostGIS 3.5

- ST_Clip - Enhanced: 3.5.0 - touched argument added. Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.

Functions changed in PostGIS 3.5

- ST_AsGeoJSON - Changed: 3.5.0 allow specifying the column containing the feature id Return a geometry or feature in GeoJSON format.
- ST_DFullyWithin - Changed: 3.5.0 : the logic behind the function now uses a test of containment within a buffer, rather than the ST_MaxDistance algorithm. Results will differ from prior versions, but should be closer to user expectations. Tests if a geometry is entirely inside a distance of another

13.12.4 PostGIS Functions new or enhanced in 3.4

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.4

- PostGIS_GEOS_Compiled_Version - Availability: 3.4.0 Returns the version number of the GEOS library against which PostGIS was built.
- PostGIS_PROJ_Compiled_Version - Availability: 3.5.0 Returns the version number of the PROJ library against which PostGIS was built.
- RenameTopoGeometryColumn - Availability: 3.4.0 Renames a topogeometry column
- RenameTopology - Availability: 3.4.0 Renames a topology
- ST_ClusterIntersectingWin - Availability: 3.4.0 Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.
- ST_ClusterWithinWin - Availability: 3.4.0 Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_CoverageInvalidEdges - Availability: 3.4.0 Window function that finds locations where polygons fail to form a valid coverage.

- ST_CoverageSimplify - Availability: 3.4.0 Window function that simplifies the edges of a polygonal coverage.
- ST_CoverageUnion - Availability: 3.4.0 - requires GEOS >= 3.8.0 Computes the union of a set of polygons forming a coverage by removing shared edges.
- ST_InverseTransformPipeline - Availability: 3.4.0 Return a new geometry with coordinates transformed to a different spatial reference system using the inverse of a defined coordinate transformation pipeline.
- ST_LargestEmptyCircle - Availability: 3.4.0. Computes the largest circle not overlapping a geometry.
- ST_LineExtend - Availability: 3.4.0 Returns a line extended forwards and backwards by specified distances.
- ST_TransformPipeline - Availability: 3.4.0 Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline.
- TopoElement - Availability: 3.4.0 Converts a topogeometry to a topoelement.
- postgis_srs - Availability: 3.4.0 Return a metadata record for the requested authority and srid.
- postgis_srs_all - Availability: 3.4.0 Return metadata records for every spatial reference system in the underlying Proj database.
- postgis_srs_codes - Availability: 3.4.0 Return the list of SRS codes associated with the given authority.
- postgis_srs_search - Availability: 3.4.0 Return metadata records for projected coordinate systems that have areas of usage that fully contain the bounds parameter.

Functions enhanced in PostGIS 3.4

- PostGIS_Full_Version - Enhanced: 3.4.0 now includes extra PROJ configurations NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location Reports full PostGIS version and build configuration infos.
- PostGIS_PROJ_Version - Enhanced: 3.4.0 now includes NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location Returns the version number of the PROJ4 library.
- ST_AsSVG - Enhanced: 3.4.0 to support all curve types Returns SVG path data for a geometry.
- ST_ClosestPoint - Enhanced: 3.4.0 - Support for geography. Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.
- ST_LineSubstring - Enhanced: 3.4.0 - Support for geography was introduced. Returns the part of a line between two fractional locations.
- ST_Project - Enhanced: 3.4.0 Allow geometry arguments and two-point form omitting azimuth. Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Resample - Enhanced: 3.4.0 max and min resampling options added Resample a raster using a specified resampling algorithm, new dimensions, an arbitrary grid corner and a set of raster georeferencing attributes defined or borrowed from another raster.
- ST_Rescale - Enhanced: 3.4.0 max and min resampling options added Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_ShortestLine - Enhanced: 3.4.0 - support for geography. Returns the 2D shortest line between two geometries

Functions changed in PostGIS 3.4

- PostGIS_Extensions_Upgrade - Changed: 3.4.0 to add target_version argument. Packages and upgrades PostGIS extensions (e.g. postgis_raster,postgis_topology, postgis_sfcgal) to given or latest version.

13.12.5 PostGIS Functions new or enhanced in 3.3

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.3

- RemoveUnusedPrimitives - Availability: 3.3.0 Removes topology primitives which not needed to define existing TopoGeometry objects.
- ST_AsMARC21 - Availability: 3.3.0 Returns geometry as a MARC21/XML record with a geographic datafield (034).
- ST_GeomFromMARC21 - Availability: 3.3.0, requires libxml2 2.6+ Takes MARC21/XML geographic data as input and returns a PostGIS geometry object.
- ST_Letters - Availability: 3.3.0 Returns the input letters rendered as geometry with a default start position at the origin and default text height of 100.
- ST_SimplifyPolygonHull - Availability: 3.3.0. Computes a simplified topology-preserving outer or inner hull of a polygonal geometry.
- ST_TriangulatePolygon - Availability: 3.3.0. Computes the constrained Delaunay triangulation of polygons
- postgis_sfcgal_full_version - Availability: 3.3.0 Returns the full version of SFCGAL in use including CGAL and Boost versions

Functions enhanced in PostGIS 3.3

- ST_ConcaveHull - Enhanced: 3.3.0, GEOS native implementation enabled for GEOS 3.11+ Computes a possibly concave geometry that contains all input geometry vertices
- ST_LineMerge - Enhanced: 3.3.0 accept a directed parameter. Return the lines formed by sewing together a MultiLineString.

Functions changed in PostGIS 3.3

- PostGIS_Extensions_Upgrade - Changed: 3.3.0 support for upgrades from any PostGIS version. Does not work on all systems. Packages and upgrades PostGIS extensions (e.g. postgis_raster,postgis_topology, postgis_sfcgal) to given or latest version.

13.12.6 PostGIS Functions new or enhanced in 3.2

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.2

- FindLayer - Availability: 3.2.0 Returns a topology.layer record by different means.
- FindTopology - Availability: 3.2.0 Returns a topology record by different means.
- GetFaceContainingPoint - Availability: 3.2.0 Finds the face containing a point.
- ST_AsFlatGeobuf - Availability: 3.2.0 Return a FlatGeobuf representation of a set of rows.
- ST_Contour - Availability: 3.2.0 Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.
- ST_DumpSegments - Availability: 3.2.0 Returns a set of geometry_dump rows for the segments in a geometry.
- ST_FromFlatGeobuf - Availability: 3.2.0 Reads FlatGeobuf data.
- ST_FromFlatGeobufToTable - Availability: 3.2.0 Creates a table based on the structure of FlatGeobuf data.
- ST_InterpolateRaster - Availability: 3.2.0 Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation.

- ST_SRID - Availability: 3.2.0 Returns the spatial reference identifier for a topogeometry.
- ST_Scroll - Availability: 3.2.0 Change start point of a closed LineString.
- ST_SetM - Availability: 3.2.0 Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimension using the requested resample algorithm.
- ST_SetZ - Availability: 3.2.0 Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimension using the requested resample algorithm.
- TopoGeom_addTopoGeom - Availability: 3.2 Adds element of a TopoGeometry to the definition of another TopoGeometry.
- ValidateTopologyRelation - Availability: 3.2.0 Returns info about invalid topology relation records
- postgis.gdal_vsi_options - Availability: 3.2.0 A string configuration to set options used when working with an out-db raster.

Functions enhanced in PostGIS 3.2

- GetFaceByPoint - Enhanced: 3.2.0 more efficient implementation and clearer contract, stops working with invalid topologies. Finds face intersecting a given point.
- ST_ClusterKMeans - Enhanced: 3.2.0 Support for max_radius Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_MakeValid - Enhanced: 3.2.0, added algorithm options, 'linework' and 'structure' which requires GEOS >= 3.10.0. Attempts to make an invalid geometry valid without losing vertices.
- ST_PixelAsCentroid - Enhanced: 3.2.0 Faster now implemented in C. Returns the centroid (point geometry) of the area represented by a pixel.
- ST_PixelAsCentroids - Enhanced: 3.2.0 Faster now implemented in C. Returns the centroid (point geometry) for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The point geometry is the centroid of the area represented by a pixel.
- ST_Point - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y and SRID values.
- ST_PointM - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y, M and SRID values.
- ST_PointZ - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y, Z and SRID values.
- ST_PointZM - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y, Z, M and SRID values.
- ST_RemovePoint - Enhanced: 3.2.0 Remove a point from a linestring.
- ST_RemoveRepeatedPoints - Enhanced: 3.2.0 Returns a version of a geometry with duplicate points removed.
- ST_StartPoint - Enhanced: 3.2.0 returns a point for all geometries. Prior behavior returns NULLs if input was not a LineString. Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_Value - Enhanced: 3.2.0 resample optional argument was added. Returns the value of a given band in a given columnx, rowy pixel or at a particular geometric point. Band numbers start at 1 and assumed to be 1 if not specified. If exclude_nodata_value is set to false, then all pixels include nodata pixels are considered to intersect and return value. If exclude_nodata_value is not passed in then reads it from metadata of raster.
- TopoGeo_AddLineString - Enhanced: 3.2.0 added support for returning signed identifier. Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.

Functions changed in PostGIS 3.2

- ST_Boundary - Changed: 3.2.0 support for TIN, does not use geos, does not linearize curves Returns the boundary of a geometry.
- ValidateTopology - Changed: 3.2.0 added optional bbox parameter, perform face labeling and edge linking checks. Returns a set of validatetopology_returntype objects detailing issues with topology.

13.12.7 PostGIS Functions new or enhanced in 3.1

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.1

- ST_Hexagon - Availability: 3.1.0 Returns a single hexagon, using the provided edge size and cell coordinate within the hexagon grid space.
- ST_HexagonGrid - Availability: 3.1.0 Returns a set of hexagons and cell indices that completely cover the bounds of the geometry argument.
- ST_MaximumInscribedCircle - Availability: 3.1.0. Computes the largest circle contained within a geometry.
- ST_ReducePrecision - Availability: 3.1.0. Returns a valid geometry with points rounded to a grid tolerance.
- ST_Square - Availability: 3.1.0 Returns a single square, using the provided edge size and cell coordinate within the square grid space.
- ST_SquareGrid - Availability: 3.1.0 Returns a set of grid squares and cell indices that completely cover the bounds of the geometry argument.

Functions enhanced in PostGIS 3.1

- ST_AsEWKT - Enhanced: 3.1.0 support for optional precision parameter. Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.
- ST_ClusterKMeans - Enhanced: 3.1.0 Support for 3D geometries and weights Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_Difference - Enhanced: 3.1.0 accept a gridSize parameter. Computes a geometry representing the part of geometry A that does not intersect geometry B.
- ST_Intersection - Enhanced: 3.1.0 accept a gridSize parameter Computes a geometry representing the shared portion of geometries A and B.
- ST_MakeValid - Enhanced: 3.1.0, added removal of Coordinates with NaN values. Attempts to make an invalid geometry valid without losing vertices.
- ST_Subdivide - Enhanced: 3.1.0 accept a gridSize parameter. Computes a rectilinear subdivision of a geometry.
- ST_SymDifference - Enhanced: 3.1.0 accept a gridSize parameter. Computes a geometry representing the portions of geometries A and B that do not intersect.
- ST_TileEnvelope - Enhanced: 3.1.0 Added margin parameter. Creates a rectangular Polygon in Web Mercator (SRID:3857) using the XYZ tile system.
- ST_UnaryUnion - Enhanced: 3.1.0 accept a gridSize parameter. Computes the union of the components of a single geometry.
- ST_Union - Enhanced: 3.1.0 accept a gridSize parameter. Computes a geometry representing the point-set union of the input geometries.

Functions changed in PostGIS 3.1

- ST_Count - Changed: 3.1.0 - The ST_Count(rastertable, rastercolumn, ...) variants removed. Use instead. Returns the number of pixels in a given band of a raster or raster coverage. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the nodata value.
- ST_Force3D - Changed: 3.1.0. Added support for supplying a non-zero Z value. Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.
- ST_Force3DM - Changed: 3.1.0. Added support for supplying a non-zero M value. Force the geometries into XYM mode.

- ST_Force3DZ - Changed: 3.1.0. Added support for supplying a non-zero Z value. Force the geometries into XYZ mode.
- ST_Force4D - Changed: 3.1.0. Added support for supplying non-zero Z and M values. Force the geometries into XYZM mode.
- ST_Histogram - Changed: 3.1.0 Removed ST_Histogram(table_name, column_name) variant. Returns a set of record summarizing a raster or raster coverage data distribution separate bin ranges. Number of bins are autocomputed if not specified.
- ST_Quantile - Changed: 3.1.0 Removed ST_Quantile(table_name, column_name) variant. Compute quantiles for a raster or raster table coverage in the context of the sample or population. Thus, a value could be examined to be at the raster's 25%, 50%, 75% percentile.
- ST_SummaryStats - Changed: 3.1.0 ST_SummaryStats(rastertable, rastercolumn, ...) variants are removed. Use instead. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.

13.12.8 PostGIS Functions new or enhanced in 3.0

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.0

- CG_ConstrainedDelaunayTriangles - Availability: 3.0.0 Return a constrained Delaunay triangulation around the given input geometry.
- ST_3DLineInterpolatePoint - Availability: 3.0.0 Returns a point interpolated along a 3D line at a fractional location.
- ST_TileEnvelope - Availability: 3.0.0 Creates a rectangular Polygon in Web Mercator (SRID:3857) using the XYZ tile system.

Functions enhanced in PostGIS 3.0

- ST_AsMVT - Enhanced: 3.0 - added support for Feature ID. Aggregate function returning a MVT representation of a set of rows.
- ST_Contains - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of B lies in A, and their interiors have a point in common
- ST_ContainsProperly - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of B lies in the interior of A
- ST_CoveredBy - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of A lies in B
- ST_Covers - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of B lies in A
- ST_Crosses - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have some, but not all, interior points in common
- ST_CurveToLine - Enhanced: 3.0.0 implemented a minimum number of segments per linearized arc to prevent topological collapse. Converts a geometry containing curves to a linear geometry.
- ST_Disjoint - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have no points in common
- ST_Equals - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries include the same set of points
- ST_GeneratePoints - Enhanced: 3.0.0, added seed parameter Generates a multipoint of random points contained in a Polygon or MultiPolygon.
- ST_GeomFromGeoJSON - Enhanced: 3.0.0 parsed geometry defaults to SRID=4326 if not specified otherwise. Takes as input a geojson representation of a geometry and outputs a PostGIS geometry object

- ST_LocateBetween - Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE. Returns the portions of a geometry that match a measure range.
- ST_LocateBetweenElevations - Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE. Returns the portions of a geometry that lie in an elevation (Z) range.
- ST_Overlaps - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have the same dimension and intersect, but each has at least one point not in the other
- ST_Relate - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix
- ST_Segmentize - Enhanced: 3.0.0 Segmentize geometry now produces equal-length subsegments Returns a modified geometry/geography having no segment longer than a given distance.
- ST_Touches - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have at least one point in common, but their interiors do not intersect
- ST_Within - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of A lies in B, and their interiors have a point in common

Functions changed in PostGIS 3.0

- PostGIS_Extensions_Upgrade - Changed: 3.0.0 to repackage loose extensions and support postgis_raster. Packages and upgrades PostGIS extensions (e.g. postgis_raster,postgis_topology, postgis_sfcgal) to given or latest version.
- ST_3DDistance - Changed: 3.0.0 - SFCGAL version removed Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.
- ST_3DIntersects - Changed: 3.0.0 SFCGAL backend removed, GEOS backend supports TINs. Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_Area - Changed: 3.0.0 - does not depend on SFCGAL anymore. Returns the area of a polygonal geometry.
- ST_AsGeoJSON - Changed: 3.0.0 support records as input Return a geometry or feature in GeoJSON format.
- ST_AsGeoJSON - Changed: 3.0.0 output SRID if not EPSG:4326. Return a geometry or feature in GeoJSON format.
- ST_AsKML - Changed: 3.0.0 - Removed the "versioned" variant signature Return the geometry as a KML element.
- ST_Distance - Changed: 3.0.0 - does not depend on SFCGAL anymore. Returns the distance between two geometry or geography values.
- ST_Intersection - Changed: 3.0.0 does not depend on SFCGAL. Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Changed: 3.0.0 SFCGAL version removed and native support for 2D TINS added. Tests if two geometries intersect (they have at least one point in common)
- ST_Union - Changed: 3.0.0 does not depend on SFCGAL. Computes a geometry representing the point-set union of the input geometries.

13.12.9 PostGIS Functions new or enhanced in 2.5

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.5

- PostGIS_Extensions_Upgrade - Availability: 2.5.0 Packages and upgrades PostGIS extensions (e.g. postgis_raster,postgis_topology, postgis_sfcgal) to given or latest version.
- ST_Angle - Availability: 2.5.0 Returns the angle between two vectors defined by 3 or 4 points, or 2 lines.

- ST_AsHexWKB - Availability: 2.5.0 Return the Well-Known Binary (WKB) in Hex representation of the raster.
- ST_BandFileSize - Availability: 2.5.0 Returns the file size of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandFileTimestamp - Availability: 2.5.0 Returns the file timestamp of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_ChaikinSmoothing - Availability: 2.5.0 Returns a smoothed version of a geometry, using the Chaikin algorithm
- ST_FilterByM - Availability: 2.5.0 Removes vertices based on their M value
- ST_Grayscale - Availability: 2.5.0 Creates a new one-8BUI band raster from the source raster and specified bands representing Red, Green and Blue
- ST_LineInterpolatePoints - Availability: 2.5.0 Returns points interpolated along a line at a fractional interval.
- ST_OrientedEnvelope - Availability: 2.5.0. Returns a minimum-area rectangle containing a geometry.
- ST_QuantizeCoordinates - Availability: 2.5.0 Sets least significant bits of coordinates to zero
- ST_RastFromHexWKB - Availability: 2.5.0 Return a raster value from a Hex representation of Well-Known Binary (WKB) raster.
- ST_RastFromWKB - Availability: 2.5.0 Return a raster value from a Well-Known Binary (WKB) raster.
- ST_SetBandIndex - Availability: 2.5.0 Update the external band number of an out-db band
- ST_SetBandPath - Availability: 2.5.0 Update the external path and band number of an out-db band

Functions enhanced in PostGIS 2.5

- ST_AsBinary/ST_AsWKB - Enhanced: 2.5.0 Addition of ST_AsWKB Return the Well-Known Binary (WKB) representation of the raster.
- ST_AsMVT - Enhanced: 2.5.0 - added support parallel query. Aggregate function returning a MVT representation of a set of rows.
- ST_AsText - Enhanced: 2.5 - optional parameter precision introduced. Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.
- ST_BandMetaData - Enhanced: 2.5.0 to include outdbbandnum, filesize and filetimestamp for outdb rasters. Returns basic meta data for a specific raster band. band num 1 is assumed if none-specified.
- ST_Buffer - Enhanced: 2.5.0 - ST_Buffer geometry support was enhanced to allow for side buffering specification side=both|left|right. Computes a geometry covering all points within a given distance from a geometry.
- ST_GeomFromGeoJSON - Enhanced: 2.5.0 can now accept json and jsonb as inputs. Takes as input a geojson representation of a geometry and outputs a PostGIS geometry object
- ST_GeometricMedian - Enhanced: 2.5.0 Added support for M as weight of points. Returns the geometric median of a Multi-Point.
- ST_Intersects - Enhanced: 2.5.0 Supports GEOMETRYCOLLECTION. Tests if two geometries intersect (they have at least one point in common)
- ST_OffsetCurve - Enhanced: 2.5 - added support for GEOMETRYCOLLECTION and MULTILINESTRING Returns an offset line at a given distance and side from an input line.
- ST_Scale - Enhanced: 2.5.0 support for scaling relative to a local origin (origin parameter) was introduced. Scales a geometry by given factors.
- ST_Split - Enhanced: 2.5.0 support for splitting a polygon by a multiline was introduced. Returns a collection of geometries created by splitting a geometry by another geometry.

- ST_Subdivide - Enhanced: 2.5.0 reuses existing points on polygon split, vertex count is lowered from 8 to 5. Computes a rectilinear subdivision of a geometry.

Functions changed in PostGIS 2.5

- ST_GDALDrivers - Changed: 2.5.0 - add can_read and can_write columns. Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster

13.12.10 PostGIS Functions new or enhanced in 2.4

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.4

- ST_AsGeobuf - Availability: 2.4.0 Return a Geobuf representation of a set of rows.
- ST_AsMVT - Availability: 2.4.0 Aggregate function returning a MVT representation of a set of rows.
- ST_AsMVTGeom - Availability: 2.4.0 Transforms a geometry into the coordinate space of a MVT tile.
- ST_Centroid - Availability: 2.4.0 support for geography was introduced. Returns the geometric center of a geometry.
- ST_ForcePolygonCCW - Availability: 2.4.0 Orients all exterior rings counter-clockwise and all interior rings clockwise.
- ST_ForcePolygonCW - Availability: 2.4.0 Orients all exterior rings clockwise and all interior rings counter-clockwise.
- ST_FrechetDistance - Availability: 2.4.0 - requires GEOS >= 3.7.0 Returns the Fréchet distance between two geometries.
- ST_IsPolygonCCW - Availability: 2.4.0 Tests if Polygons have exterior rings oriented counter-clockwise and interior rings oriented clockwise.
- ST_IsPolygonCW - Availability: 2.4.0 Tests if Polygons have exterior rings oriented clockwise and interior rings oriented counter-clockwise.
- ST_MakeEmptyCoverage - Availability: 2.4.0 Cover georeferenced area with a grid of empty raster tiles.

Functions enhanced in PostGIS 2.4

- ST_AsTWKB - Enhanced: 2.4.0 memory and speed improvements. Returns the geometry as TWKB, aka "Tiny Well-Known Binary"
- ST_Covers - Enhanced: 2.4.0 Support for polygon in polygon and line in polygon added for geography type Tests if every point of B lies in A
- ST_CurveToLine - Enhanced: 2.4.0 added support for max-deviation and max-angle tolerance, and for symmetric output. Converts a geometry containing curves to a linear geometry.
- ST_Project - Enhanced: 2.4.0 Allow negative distance and non-normalized azimuth. Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Reverse - Enhanced: 2.4.0 support for curves was introduced. Return the geometry with vertex order reversed.

Functions changed in PostGIS 2.4

- = - Changed: 2.4.0, in prior versions this was bounding box equality not a geometric equality. If you need bounding box equality, use instead. Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.
- ST_Node - Changed: 2.4.0 this function uses GEOSNode internally instead of GEOSUnaryUnion. This may cause the resulting linestrings to have a different order and direction compared to PostGIS < 2.4. Nodes a collection of lines.

13.12.11 PostGIS Functions new or enhanced in 2.3

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.3

- &&&(geometry,gidx) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.
- &&(box2df,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).
- @(box2df,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- Populate_Topology_Layer - Availability: 2.3.0 Adds missing entries to topology.layer table by reading metadata from topo tables.
- ST_ClusterDBSCAN - Availability: 2.3.0 Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterKMeans - Availability: 2.3.0 Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_GeneratePoints - Availability: 2.3.0 Generates a multipoint of random points contained in a Polygon or MultiPolygon.
- ST_GeometricMedian - Availability: 2.3.0 Returns the geometric median of a MultiPoint.
- ST_MakeLine - Availability: 2.3.0 - Support for MultiPoint input elements was introduced Creates a LineString from Point, MultiPoint, or LineString geometries.
- ST_MinimumBoundingRadius - Availability - 2.3.0 Returns the center point and radius of the smallest circle that contains a geometry.
- ST_MinimumClearance - Availability: 2.3.0 Returns the minimum clearance of a geometry, a measure of a geometry's robustness.
- ST_MinimumClearanceLine - Availability: 2.3.0 - requires GEOS >= 3.6.0 Returns the two-point LineString spanning a geometry's minimum clearance.
- ST_Normalize - Availability: 2.3.0 Return the geometry in its canonical form.
- ST_Points - Availability: 2.3.0 Returns a MultiPoint containing the coordinates of a geometry.
- ST_VoronoiLines - Availability: 2.3.0 Returns the boundaries of the Voronoi diagram of the vertices of a geometry.
- ST_VoronoiPolygons - Availability: 2.3.0 Returns the cells of the Voronoi diagram of the vertices of a geometry.

- ST_WrapX - Availability: 2.3.0 requires GEOS Wrap a geometry around an X value.
- TopoGeom_addElement - Availability: 2.3 Adds an element to the definition of a TopoGeometry.
- TopoGeom_remElement - Availability: 2.3 Removes an element from the definition of a TopoGeometry.
- ~(box2df,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).

Functions enhanced in PostGIS 2.3

- ST_Contains - Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if every point of B lies in A, and their interiors have a point in common
- ST_Covers - Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if every point of B lies in A
- ST_Expand - Enhanced: 2.3.0 support was added to expand a box by different amounts in different dimensions. Returns a bounding box expanded from another bounding box or a geometry.
- ST_Intersects - Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if two geometries intersect (they have at least one point in common)
- ST_Segmentize - Enhanced: 2.3.0 Segmentize geography now produces equal-length subsegments Returns a modified geometry/geography having no segment longer than a given distance.
- ST_Transform - Enhanced: 2.3.0 support for direct PROJ.4 text was introduced. Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Within - Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if every point of A lies in B, and their interiors have a point in common

Functions changed in PostGIS 2.3

- ST_PointN - Changed: 2.3.0 : negative indexing available (-1 is last point) Returns the Nth point in the first LineString or circular LineString in a geometry.

13.12.12 PostGIS Functions new or enhanced in 2.2

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.2

- <<->> - Availability: 2.2.0. Returns the n-D distance between the A and B geometries or bounding boxes
- ST_AsEncodedPolyline - Availability: 2.2.0 Returns an Encoded Polyline from a LineString geometry.
- ST_AsTWKB - Availability: 2.2.0 Returns the geometry as TWKB, aka "Tiny Well-Known Binary"
- ST_BoundingDiagonal - Availability: 2.2.0 Returns the diagonal of a geometry's bounding box.
- ST_CPAWithin - Availability: 2.2.0 Tests if the closest point of approach of two trajectoriesis within the specified distance.
- ST_ClipByBox2D - Availability: 2.2.0 Computes the portion of a geometry falling within a rectangle.
- ST_ClosestPointOfApproach - Availability: 2.2.0 Returns a measure at the closest point of approach of two trajectories.

- ST_ClusterIntersecting - Availability: 2.2.0 Aggregate function that clusters input geometries into connected sets.
- ST_ClusterWithin - Availability: 2.2.0 Aggregate function that clusters geometries by separation distance.
- ST_CountAgg - Availability: 2.2.0 Aggregate. Returns the number of pixels in a given band of a set of rasters. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the NODATA value.
- ST_CreateOverview - Availability: 2.2.0 Create an reduced resolution version of a given raster coverage.
- ST_DistanceCPA - Availability: 2.2.0 Returns the distance between the closest point of approach of two trajectories.
- ST_ForceCurve - Availability: 2.2.0 Upcast a geometry into its curved type, if applicable.
- ST_IsValidTrajectory - Availability: 2.2.0 Tests if the geometry is a valid trajectory.
- ST_LineFromEncodedPolyline - Availability: 2.2.0 Creates a LineString from an Encoded Polyline.
- ST_MapAlgebra (callback function version) - Availability: 2.2.0: Ability to add a mask Callback function version - Returns a one-band raster given one or more input rasters, band indexes and one user-specified callback function.
- ST_MemSize - Availability: 2.2.0 Returns the amount of space (in bytes) the raster takes.
- ST_RemoveRepeatedPoints - Availability: 2.2.0 Returns a version of a geometry with duplicate points removed.
- ST_Retile - Availability: 2.2.0 Return a set of configured tiles from an arbitrarily tiled raster coverage.
- ST_SetEffectiveArea - Availability: 2.2.0 Sets the effective area for each vertex, using the Visvalingam-Whyatt algorithm.
- ST_SimplifyVW - Availability: 2.2.0 Returns a simplified representation of a geometry, using the Visvalingam-Whyatt algorithm
- ST_Subdivide - Availability: 2.2.0 Computes a rectilinear subdivision of a geometry.
- ST_SummaryStatsAgg - Availability: 2.2.0 Aggregate. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a set of raster. Band 1 is assumed if no band is specified.
- ST_SwapOrdinates - Availability: 2.2.0 Returns a version of the given geometry with given ordinate values swapped.
- postgis.enable_outdb_rasters - Availability: 2.2.0 A boolean configuration option to enable access to out-db raster bands.
- postgis.gdal_datapath - Availability: 2.2.0 A configuration option to assign the value of GDAL's GDAL_DATA option. If not set, the environmentally set GDAL_DATA variable is used.
- postgis.gdal_enabled_drivers - Availability: 2.2.0 A configuration option to set the enabled GDAL drivers in the PostGIS environment. Affects the GDAL configuration variable GDAL_SKIP.
- |=| - Availability: 2.2.0. Returns the distance between A and B trajectories at their closest point of approach.

Functions enhanced in PostGIS 2.2

- <-> - Enhanced: 2.2.0 -- True KNN ("K nearest neighbor") behavior for geometry and geography. Note for geography KNN is based on sphere rather than spheroid. Returns the 2D distance between A and B.
- AsTopoJSON - Enhanced: 2.2.1 added support for puntal inputs Returns the TopoJSON representation of a topogeometry.
- ST_Area - Enhanced: 2.2.0 - measurement on spheroid performed with GeographicLib for improved accuracy and robustness. Requires PROJ >= 4.9.0 to take advantage of the new feature. Returns the area of a polygonal geometry.
- ST_AsX3D - Enhanced: 2.2.0: Support for GeoCoordinates and axis (x/y, long/lat) flipping. Look at options for details. Returns a Geometry in X3D xml node element format: ISO-IEC-19776-1.2-X3DEncodings-XML
- ST_Azimuth - Enhanced: 2.2.0 measurement on spheroid performed with GeographicLib for improved accuracy and robustness. Requires PROJ >= 4.9.0 to take advantage of the new feature. Returns the north-based azimuth of a line between two points.

- ST_Distance - Enhanced: 2.2.0 - measurement on spheroid performed with GeographicLib for improved accuracy and robustness. Requires PROJ >= 4.9.0 to take advantage of the new feature. Returns the distance between two geometry or geography values.
- ST_Scale - Enhanced: 2.2.0 support for scaling all dimension (factor parameter) was introduced. Scales a geometry by given factors.
- ST_Split - Enhanced: 2.2.0 support for splitting a line by a multiline, a multipoint or (multi)polygon boundary was introduced. Returns a collection of geometries created by splitting a geometry by another geometry.
- ST_Summary - Enhanced: 2.2.0 Added support for TIN and Curves Returns a text summary of the contents of a geometry.

Functions changed in PostGIS 2.2

- <-> - Changed: 2.2.0 -- Old hybrid-syntax workarounds may be slower once true KNN is available. See examples below. Returns the 2D distance between A and B.
- ST_3DClosestPoint - Changed: 2.2.0 - if 2 2D geometries are input, a 2D point is returned (instead of old behavior assuming 0 for missing Z). In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z. Returns the 3D point on g1 that is closest to g2. This is the first point of the 3D shortest line.
- ST_3DDistance - Changed: 2.2.0 - In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z. Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.
- ST_3DLongestLine - Changed: 2.2.0 - if 2 2D geometries are input, a 2D point is returned (instead of old behavior assuming 0 for missing Z). In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z. Returns the 3D longest line between two geometries
- ST_3DMaxDistance - Changed: 2.2.0 - In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z. Returns the 3D cartesian maximum distance (based on spatial ref) between two geometries in projected units.
- ST_3DShortestLine - Changed: 2.2.0 - if 2 2D geometries are input, a 2D point is returned (instead of old behavior assuming 0 for missing Z). In case of 2D and 3D, Z is no longer assumed to be 0 for missing Z. Returns the 3D shortest line between two geometries
- ST_DistanceSphere - Changed: 2.2.0 In prior versions this used to be called ST_Distance_Sphere Returns minimum distance in meters between two lon/lat geometries using a spherical earth model.
- ST_DistanceSpheroid - Changed: 2.2.0 In prior versions this was called ST_Distance_Spheroid Returns the minimum distance between two lon/lat geometries using a spheroidal earth model.
- ST_Equals - Changed: 2.2.0 Returns true even for invalid geometries if they are binary equal Tests if two geometries include the same set of points
- ST_LengthSpheroid - Changed: 2.2.0 In prior versions this was called ST_Length_Spheroid and had the alias ST_3DLength_Spheroid Returns the 2D or 3D length/perimeter of a lon/lat geometry on a spheroid.
- ST_MemSize - Changed: 2.2.0 name changed to ST_MemSize to follow naming convention. Returns the amount of memory space a geometry takes.
- ST_PointInsideCircle - Changed: 2.2.0 In prior versions this was called ST_Point_Inside_Circle Tests if a point geometry is inside a circle defined by a center and radius
- ValidateTopology - Changed: 2.2.0 values for id1 and id2 were swapped for 'edge crosses node' to be consistent with error description. Returns a set of validatetopology_returntype objects detailing issues with topology.

13.12.13 PostGIS Functions new or enhanced in 2.1

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.1

- = - Availability: 2.1.0 Returns TRUE if A's bounding box is the same as B's. Uses double precision bounding box.
- AsTopoJSON - Availability: 2.1.0 Returns the TopoJSON representation of a topogeometry.
- ST_Box2dFromGeoHash - Availability: 2.1.0 Return a BOX2D from a GeoHash string.
- ST_ColorMap - Availability: 2.1.0 Creates a new raster of up to four 8BUI bands (grayscale, RGB, RGBA) from the source raster and a specified band. Band 1 is assumed if not specified.
- ST_Contains - Availability: 2.1.0 Return true if no points of raster rastB lie in the exterior of raster rastA and at least one point of the interior of rastB lies in the interior of rastA.
- ST_ContainsProperly - Availability: 2.1.0 Return true if rastB intersects the interior of rastA but not the boundary or exterior of rastA.
- ST_CoveredBy - Availability: 2.1.0 Return true if no points of raster rastA lie outside raster rastB.
- ST_Covers - Availability: 2.1.0 Return true if no points of raster rastB lie outside raster rastA.
- ST_DFullyWithin - Availability: 2.1.0 Return true if rasters rastA and rastB are fully within the specified distance of each other.
- ST_DWithin - Availability: 2.1.0 Return true if rasters rastA and rastB are within the specified distance of each other.
- ST_DelaunayTriangles - Availability: 2.1.0 Returns the Delaunay triangulation of the vertices of a geometry.
- ST_Disjoint - Availability: 2.1.0 Return true if raster rastA does not spatially intersect rastB.
- ST_DumpValues - Availability: 2.1.0 Get the values of the specified band as a 2-dimension array.
- ST_ForceSFS - Availability: 2.1.0 Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_FromGDALRaster - Availability: 2.1.0 Returns a raster from a supported GDAL raster file.
- ST_GeomFromGeoHash - Availability: 2.1.0 Return a geometry from a GeoHash string.
- ST_InvDistWeight4ma - Availability: 2.1.0 Raster processing function that interpolates a pixel's value from the pixel's neighborhood.
- ST_MapAlgebra (callback function version) - Availability: 2.1.0 Callback function version - Returns a one-band raster given one or more input rasters, band indexes and one user-specified callback function.
- ST_MapAlgebra (expression version) - Availability: 2.1.0 Expression version - Returns a one-band raster given one or two input rasters, band indexes and one or more user-specified SQL expressions.
- ST_MinConvexHull - Availability: 2.1.0 Return the convex hull geometry of the raster excluding NODATA pixels.
- ST_MinDist4ma - Availability: 2.1.0 Raster processing function that returns the minimum distance (in number of pixels) between the pixel of interest and a neighboring pixel with value.
- ST_NearestValue - Availability: 2.1.0 Returns the nearest non-NODATA value of a given band's pixel specified by a columnx and rowy or a geometric point expressed in the same spatial reference coordinate system as the raster.
- ST_Neighborhood - Availability: 2.1.0 Returns a 2-D double precision array of the non-NODATA values around a given band's pixel specified by either a columnX and rowY or a geometric point expressed in the same spatial reference coordinate system as the raster.
- ST_NotSameAlignmentReason - Availability: 2.1.0 Returns text stating if rasters are aligned and if not aligned, a reason why.
- ST_Overlaps - Availability: 2.1.0 Return true if raster rastA and rastB intersect but one does not completely contain the other.

- ST_PixelAsCentroid - Availability: 2.1.0 Returns the centroid (point geometry) of the area represented by a pixel.
- ST_PixelAsCentroids - Availability: 2.1.0 Returns the centroid (point geometry) for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The point geometry is the centroid of the area represented by a pixel.
- ST_PixelAsPoint - Availability: 2.1.0 Returns a point geometry of the pixel's upper-left corner.
- ST_PixelAsPoints - Availability: 2.1.0 Returns a point geometry for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The coordinates of the point geometry are of the pixel's upper-left corner.
- ST_PixelOfValue - Availability: 2.1.0 Get the columnx, rowy coordinates of the pixel whose value equals the search value.
- ST_PointFromGeoHash - Availability: 2.1.0 Return a point from a GeoHash string.
- ST_RasterToWorldCoord - Availability: 2.1.0 Returns the raster's upper left corner as geometric X and Y (longitude and latitude) given a column and row. Column and row starts at 1.
- ST_Resize - Availability: 2.1.0 Requires GDAL 1.6.1+ Resize a raster to a new width/height
- ST_Roughness - Availability: 2.1.0 Returns a raster with the calculated "roughness" of a DEM.
- ST_SetValues - Availability: 2.1.0 Returns modified raster resulting from setting the values of a given band.
- ST_Simplify - Availability: 2.1.0 Returns a "simplified" geometry version of the given TopoGeometry using the Douglas-Peucker algorithm.
- ST_Summary - Availability: 2.1.0 Returns a text summary of the contents of the raster.
- ST_TPI - Availability: 2.1.0 Returns a raster with the calculated Topographic Position Index.
- ST_TRI - Availability: 2.1.0 Returns a raster with the calculated Terrain Ruggedness Index.
- ST_Tile - Availability: 2.1.0 Returns a set of rasters resulting from the split of the input raster based upon the desired dimensions of the output rasters.
- ST_Touches - Availability: 2.1.0 Return true if raster rastA and rastB have at least one point in common but their interiors do not intersect.
- ST_Union - Availability: 2.1.0 ST_Union(rast, unionarg) variant was introduced. Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- ST_Within - Availability: 2.1.0 Return true if no points of raster rastA lie in the exterior of raster rastB and at least one point of the interior of rastA lies in the interior of rastB.
- ST_WorldToRasterCoord - Availability: 2.1.0 Returns the upper left corner as column and row given geometric X and Y (longitude and latitude) or a point geometry expressed in the spatial reference coordinate system of the raster.
- UpdateRasterSRID - Availability: 2.1.0 Change the SRID of all rasters in the user-specified column and table.
- clearTopoGeom - Availability: 2.1 Clears the content of a topo geometry.
- postgis_sfcgal_version - Availability: 2.1.0 Returns the version of SFCGAL in use

Functions enhanced in PostGIS 2.1

- ST_AddBand - Enhanced: 2.1.0 support for addbandarg added. Returns a raster with the new band(s) of given type added with given initial value in the given index location. If no index is specified, the band is added to the end.
- ST_AddBand - Enhanced: 2.1.0 support for new out-db bands added. Returns a raster with the new band(s) of given type added with given initial value in the given index location. If no index is specified, the band is added to the end.
- ST_AsBinary/ST_AsWKB - Enhanced: 2.1.0 Addition of outasin Return the Well-Known Binary (WKB) representation of the raster.
- ST_AsGML - Enhanced: 2.1.0 id support was introduced, for GML 3. Return the geometry as a GML version 2 or 3 element.

- ST_Aspect - Enhanced: 2.1.0 Uses ST_MapAlgebra() and added optional interpolate_nodata function parameter Returns the aspect (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_Boundary - Enhanced: 2.1.0 support for Triangle was introduced Returns the boundary of a geometry.
- ST_Clip - Enhanced: 2.1.0 Rewritten in C Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.
- ST_DWithin - Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details. Tests if two geometries are within a given distance
- ST_DWithin - Enhanced: 2.1.0 support for curved geometries was introduced. Tests if two geometries are within a given distance
- ST_Distance - Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details. Returns the distance between two geometry or geography values.
- ST_Distance - Enhanced: 2.1.0 - support for curved geometries was introduced. Returns the distance between two geometry or geography values.
- ST_Distinct4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the number of unique pixel values in a neighborhood.
- ST_DumpPoints - Enhanced: 2.1.0 Faster speed. Reimplemented as native-C. Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_HillShade - Enhanced: 2.1.0 Uses ST_MapAlgebra() and added optional interpolate_nodata function parameter Returns the hypothetical illumination of an elevation raster band using provided azimuth, altitude, brightness and scale inputs.
- ST_MakeValid - Enhanced: 2.1.0, added support for GEOMETRYCOLLECTION and MULTIPOINT. Attempts to make an invalid geometry valid without losing vertices.
- ST_Max4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the maximum pixel value in a neighborhood.
- ST_Mean4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the mean pixel value in a neighborhood.
- ST_Min4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the minimum pixel value in a neighborhood.
- ST_PixelAsPolygons - Enhanced: 2.1.0 exclude_nodata_value optional argument was added. Returns the polygon geometry that bounds every pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel.
- ST_Polygon - Enhanced: 2.1.0 Improved Speed (fully C-Based) and the returning multipolygon is ensured to be valid. Returns a multipolygon geometry formed by the union of pixels that have a pixel value that is not no data value. If no band number is specified, band num defaults to 1.
- ST_Range4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the range of pixel values in a neighborhood.
- ST_SameAlignment - Enhanced: 2.1.0 addition of Aggregate variant Returns true if rasters have same skew, scale, spatial ref, and offset (pixels can be put on same grid without cutting into pixels) and false if they don't with notice detailing issue.
- ST_Segmentize - Enhanced: 2.1.0 support for geography was introduced. Returns a modified geometry/geography having no segment longer than a given distance.
- ST_SetGeoReference - Enhanced: 2.1.0 Addition of ST_SetGeoReference(raster, double precision, ...) variant Set Georeference 6 georeference parameters in a single call. Numbers should be separated by white space. Accepts inputs in GDAL or ESRI format. Default is GDAL.

- ST_SetValue - Enhanced: 2.1.0 Geometry variant of ST_SetValue() now supports any geometry type, not just point. The geometry variant is a wrapper around the geomval[] variant of ST_SetValues() Returns modified raster resulting from setting the value of a given band in a given columnx, rowy pixel or the pixels that intersect a particular geometry. Band numbers start at 1 and assumed to be 1 if not specified.
- ST_Slope - Enhanced: 2.1.0 Uses ST_MapAlgebra() and added optional units, scale, interpolate_nodata function parameters Returns the slope (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_StdDev4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the standard deviation of pixel values in a neighborhood.
- ST_Sum4ma - Enhanced: 2.1.0 Addition of Variant 2 Raster processing function that calculates the sum of all pixel values in a neighborhood.
- ST_Summary - Enhanced: 2.1.0 S flag to denote if has a known spatial reference system Returns a text summary of the contents of a geometry.
- ST_Transform - Enhanced: 2.1.0 Addition of ST_Transform(rast, alignto) variant Reprojects a raster in a known spatial reference system to another known spatial reference system using specified resampling algorithm. Options are NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos defaulting to NearestNeighbor.
- ST_Union - Enhanced: 2.1.0 Improved Speed (fully C-Based). Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- ST_Union - Enhanced: 2.1.0 ST_Union(rast) (variant 1) unions all bands of all input rasters. Prior versions of PostGIS assumed the first band. Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- ST_Union - Enhanced: 2.1.0 ST_Union(rast, uniontype) (variant 4) unions all bands of all input rasters. Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- toTopoGeom - Enhanced: 2.1.0 adds the version taking an existing TopoGeometry. Converts a simple Geometry into a topo geometry.

Functions changed in PostGIS 2.1

- ST_Aspect - Changed: 2.1.0 In prior versions, return values were in radians. Now, return values default to degrees Returns the aspect (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_EstimatedExtent - Changed: 2.1.0. Up to 2.0.x this was called ST_Estimated_Extent. Returns the estimated extent of a spatial table.
- ST_Force2D - Changed: 2.1.0. Up to 2.0.x this was called ST_Force_2D. Force the geometries into a "2-dimensional mode".
- ST_Force3D - Changed: 2.1.0. Up to 2.0.x this was called ST_Force_3D. Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.
- ST_Force3DM - Changed: 2.1.0. Up to 2.0.x this was called ST_Force_3DM. Force the geometries into XYM mode.
- ST_Force3DZ - Changed: 2.1.0. Up to 2.0.x this was called ST_Force_3DZ. Force the geometries into XYZ mode.
- ST_Force4D - Changed: 2.1.0. Up to 2.0.x this was called ST_Force_4D. Force the geometries into XYZM mode.
- ST_ForceCollection - Changed: 2.1.0. Up to 2.0.x this was called ST_Force_Collection. Convert the geometry into a GEOMETRYCOLLECTION.
- ST_HillShade - Changed: 2.1.0 In prior versions, azimuth and altitude were expressed in radians. Now, azimuth and altitude are expressed in degrees Returns the hypothetical illumination of an elevation raster band using provided azimuth, altitude, brightness and scale inputs.
- ST_LineInterpolatePoint - Changed: 2.1.0. Up to 2.0.x this was called ST_Line_Interpolate_Point. Returns a point interpolated along a line at a fractional location.
- ST_LineLocatePoint - Changed: 2.1.0. Up to 2.0.x this was called ST_Line_Locate_Point. Returns the fractional location of the closest point on a line to a point.

- ST_LineSubstring - Changed: 2.1.0. Up to 2.0.x this was called ST_Line_Substring. Returns the part of a line between two fractional locations.
- ST_PixelAsCentroids - Changed: 2.1.1 Changed behavior of exclude_nodata_value. Returns the centroid (point geometry) for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The point geometry is the centroid of the area represented by a pixel.
- ST_PixelAsPoints - Changed: 2.1.1 Changed behavior of exclude_nodata_value. Returns a point geometry for each pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel. The coordinates of the point geometry are of the pixel's upper-left corner.
- ST_PixelAsPolygons - Changed: 2.1.1 Changed behavior of exclude_nodata_value. Returns the polygon geometry that bounds every pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel.
- ST_Polygon - Changed: 2.1.0 In prior versions would sometimes return a polygon, changed to always return multipolygon. Returns a multipolygon geometry formed by the union of pixels that have a pixel value that is not no data value. If no band number is specified, band num defaults to 1.
- ST_RasterToWorldCoordX - Changed: 2.1.0 In prior versions, this was called ST_Raster2WorldCoordX Returns the geometric X coordinate upper left of a raster, column and row. Numbering of columns and rows starts at 1.
- ST_RasterToWorldCoordY - Changed: 2.1.0 In prior versions, this was called ST_Raster2WorldCoordY Returns the geometric Y coordinate upper left corner of a raster, column and row. Numbering of columns and rows starts at 1.
- ST_Rescale - Changed: 2.1.0 Works on rasters with no SRID Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_Reskew - Changed: 2.1.0 Works on rasters with no SRID Resample a raster by adjusting only its skew (or rotation parameters). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.
- ST_Segmentize - Changed: 2.1.0 As a result of the introduction of geography support, the usage ST_Segmentize('LINESTRING(1 2, 3 4)', 0.5) causes an ambiguous function error. The input needs to be properly typed as a geometry or geography. Use ST_GeomFromText, ST_GeogFromText or a cast to the required type (e.g. ST_Segmentize('LINESTRING(1 2, 3 4)'::geometry, 0.5)) Returns a modified geometry/geography having no segment longer than a given distance.
- ST_Slope - Changed: 2.1.0 In prior versions, return values were in radians. Now, return values default to degrees Returns the slope (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_SnapToGrid - Changed: 2.1.0 Works on rasters with no SRID Resample a raster by snapping it to a grid. New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.
- ST_WorldToRasterCoordX - Changed: 2.1.0 In prior versions, this was called ST_World2RasterCoordX Returns the column in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw) represented in world spatial reference system of raster.
- ST_WorldToRasterCoordY - Changed: 2.1.0 In prior versions, this was called ST_World2RasterCoordY Returns the row in the raster of the point geometry (pt) or a X and Y world coordinate (xw, yw) represented in world spatial reference system of raster.

13.12.14 PostGIS Functions new or enhanced in 2.0

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.0

- && - Availability: 2.0.0 Returns TRUE if A's bounding box intersects B's bounding box.
- &&& - Availability: 2.0.0 Returns TRUE if A's n-D bounding box intersects B's n-D bounding box.

- <#> - Availability: 2.0.0. Returns the 2D distance between A and B bounding boxes.
- <-> - Availability: 2.0.0 -- Weak KNN provides nearest neighbors based on geometry centroid distances instead of true distances. Exact results for points, inexact for all other types. Returns the 2D distance between A and B.
- @ - Availability: 2.0.0 raster @ raster, raster @ geometry introduced Returns TRUE if A's bounding box is contained by B's. Uses double precision bounding box.
- @ - Availability: 2.0.5 geometry @ raster introduced Returns TRUE if A's bounding box is contained by B's. Uses double precision bounding box.
- AddEdge - Availability: 2.0.0 Adds a linestring edge to the edge table and associated start and end points to the point nodes table of the specified topology schema using the specified linestring geometry and returns the edgeid of the new (or existing) edge.
- AddFace - Availability: 2.0.0 Registers a face primitive to a topology and gets its identifier.
- AddNode - Availability: 2.0.0 Adds a point node to the node table in the specified topology schema and returns the nodeid of new node. If point already exists as node, the existing nodeid is returned.
- AddOverviewConstraints - Availability: 2.0.0 Tag a raster column as being an overview of another.
- AddRasterConstraints - Availability: 2.0.0 Adds raster constraints to a loaded raster table for a specific column that constrains spatial ref, scaling, blocksize, alignment, bands, band type and a flag to denote if raster column is regularly blocked. The table must be loaded with data for the constraints to be inferred. Returns true if the constraint setting was accomplished and issues a notice otherwise.
- AsGML - Availability: 2.0.0 Returns the GML representation of a topogeometry.
- CopyTopology - Availability: 2.0.0 Makes a copy of a topology (nodes, edges, faces, layers and TopoGeometries) into a new schema
- DropOverviewConstraints - Availability: 2.0.0 Untag a raster column from being an overview of another.
- DropRasterConstraints - Availability: 2.0.0 Drops PostGIS raster constraints that refer to a raster table column. Useful if you need to reload data or update your raster column data.
- GetEdgeByPoint - Availability: 2.0.0 Finds the edge-id of an edge that intersects a given point.
- GetFaceByPoint - Availability: 2.0.0 Finds face intersecting a given point.
- GetNodeByPoint - Availability: 2.0.0 Finds the node-id of a node at a point location.
- GetNodeEdges - Availability: 2.0 Returns an ordered set of edges incident to the given node.
- GetRingEdges - Availability: 2.0.0 Returns the ordered set of signed edge identifiers met by walking on ana given edge side.
- GetTopoGeomElements - Availability: 2.0.0 Returns a set of topoelement objects containing the topological element_id,element_type of the given TopoGeometry (primitive elements).
- GetTopologySRID - Availability: 2.0.0 Returns the SRID of a topology in the topology.topology table given the name of the topology.
- Polygonize - Availability: 2.0.0 Finds and registers all faces defined by topology edges.
- ST_3DClosestPoint - Availability: 2.0.0 Returns the 3D point on g1 that is closest to g2. This is the first point of the 3D shortest line.
- ST_3DDFullyWithin - Availability: 2.0.0 Tests if two 3D geometries are entirely within a given 3D distance
- ST_3DDWithin - Availability: 2.0.0 Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - Availability: 2.0.0 Returns the 3D cartesian minimum distance (based on spatial ref) between two geometries in projected units.

- ST_3DIntersects - Availability: 2.0.0 Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_3DLongestLine - Availability: 2.0.0 Returns the 3D longest line between two geometries
- ST_3DMaxDistance - Availability: 2.0.0 Returns the 3D cartesian maximum distance (based on spatial ref) between two geometries in projected units.
- ST_3DShortestLine - Availability: 2.0.0 Returns the 3D shortest line between two geometries
- ST_AddEdgeModFace - Availability: 2.0 Add a new edge and, if in doing so it splits a face, modify the original face and add a new face.
- ST_AddEdgeNewFaces - Availability: 2.0 Add a new edge and, if in doing so it splits a face, delete the original face and replace it with two new faces.
- ST_AsGDALRaster - Availability: 2.0.0 - requires GDAL >= 1.6.0. Return the raster tile in the designated GDAL Raster format. Raster formats are one of those supported by your compiled library. Use ST_GDALDrivers() to get a list of formats supported by your library.
- ST_AsJPEG - Availability: 2.0.0 - requires GDAL >= 1.6.0. Return the raster tile selected bands as a single Joint Photographic Exports Group (JPEG) image (byte array). If no band is specified and 1 or more than 3 bands, then only the first band is used. If only 3 bands then all 3 bands are used and mapped to RGB.
- ST_AsLatLonText - Availability: 2.0 Return the Degrees, Minutes, Seconds representation of the given point.
- ST_AsPNG - Availability: 2.0.0 - requires GDAL >= 1.6.0. Return the raster tile selected bands as a single portable network graphics (PNG) image (byte array). If 1, 3, or 4 bands in raster and no bands are specified, then all bands are used. If more 2 or more than 4 bands and no bands specified, then only band 1 is used. Bands are mapped to RGB or RGBA space.
- ST_AsRaster - Availability: 2.0.0 - requires GDAL >= 1.6.0. Converts a PostGIS geometry to a PostGIS raster.
- ST_AsTIFF - Availability: 2.0.0 - requires GDAL >= 1.6.0. Return the raster selected bands as a single TIFF image (byte array). If no band is specified or any of specified bands does not exist in the raster, then will try to use all bands.
- ST_AsX3D - Availability: 2.0.0: ISO-IEC-19776-1.2-X3DEncodings-XML Returns a Geometry in X3D xml node element format: ISO-IEC-19776-1.2-X3DEncodings-XML
- ST_Aspect - Availability: 2.0.0 Returns the aspect (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_Band - Availability: 2.0.0 Returns one or more bands of an existing raster as a new raster. Useful for building new rasters from existing rasters.
- ST_BandIsNoData - Availability: 2.0.0 Returns true if the band is filled with only nodata values.
- ST_Clip - Availability: 2.0.0 Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.
- ST_CollectionHomogenize - Availability: 2.0.0 Returns the simplest representation of a geometry collection.
- ST_ConcaveHull - Availability: 2.0.0 Computes a possibly concave geometry that contains all input geometry vertices
- ST_Count - Availability: 2.0.0 Returns the number of pixels in a given band of a raster or raster coverage. If no band is specified defaults to band 1. If exclude_nodata_value is set to true, will only count pixels that are not equal to the nodata value.
- ST_CreateTopoGeo - Availability: 2.0 Adds a collection of geometries to a given empty topology and returns a message detailing success.
- ST_Distinct4ma - Availability: 2.0.0 Raster processing function that calculates the number of unique pixel values in a neighborhood.
- ST_FlipCoordinates - Availability: 2.0.0 Returns a version of a geometry with X and Y axis flipped.

- ST_GDALDrivers - Availability: 2.0.0 - requires GDAL >= 1.6.0. Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster
- ST_GeomFromGeoJSON - Availability: 2.0.0 requires - JSON-C >= 0.9 Takes as input a geojson representation of a geometry and outputs a PostGIS geometry object
- ST_GetFaceEdges - Availability: 2.0 Returns a set of ordered edges that bound aface.
- ST_HasNoBand - Availability: 2.0.0 Returns true if there is no band with given band number. If no band number is specified, then band number 1 is assumed.
- ST_HillShade - Availability: 2.0.0 Returns the hypothetical illumination of an elevation raster band using provided azimuth, altitude, brightness and scale inputs.
- ST_Histogram - Availability: 2.0.0 Returns a set of record summarizing a raster or raster coverage data distribution separate bin ranges. Number of bins are autocomputed if not specified.
- ST_InterpolatePoint - Availability: 2.0.0 Returns the interpolated measure of a geometry closest to a point.
- ST_IsEmpty - Availability: 2.0.0 Returns true if the raster is empty (width = 0 and height = 0). Otherwise, returns false.
- ST_IsValidDetail - Availability: 2.0.0 Returns a valid_detail row stating if a geometry is valid or if not a reason and a location.
- ST_IsValidReason - Availability: 2.0 version taking flags. Returns text stating if a geometry is valid, or a reason for invalidity.
- ST_MakeLine - Availability: 2.0.0 - Support for LineString input elements was introduced Creates a LineString from Point, MultiPoint, or LineString geometries.
- ST_MakeValid - Availability: 2.0.0 Attempts to make an invalid geometry valid without losing vertices.
- ST_MapAlgebraExpr - Availability: 2.0.0 1 raster band version: Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation on the input raster band and of pixeltype provided. Band 1 is assumed if no band is specified.
- ST_MapAlgebraExpr - Availability: 2.0.0 2 raster band version: Creates a new one band raster formed by applying a valid PostgreSQL algebraic operation on the two input raster bands and of pixeltype provided. band 1 of each raster is assumed if no band numbers are specified. The resulting raster will be aligned (scale, skew and pixel corners) on the grid defined by the first raster and have its extent defined by the "extenttype" parameter. Values for "extenttype" can be: INTERSECTION, UNION, FIRST, SECOND.
- ST_MapAlgebraFct - Availability: 2.0.0 1 band version - Creates a new one band raster formed by applying a valid PostgreSQL function on the input raster band and of pixeltype provided. Band 1 is assumed if no band is specified.
- ST_MapAlgebraFct - Availability: 2.0.0 2 band version - Creates a new one band raster formed by applying a valid PostgreSQL function on the 2 input raster bands and of pixeltype provided. Band 1 is assumed if no band is specified. Extent type defaults to INTERSECTION if not specified.
- ST_MapAlgebraFctNgb - Availability: 2.0.0 1-band version: Map Algebra Nearest Neighbor using user-defined PostgreSQL function. Return a raster which values are the result of a PLPGSQL user function involving a neighborhood of values from the input raster band.
- ST_Max4ma - Availability: 2.0.0 Raster processing function that calculates the maximum pixel value in a neighborhood.
- ST_Mean4ma - Availability: 2.0.0 Raster processing function that calculates the mean pixel value in a neighborhood.
- ST_Min4ma - Availability: 2.0.0 Raster processing function that calculates the minimum pixel value in a neighborhood.
- ST_ModEdgeHeal - Availability: 2.0 Heals two edges by deleting the node connecting them, modifying the first edgeand deleting the second edge. Returns the id of the deleted node.
- ST_MoveIsoNode - Availability: 2.0.0 Moves an isolated node in a topology from one point to another. If new apoint geometry exists as a node an error is thrown. Returns description of move.
- ST_NewEdgeHeal - Availability: 2.0 Heals two edges by deleting the node connecting them, deleting both edges,and replacing them with an edge whose direction is the same as the firstedge provided.

- ST_Node - Availability: 2.0.0 Nodes a collection of lines.
- ST_NumPatches - Availability: 2.0.0 Returns the number of faces in a PolyhedralSurface or TIN.
- ST_OffsetCurve - Availability: 2.0 Returns an offset line at a given distance and side from an input line.
- ST_PatchN - Availability: 2.0.0 Returns the Nth face of a PolyhedralSurface or TIN.
- ST_Perimeter - Availability 2.0.0: Support for geography was introduced Returns the length of the boundary of a polygonal geometry or geography.
- ST_PixelAsPolygon - Availability: 2.0.0 Returns the polygon geometry that bounds the pixel for a particular row and column.
- ST_PixelAsPolygons - Availability: 2.0.0 Returns the polygon geometry that bounds every pixel of a raster band along with the value, the X and the Y raster coordinates of each pixel.
- ST_Project - Availability: 2.0.0 Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Quantile - Availability: 2.0.0 Compute quantiles for a raster or raster table coverage in the context of the sample or population. Thus, a value could be examined to be at the raster's 25%, 50%, 75% percentile.
- ST_Range4ma - Availability: 2.0.0 Raster processing function that calculates the range of pixel values in a neighborhood.
- ST_Reclass - Availability: 2.0.0 Creates a new raster composed of band types reclassified from original. The nband is the band to be changed. If nband is not specified assumed to be 1. All other bands are returned unchanged. Use case: convert a 16BUI band to a 8BUI and so forth for simpler rendering as viewable formats.
- ST_RelateMatch - Availability: 2.0.0 Tests if a DE-9IM Intersection Matrix matches an Intersection Matrix pattern
- ST_RemEdgeModFace - Availability: 2.0 Removes an edge, and if the edge separates two facesdeletes one face and modifies the other face to cover the space of both.
- ST_RemEdgeNewFace - Availability: 2.0 Removes an edge and, if the removed edge separated two faces,delete the original faces and replace them with a new face.
- ST_Resample - Availability: 2.0.0 Requires GDAL 1.6.1+ Resample a raster using a specified resampling algorithm, new dimensions, an arbitrary grid corner and a set of raster georeferencing attributes defined or borrowed from another raster.
- ST_Rescale - Availability: 2.0.0 Requires GDAL 1.6.1+ Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_Reskew - Availability: 2.0.0 Requires GDAL 1.6.1+ Resample a raster by adjusting only its skew (or rotation parameters). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.
- ST_SameAlignment - Availability: 2.0.0 Returns true if rasters have same skew, scale, spatial ref, and offset (pixels can be put on same grid without cutting into pixels) and false if they don't with notice detailing issue.
- ST_SetBandIsNoData - Availability: 2.0.0 Sets the isnodata flag of the band to TRUE.
- ST_SharedPaths - Availability: 2.0.0 Returns a collection containing paths shared by the two input linestrings/multilinestrings.
- ST_Slope - Availability: 2.0.0 Returns the slope (in degrees by default) of an elevation raster band. Useful for analyzing terrain.
- ST_Snap - Availability: 2.0.0 Snap segments and vertices of input geometry to vertices of a reference geometry.
- ST_SnapToGrid - Availability: 2.0.0 Requires GDAL 1.6.1+ Resample a raster by snapping it to a grid. New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline or Lanczos resampling algorithm. Default is NearestNeighbor.
- ST_Split - Availability: 2.0.0 requires GEOS Returns a collection of geometries created by splitting a geometry by another geometry.

- ST_StdDev4ma - Availability: 2.0.0 Raster processing function that calculates the standard deviation of pixel values in a neighborhood.
- ST_Sum4ma - Availability: 2.0.0 Raster processing function that calculates the sum of all pixel values in a neighborhood.
- ST_SummaryStats - Availability: 2.0.0 Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.
- ST_Transform - Availability: 2.0.0 Requires GDAL 1.6.1+ Reprojects a raster in a known spatial reference system to another known spatial reference system using specified resampling algorithm. Options are NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos defaulting to NearestNeighbor.
- ST_UnaryUnion - Availability: 2.0.0 Computes the union of the components of a single geometry.
- ST_Union - Availability: 2.0.0 Returns the union of a set of raster tiles into a single raster composed of 1 or more bands.
- ST_ValueCount - Availability: 2.0.0 Returns a set of records containing a pixel band value and count of the number of pixels in a given band of a raster (or a raster coverage) that have a given set of values. If no band is specified defaults to band 1. By default nodata value pixels are not counted. and all other values in the pixel are output and pixel band values are rounded to the nearest integer.
- TopoElementArray_Agg - Availability: 2.0.0 Returns a topoelementarray for a set of element_id, type arrays (topoelements).
- TopoGeo_AddLineString - Availability: 2.0.0 Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.
- TopoGeo_AddPoint - Availability: 2.0.0 Adds a point to an existing topology using a tolerance and possibly splitting an existing edge.
- TopoGeo_AddPolygon - Availability: 2.0.0 Adds a polygon to an existing topology using a tolerance and possibly splitting existing edges/faces. Returns face identifiers.
- TopologySummary - Availability: 2.0.0 Takes a topology name and provides summary totals of types of objects in topology.
- toTopoGeom - Availability: 2.0 Converts a simple Geometry into a topo geometry.
- ~ - Availability: 2.0.0 Returns TRUE if A's bounding box is contains B's. Uses double precision bounding box.
- ~= - Availability: 2.0.0 Returns TRUE if A's bounding box is the same as B's.

Functions enhanced in PostGIS 2.0

- && - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.
- AddGeometryColumn - Enhanced: 2.0.0 use_typmod argument introduced. Defaults to creating typmod geometry column instead of constraint-based. Adds a geometry column to an existing table.
- Box2D - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns a BOX3D representing the 3D extent of a geometry.
- CreateTopology - Enhanced: 2.0 added support for hasZ. Creates a new topology schema and registers it in the topology.topology table.
- GeometryType - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns the type of a geometry as text.
- Populate_Geometry_Columns - Enhanced: 2.0.0 use_typmod optional argument was introduced that allows controlling if columns are created with typmodifiers or with check constraints. Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.

- ST_3DExtent - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Aggregate function that returns the 3D bounding box of geometries.
- ST_Affine - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Apply a 3D affine transformation to a geometry.
- ST_Area - Enhanced: 2.0.0 - support for 2D polyhedral surfaces was introduced. Returns the area of a polygonal geometry.
- ST_AsBinary - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsBinary - Enhanced: 2.0.0 support for higher coordinate dimensions was introduced. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsBinary - Enhanced: 2.0.0 support for specifying endian with geography was introduced. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - Enhanced: 2.0.0 support for Geography, Polyhedral surfaces, Triangles and TIN was introduced. Return the Well-Known Text (WKT) representation of the geometry with SRID meta data.
- ST_AsGML - Enhanced: 2.0.0 prefix support was introduced. Option 4 for GML3 was introduced to allow using LineString instead of Curve tag for lines. GML3 Support for Polyhedral surfaces and TINS was introduced. Option 32 was introduced to output the box. Return the geometry as a GML version 2 or 3 element.
- ST_AsKML - Enhanced: 2.0.0 - Add prefix namespace, use default and named args Return the geometry as a KML element.
- ST_Azimuth - Enhanced: 2.0.0 support for geography was introduced. Returns the north-based azimuth of a line between two points.
- ST_Dimension - Enhanced: 2.0.0 support for Polyhedral surfaces and TINs was introduced. No longer throws an exception if given empty geometry. Returns the topological dimension of a geometry.
- ST_Dump - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_Expand - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns a bounding box expanded from another bounding box or a geometry.
- ST_Extent - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Aggregate function that returns the bounding box of geometries.
- ST_Force2D - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Force the geometries into a "2-dimensional mode".
- ST_Force3D - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Force the geometries into XYZ mode. This is an alias for ST_Force3DZ.
- ST_Force3DZ - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Force the geometries into XYZ mode.
- ST_ForceCollection - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Convert the geometry into a GEOMETRYCOLLECTION.
- ST_ForceRHR - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Force the orientation of the vertices in a polygon to follow the Right-Hand-Rule.
- ST_GMLToSQL - Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced. Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GMLToSQL - Enhanced: 2.0.0 default srid optional parameter added. Returns a geometry from GML. Alias for ST_GeomFromGM

- ST_GeomFromEWKB - Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced. Return a specified ST_Geometry value from Extended Well-Known Binary representation (EWKB).
- ST_GeomFromEWKT - Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced. Return a specified ST_Geometry value from Extended Well-Known Text representation (EWKT).
- ST_GeomFromGML - Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced. Takes as input GML representation of geometry and outputs a PostGIS geometry object
- ST_GeomFromGML - Enhanced: 2.0.0 default srid optional parameter added. Takes as input GML representation of geometry and outputs a PostGIS geometry object
- ST_GeometryN - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Return an element of a geometry collection.
- ST_GeometryType - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Returns the SQL-MM type of a geometry as text.
- ST_IsClosed - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Tests if a LineStrings's start and end points are coincident. For a PolyhedralSurface tests if it is closed (volumetric).
- ST_MakeEnvelope - Enhanced: 2.0: Ability to specify an envelope without specifying an SRID was introduced. Creates a rectangular Polygon from minimum and maximum coordinates.
- ST_MakeValid - Enhanced: 2.0.1, speed improvements Attempts to make an invalid geometry valid without losing vertices.
- ST_NPoints - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Returns the number of points (vertices) in a geometry.
- ST_NumGeometries - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Returns the number of elements in a geometry collection.
- ST_Relate - Enhanced: 2.0.0 - added support for specifying boundary node rule. Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix
- ST_Rotate - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Rotates a geometry about an origin point.
- ST_Rotate - Enhanced: 2.0.0 additional parameters for specifying the origin of rotation were added. Rotates a geometry about an origin point.
- ST_RotateX - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Rotates a geometry about the X axis.
- ST_RotateY - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Rotates a geometry about the Y axis.
- ST_RotateZ - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Rotates a geometry about the Z axis.
- ST_Scale - Enhanced: 2.0.0 support for Polyhedral surfaces, Triangles and TIN was introduced. Scales a geometry by given factors.
- ST_ShiftLongitude - Enhanced: 2.0.0 support for Polyhedral surfaces and TIN was introduced. Shifts the longitude coordinates of a geometry between -180..180 and 0..360.
- ST_Summary - Enhanced: 2.0.0 added support for geography Returns a text summary of the contents of a geometry.
- ST_Transform - Enhanced: 2.0.0 support for Polyhedral surfaces was introduced. Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Value - Enhanced: 2.0.0 exclude_nodata_value optional argument was added. Returns the value of a given band in a given columnx, rowy pixel or at a particular geometric point. Band numbers start at 1 and assumed to be 1 if not specified. If exclude_nodata_value is set to false, then all pixels include nodata pixels are considered to intersect and return value. If exclude_nodata_value is not passed in then reads it from metadata of raster.

- ValidateTopology - Enhanced: 2.0.0 more efficient edge crossing detection and fixes for false positives that were existent in prior versions. Returns a set of validatetopology_returntype objects detailing issues with topology.

Functions changed in PostGIS 2.0

- AddGeometryColumn - Changed: 2.0.0 This function no longer updates geometry_columns since geometry_columns is a view that reads from system catalogs. It by default also does not create constraints, but instead uses the built in type modifier behavior of PostgreSQL. So for example building a wgs84 POINT column with this function is now equivalent to: ALTER TABLE some_table ADD COLUMN geom geometry(Point,4326); Adds a geometry column to an existing table.
- AddGeometryColumn - Changed: 2.0.0 If you require the old behavior of constraints use the default use_typmod, but set it to false. Adds a geometry column to an existing table.
- AddGeometryColumn - Changed: 2.0.0 Views can no longer be manually registered in geometry_columns, however views built against geometry typmod tables geometries and used without wrapper functions will register themselves correctly because they inherit the typmod behavior of their parent table column. Views that use geometry functions that output other geometries will need to be cast to typmod geometries for these view geometry columns to be registered correctly in geometry_columns. Refer to . Adds a geometry column to an existing table.
- Box3D - Changed: 2.0.0 In pre-2.0 versions, there used to be a box2d instead of box3d. Since box2d is a deprecated type, this was changed to box3d. Returns the box 3d representation of the enclosing box of the raster.
- DropGeometryColumn - Changed: 2.0.0. Retained for backward compatibility after geometry_columns became a view of the system catalogs. Removes a geometry column from a spatial table.
- DropGeometryTable - Changed: 2.0.0. Retained for backward compatibility after geometry_columns became a view of the system catalogs. Drops a table and all its references in geometry_columns.
- Populate_Geometry_Columns - Changed: 2.0.0 By default, now uses type modifiers instead of check constraints to constrain geometry types. You can still use check constraint behavior instead by using the new use_typmod and setting it to false. Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.
- ST_3DExtent - Changed: 2.0.0 In prior versions this used to be called ST_Extent3D Aggregate function that returns the 3D bounding box of geometries.
- ST_3DLength - Changed: 2.0.0 In prior versions this used to be called ST_Length3D Returns the 3D length of a linear geometry.
- ST_3DMakeBox - Changed: 2.0.0 In prior versions this used to be called ST_MakeBox3D Creates a BOX3D defined by two 3D point geometries.
- ST_3DPerimeter - Changed: 2.0.0 In prior versions this used to be called ST_Perimeter3D Returns the 3D perimeter of a polygonal geometry.
- ST_AsBinary - Changed: 2.0.0 Inputs to this function can not be unknown -- must be geometry. Constructs such as ST_AsBinary('POI 2)') are no longer valid and you will get an n ST_AsBinary(unknown) is not unique error. Code like that needs to be changed to ST_AsBinary('POINT(1 2)'::geometry);. If that is not possible, then install legacy.sql. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsGML - Changed: 2.0.0 use default named args Return the geometry as a GML version 2 or 3 element.
- ST_AsGeoJSON - Changed: 2.0.0 support default args and named args. Return a geometry or feature in GeoJSON format.
- ST_AsSVG - Changed: 2.0.0 to use default args and support named args Returns SVG path data for a geometry.
- ST_EndPoint - Changed: 2.0.0 no longer works with single geometry MultiLineStrings. In older versions of PostGIS a single-line MultiLineString would work with this function and return the end point. In 2.0.0 it returns NULL like any other MultiLineString. The old behavior was an undocumented feature, but people who assumed they had their data stored as LINESTRING may experience these returning NULL in 2.0.0. Returns the last point of a LineString, CircularLineString, or NURBSCurve.

- ST_GDALDrivers - Changed: 2.0.6, 2.1.3 - by default no drivers are enabled, unless GUC or Environment variable gdal_enabled_drive is set. Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster
- ST_GeomFromText - Changed: 2.0.0 In prior versions of PostGIS ST_GeomFromText('GEOMETRYCOLLECTION(EMPTY)') was allowed. This is now illegal in PostGIS 2.0.0 to better conform with SQL/MM standards. This should now be written as ST_GeomFromText('GEOMETRYCOLLECTION EMPTY') Return a specified ST_Geometry value from Well-Known Text representation (WKT).
- ST_GeometryN - Changed: 2.0.0 Prior versions would return NULL for singular geometries. This was changed to return the geometry for ST_GeometryN(..,1) case. Return an element of a geometry collection.
- ST_IsEmpty - Changed: 2.0.0 In prior versions of PostGIS ST_GeomFromText('GEOMETRYCOLLECTION(EMPTY)') was allowed. This is now illegal in PostGIS 2.0.0 to better conform with SQL/MM standards Tests if a geometry is empty.
- ST_Length - Changed: 2.0.0 Breaking change -- in prior versions applying this to a MULTI/POLYGON of type geography would give you the perimeter of the POLYGON/MULTIPOLYGON. In 2.0.0 this was changed to return 0 to be in line with geometry behavior. Please use ST_Perimeter if you want the perimeter of a polygon Returns the 2D length of a linear geometry.
- ST_LocateAlong - Changed: 2.0.0 in prior versions this used to be called ST_Locate_Along_Measure. Returns the point(s) on a geometry that match a measure value.
- ST_LocateBetween - Changed: 2.0.0 - in prior versions this used to be called ST_Locate_Between_Measures. Returns the portions of a geometry that match a measure range.
- ST_ModEdgeSplit - Changed: 2.0 - In prior versions, this was misnamed ST_ModEdgesSplit Split an edge by creating a new node along an existing edge, modifying the original edge and adding a new edge.
- ST_NumGeometries - Changed: 2.0.0 In prior versions this would return NULL if the geometry was not a collection/MULTI type. 2.0.0+ now returns 1 for single geometries e.g POLYGON, LINESTRING, POINT. Returns the number of elements in a geometry collection.
- ST_NumInteriorRings - Changed: 2.0.0 - in prior versions it would allow passing a MULTIPOLYGON, returning the number of interior rings of first POLYGON. Returns the number of interior rings (holes) of a Polygon.
- ST_PointN - Changed: 2.0.0 no longer works with single geometry multilinestrings. In older versions of PostGIS -- a single line multilinestring would work happily with this function and return the start point. In 2.0.0 it just returns NULL like any other multilinestring. Returns the Nth point in the first LineString or circular LineString in a geometry.
- ST_ScaleX - Changed: 2.0.0. In WKTRaster versions this was called ST_PixelSizeX. Returns the X component of the pixel width in units of coordinate reference system.
- ST_ScaleY - Changed: 2.0.0. In WKTRaster versions this was called ST_PixelSizeY. Returns the Y component of the pixel height in units of coordinate reference system.
- ST_SetScale - Changed: 2.0.0 In WKTRaster versions this was called ST_SetPixelSize. This was changed in 2.0.0. Sets the X and Y size of pixels in units of coordinate reference system. Number units/pixel width/height.
- ST_StartPoint - Changed: 2.0.0 no longer works with single geometry MultiLineStrings. In older versions of PostGIS a single-line MultiLineString would work happily with this function and return the start point. In 2.0.0 it just returns NULL like any other MultiLineString. The old behavior was an undocumented feature, but people who assumed they had their data stored as LINESTRING may experience these returning NULL in 2.0.0. Returns the first point of a LineString, CircularLineString, or NURBSCurve.

13.12.15 PostGIS Functions new or enhanced in 1.5

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 1.5

- && - Availability: 1.5.0 support for geography was introduced. Returns TRUE if A's 2D bounding box intersects B's 2D bounding box.

- PostGIS_LibXML_Version - Availability: 1.5 Returns the version number of the libxml2 library.
- ST_AddMeasure - Availability: 1.5.0 Interpolates measures along a linear geometry.
- ST_AsBinary - Availability: 1.5.0 geography support was introduced. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsGML - Availability: 1.5.0 geography support was introduced. Return the geometry as a GML version 2 or 3 element.
- ST_AsGeoJSON - Availability: 1.5.0 geography support was introduced. Return a geometry or feature in GeoJSON format.
- ST_AsText - Availability: 1.5 - support for geography was introduced. Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.
- ST_Buffer - Availability: 1.5 - ST_Buffer was enhanced to support different endcaps and join types. These are useful for example to convert road linestrings into polygon roads with flat or square edges instead of rounded edges. Thin wrapper for geography was added. Computes a geometry covering all points within a given distance from a geometry.
- ST_ClosestPoint - Availability: 1.5.0 Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.
- ST_CollectionExtract - Availability: 1.5.0 Given a geometry collection, returns a multi-geometry containing only elements of a specified type.
- ST_Covers - Availability: 1.5 - support for geography was introduced. Tests if every point of B lies in A
- ST_DFullyWithin - Availability: 1.5.0 Tests if a geometry is entirely inside a distance of another
- ST_DWithin - Availability: 1.5.0 support for geography was introduced Tests if two geometries are within a given distance
- ST_Distance - Availability: 1.5.0 geography support was introduced in 1.5. Speed improvements for planar to better handle large or many vertex geometries Returns the distance between two geometry or geography values.
- ST_DistanceSphere - Availability: 1.5 - support for other geometry types besides points was introduced. Prior versions only work with points. Returns minimum distance in meters between two lon/lat geometries using a spherical earth model.
- ST_DistanceSpheroid - Availability: 1.5 - support for other geometry types besides points was introduced. Prior versions only work with points. Returns the minimum distance between two lon/lat geometries using a spheroidal earth model.
- ST_DumpPoints - Availability: 1.5.0 Returns a set of geometry_dump rows for the coordinates in a geometry.
- ST_Envelope - Availability: 1.5.0 behavior changed to output double precision instead of float4 Returns a geometry representing the bounding box of a geometry.
- ST_Expand - Availability: 1.5.0 behavior changed to output double precision instead of float4 coordinates. Returns a bounding box expanded from another bounding box or a geometry.
- ST_GMLToSQL - Availability: 1.5, requires libxml2 1.6+ Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GeomFromGML - Availability: 1.5, requires libxml2 1.6+ Takes as input GML representation of geometry and outputs a PostGIS geometry object
- ST_GeomFromKML - Availability: 1.5, requires libxml2 2.6+ Takes as input KML representation of geometry and outputs a PostGIS geometry object
- ST_HausdorffDistance - Availability: 1.5.0 Returns the Hausdorff distance between two geometries.
- ST_Intersection - Availability: 1.5 support for geography data type was introduced. Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Availability: 1.5 support for geography was introduced. Tests if two geometries intersect (they have at least one point in common)
- ST_Length - Availability: 1.5.0 geography support was introduced in 1.5. Returns the 2D length of a linear geometry.
- ST_LongestLine - Availability: 1.5.0 Returns the 2D longest line between two geometries.

- ST_MakeEnvelope - Availability: 1.5 Creates a rectangular Polygon from minimum and maximum coordinates.
- ST_MaxDistance - Availability: 1.5.0 Returns the 2D largest distance between two geometries in projected units.
- ST_ShortestLine - Availability: 1.5.0 Returns the 2D shortest line between two geometries
- ~= - Availability: 1.5.0 changed behavior Returns TRUE if A's bounding box is the same as B's.

13.12.16 PostGIS Functions new or enhanced in 1.4

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 1.4

- Populate_Geometry_Columns - Availability: 1.4.0 Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.
- ST_Collect - Availability: 1.4.0 - ST_Collect(geomarray) was introduced. ST_Collect was enhanced to handle more geometries faster. Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_ContainsProperly - Availability: 1.4.0 Tests if every point of B lies in the interior of A
- ST_GeoHash - Availability: 1.4.0 Return a GeoHash representation of the geometry.
- ST_IsValidReason - Availability: 1.4 Returns text stating if a geometry is valid, or a reason for invalidity.
- ST_LineCrossingDirection - Availability: 1.4 Returns a number indicating the crossing behavior of two LineStrings
- ST_LocateBetweenElevations - Availability: 1.4.0 Returns the portions of a geometry that lie in an elevation (Z) range.
- ST_MakeLine - Availability: 1.4.0 - ST_MakeLine(geomarray) was introduced. ST_MakeLine aggregate functions was enhanced to handle more points faster. Creates a LineString from Point, MultiPoint, or LineString geometries.
- ST_MinimumBoundingCircle - Availability: 1.4.0 Returns the smallest circle polygon that contains a geometry.
- ST_Union - Availability: 1.4.0 - ST_Union was enhanced. ST_Union(geomarray) was introduced and also faster aggregate collection in PostgreSQL. Computes a geometry representing the point-set union of the input geometries.

13.12.17 PostGIS Functions new or enhanced in 1.3

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 1.3

- ST_AsGML - Availability: 1.3.2 Return the geometry as a GML version 2 or 3 element.
- ST_AsGeoJSON - Availability: 1.3.4 Return a geometry or feature in GeoJSON format.
- ST_CurveToLine - Availability: 1.3.0 Converts a geometry containing curves to a linear geometry.
- ST_LineToCurve - Availability: 1.3.0 Converts a linear geometry to a curved geometry.
- ST_SimplifyPreserveTopology - Availability: 1.3.3 Returns a simplified and valid representation of a geometry, using the Douglas-Peucker algorithm.

Chapter 14

Reporting Problems

14.1 Reporting Software Bugs

Reporting bugs effectively is a fundamental way to help PostGIS development. The most effective bug report is that enabling PostGIS developers to reproduce it, so it would ideally contain a script triggering it and every information regarding the environment in which it was detected. Good enough info can be extracted running `SELECT postgis_full_version()` [for PostGIS] and `SELECT version()` [for postgresql].

If you aren't using the latest release, it's worth taking a look at its release changelog first, to find out if your bug has already been fixed.

Using the PostGIS bug tracker will ensure your reports are not discarded, and will keep you informed on its handling process. Before reporting a new bug please query the database to see if it is a known one, and if it is please add any new information you have about it.

You might want to read Simon Tatham's paper about How to Report Bugs Effectively before filing a new report.

14.2 Reporting Documentation Issues

The documentation should accurately reflect the features and behavior of the software. If it doesn't, it could be because of a software bug or because the documentation is in error or deficient.

Documentation issues can also be reported to the PostGIS bug tracker.

If your revision is trivial, just describe it in a new bug tracker issue, being specific about its location in the documentation.

If your changes are more extensive, a patch is definitely preferred. This is a four step process on Unix (assuming you already have git installed):

Before editing or building the manual, read the repository's documentation development guide. It covers the executable example, generated figure, validation, HTML, and PDF workflows used by the manual.

1. Clone the PostGIS' git repository. On Unix, type:

   git clone https://git.osgeo.org/gitea/postgis/postgis.git

   This will be stored in the directory postgis

2. Make your changes to the documentation with your favorite text editor. On Unix, type (for example):

   vim doc/postgis.xml

   Note that the documentation is written in DocBook XML rather than HTML, so if you are not familiar with it please follow the example of the rest of the documentation.

3. Make a patch file containing the differences from the master copy of the documentation. On Unix, type:
   git diff doc/postgis.xml > doc.patch
4. Attach the patch to a new issue in bug tracker.

Chapter 15

Credits and Acknowledgments

PostGIS is built and sustained by a worldwide community. This chapter records the people, organizations, funders, and upstream projects that have contributed code, documentation, infrastructure, testing, support, and funding.

15.1 Project Steering Committee

The PostGIS Project Steering Committee (PSC) coordinates the general direction, release cycles, documentation, and outreach efforts for the PostGIS project. In addition the PSC provides general user support, accepts and approves patches from the general PostGIS community and votes on miscellaneous issues involving PostGIS such as developer commit access, new PSC members or significant API changes.

Regina Obe CI and website maintenance, Windows production and experimental builds, documentation, alignment of PostGIS with PostgreSQL releases, X3D support, management functions.

Darafei Praliaskouski Index improvements, bug fixing and geometry/geography function improvements, SFCGAL, raster, GitHub curation, and ci maintenance.

Paul Ramsey (Chair) Co-founder of PostGIS project. General bug fixing, geography support, geography and geometry index support (2D, 3D, nD index and anything spatial index), underlying geometry internal structures, GEOS functionality integration and alignment with GEOS releases, alignment of PostGIS with PostgreSQL releases, loader/dumper, and Shapefile GUI loader.

Sandro Santilli Bug fixes and maintenance, ci maintenance, git mirror management, management functions, integration of new GEOS functionality and alignment with GEOS releases, topology support, and raster framework and low level API functions.

15.2 Core Contributors Present

Nicklas Avén Distance function enhancements (including 3D distance and relationship functions) and additions, Tiny WKB (TWKB) output format and general user support

Loïc Bartoletti SFCGAL enhancements and maintenance and ci support

Dan Baston Geometry clustering function additions, other geometry algorithm enhancements, GEOS enhancements and general user support

Martin Davis GEOS enhancements and documentation

Björn Harrtell MapBox Vector Tile, GeoBuf, and Flatgeobuf functions. Gitea testing and GitLab experimentation.

Aliaksandr Kalenik Geometry Processing, PostgreSQL gist, general bug fixing

Laurențiu Nicola Bug fixes, maintenance, stability, and performance

15.3 Core Contributors Past

Raúl Marín Rodríguez Prior PSC Member, MVT support, Bug fixing, Performance and stability improvements, GitHub curation, alignment of PostGIS with PostgreSQL releases

Bborie Park Prior PSC Member. Raster development, integration with GDAL, raster loader, user support, general bug fixing, testing on various OS (Slackware, Mac, Windows, and more)

Mark Cave-Ayland Prior PSC Member. Coordinated bug fixing and maintenance effort, spatial index selectivity and binding, loader/dumper, and Shapefile GUI Loader, integration of new and new function enhancements.

Jorge Arévalo Raster development, GDAL driver support, loader

Olivier Courtin (Emeritus) Input/output XML (KML,GML)/GeoJSON functions, 3D support and bug fixes.

Chris Hodgson Prior PSC Member. General development, site and buildbot maintenance, OSGeo incubation management

Mateusz Loskot CMake support for PostGIS, built original raster loader in python and low level raster API functions

Kevin Neufeld Prior PSC Member. Documentation and documentation support tools, buildbot maintenance, advanced user support on PostGIS newsgroup, and PostGIS maintenance function enhancements.

Dave Blasby The original developer/Co-founder of PostGIS. Dave wrote the server side objects, index bindings, and many of the server side analytical functions.

Jeff Lounsbury Original development of the Shapefile loader/dumper.

Mark Leslie Ongoing maintenance and development of core functions. Enhanced curve support. Shapefile GUI loader.

Pierre Racine Architect of PostGIS raster implementation. Raster overall architecture, prototyping, programming support

David Zwarg Raster development (mostly map algebra analytic functions)

15.4 PostGIS Raster Origins

PostGIS raster discussions date to 2002 and included early PGCHIP experiments. Xing Lin developed the **geotiff2pgraster** PGRaster prototype during Google Summer of Code 2007. The WKT Raster project began in 2008, Jorge Arévalo developed the first GDAL driver prototype during Google Summer of Code 2009, and the 2010 FOSS4G code sprint agreed to make WKT Raster the PostGIS raster type. Raster support shipped as part of PostGIS 2.0 in 2012.

The early implementation team included Pierre Racine, Mateusz Loskot, Sandro Santilli, Jorge Arévalo, Bborie Park, David Zwarg, and Regina Obe. Their work established the raster type, loader, GDAL integration, raster/vector operations, map algebra, documentation, and testing that became the current raster subsystem.

Other recorded contributions include Jeff Adams's early sprint work, Qing Liu's raster distance analysis, Maurício Carvalho Mathias de Paulo's QGIS raster plugin and related paper with Lubia Vinhas, and Ignacio Brodin's gvSIG raster plugin and presentation with Jorge Sanz. Bryce L. Nordgren created early raster coverage and seamless-architecture diagrams. Pierre Racine wrote the original raster/vector tutorial and created its walkthrough and early API diagrams.

The historical project planning record credits Steve Cumming and Université Laval with funding Pierre Racine's work from June 2008 through July 2012; Martin Daly and Cadcorp with funding about six months of Mateusz Loskot's work through June 2009; Tyler Erickson and Michigan Tech Research Institute with a $1,200 USD contribution; Miguel Lizondo and Deimos Space with support for about two years of Jorge Arévalo's work through June 2011; and the Center for Vectorborne Diseases at UC Davis with funding Bborie Park's work from March 2010 through July 2012.

15.5 Other Contributors

Individual Contributors

@amwps290 (GitHub user)
@kannes (GitHub user: Hannes)
@sunhm89 (GitHub user)
@wewindy (GitHub user: Chen)
Adrien Berchet
Alex Bodnaru
Alex Mayrhofer
Andrea Peri
Andreas Forø Tollefsen
Andreas Neumann
Andreas Schild
Andrew Gierth
Andrew Harvey
Anne Ghisla
Antoine Bajolet
Ariel Mashraki
Arno Renevier
Arthur Bazin
Arthur Lesuisse
Artur Zakirov
Ayo Adesugba
Barbara Phillipot
Baris Ergun
Bas Couwenberg
Ben Jubb
Benjamin Morel
Bernd Reiß
Bernhard Reiter
Bill Mill
Björn Esser
Brian Hamlin
Bruce Rindahl
Bruno Wolff III
Bryce L. Nordgren
Carl Anderson
Carlos López Quintanilla
Charles Karney
Charlie Savage
Chris Mayo
Christian Schroeder
Christoph Berg
Christoph Moench-Tegeder
Dan Faudemer
Dan Levy
Dane Springmeyer
Daniel Nylander
Dapeng Wang
Daryl Herzmann
Dave Fuhry
David G. Johnston
David Garnier
David Skea
David Techer
Dennis Tighe
Denys Kovshun
Devrim Gündüz
Dian M Fay
Dmitry Vasilyev
Edouard Choinière
Eduin Carrillo
Egor Ivkov
Eric Ridge
Esteban Zimanyi
Eugene Antimirov
Florian Weimer
François Bonzon
Frank Warmerdam
George Silva
Gerald Fenoy
Gino Lucrezi
Giovanni Zezza
Grant Hutchins
Greg Troxel
Guillaume Lelarge
Giuseppe Broccolo
Han Wang
Hannes Janetzek
Hans Lemuet
Haribabu Kommi
Harrissou Sant-anna
Havard Tveite
Ignacio Brodin
IIDA Tetsushi
Ingvild Nystuen
Jackie Leng
James Addison
James Marca
Jan Katins
Jan Tojnar
Jason Smith
Jean Felder
Jeff Adams
Jelte Fennema
Jim Jones
Joe Conway
Jonne Savolainen
Jorge Gustavo Rocha
Jorge Sanz
Jose Carlos Martinez Llari
Jostein Leira
Jörg Habenicht
Juan Sebastian Jimenez Aldana
Juergen E. Fischer
Julien Monticolo
Julien Rouhaud
Kashif Rasul
Klaus Foerster
Kris Jurka
Kristian Thy
Lars Aksel Opsahl
Laurențiu Nicola
Laurenz Albe
Lars Roessiger
Lauri Kajan
Leo Hsu
Loic Dachary
Louis Descoteaux
Luca Delucchi
Luca S. Percich
Lucas C. Villa Real
Ludovic Hirlimann
Lubia Vinhas
Maksim Korotkov
Manuel Morillo
Maria Arias de Reyna
Marc Ducobu
Mark Sondheim
Markus Schaber
Maurício Carvalho Mathias de Paulo
Maxim Kochetkov
Maxime Guillaud
Maxime van Noppen
Maxime Schoemans
Megan Ma
Mehmet Ince
Michael Fuhr
Michal Páleník
Mike Toews
Mitchell Henke
Nathan Wagner
Nathaniel Clay
Nicolas Roelandt
Nikita Shulga
Nikolai Berkoff
Norman Barker
Norman Vine
Patricia Tozer
Paul Norman
Paulo Cesar Coronado
Pavel Seleznev
Pawel Ostrowski
Peter Nguyen
Pierre Cardinal
Qing Liu
Rafal Magda
Ralph Mason
Rémi Cura
Richard Greenwood
Robert Coup
Rodrigo Smarzaro
Roger Crew
Ron Mayer
Sam Peters
Sarath Kumar
Sergei Shoulbakov
Sergey Fedoseev
Shinichi Sugiyama
Shoaib Burq
Silvio Grosso
Simon Greener
Stefan Corneliu Petrea
Steffen Macke
Stepan Kuzmin
Stephen Frost
Stephen Woodbridge
Steven Ottens
Svyatoslav Bogdanov
Talha Rizwan
Temuri Doghonadze
Teramoto Ikuhiro
Thomas Muguet
Thomas Petazzoni
Tobias Bussmann
Tom Glancy
Tom van Tilburg
Victor Collod
Vincent Bre
Vincent Mora
Vincent Picavet
Volf Tomáš
Xing Lin
Yoichi Kayama

Corporate Sponsors These are corporate entities that have contributed developer time, hosting, or direct monetary funding to the PostGIS project. In alphabetical order:

- Aiven
- Arrival 3D
- Associazione Italiana per l'Informazione Geografica Libera (GFOSS.it)
- AusVet
- Avencia
- Azavea
- Boundless
- Cadcorp
- Camptocamp
- Carto
- Crunchy Data
- City of Boston (DND)
- City of Helsinki
- Clever Elephant Solutions
- Cooperativa Alveo
- Deimos Space
- Faunalia
- Geographic Data BC
- Google
- HighGo
- Hunter Systems Group
- IITM Pravartak Security Team
- INIA-CSIC
- ISciences, LLC
- Jirotech (formerly LISAsoft)
- Koordinates
- Kontur
- Lidwala Consulting Engineers
- Logical Tracking & Tracing International AG
- Maponics
- Maumap
- Michigan Tech Research Institute
- Natural Resources Canada
- Norwegian Forest and Landscape Institute
- Norwegian Institute of Bioeconomy Research (NIBIO)
- OSGeo
- Oslandia
- Palantir Technologies
- Paragon Corporation
- PlanetScale
- Postgres Pro

- R3 GIS
- Refractions Research
- Regione Toscana - SITA
- Safe Software
- Sirius Corporation plc
- Stadt Uster
- UC Davis Center for Vectorborne Diseases
- Université Laval
- U.S. Census Bureau
- U.S. Department of State (HIU)
- Zonar Systems

Crowd Funding Campaigns Crowd funding campaigns are campaigns we run to get badly wanted features funded that can service a large number of people. Each campaign is specifically focused on a particular feature or set of features. Each sponsor chips in a small fraction of the needed funding and with enough people/organizations contributing, we have the funds to pay for the work that will help many. If you have an idea for a feature you think many others would be willing to co-fund, please post to the PostGIS newsgroup your thoughts and together we can make it happen.

PostGIS has experimented with community fundraising for features in the past. We used PledgeBank and we got two successful campaigns out of it.

postgistopology - 10 plus sponsors each contributed $250 USD to build toTopoGeometry function and beef up topology support in 2.0.0. It happened.

postgis64windows - 20 someodd sponsors each contributed $100 USD to pay for the work needed to work out PostGIS 64-bit issues on windows. It happened.

External Libraries and Platforms PostgreSQL, the extensible database system on which PostGIS is built. PostgreSQL provides the SQL engine, transactions, query planner, extension framework, and index access methods used by PostGIS.

The GEOS geometry operations library.

The PROJ coordinate transformation and geodesic calculation library.

The GDAL Geospatial Data Abstraction Library used to power much of the raster functionality introduced in PostGIS 2. In kind, improvements needed in GDAL to support PostGIS are contributed back to the GDAL project.

LibXML2 for XML-based spatial formats, and JSON-C for GeoJSON input.

The GNU Multiple Precision Arithmetic Library (GMP) for precision-sensitive topology operations.

protobuf-c for Mapbox Vector Tile and Geobuf output.

The iconv API, supplied by the C library or GNU libiconv, for character encoding conversion in the shapefile loader.

SFCGAL for advanced 2D and 3D functions, together with the CGAL algorithms that SFCGAL exposes.

Bundled Libraries and Source Code PostGIS ships selected source from FlatGeobuf and FlatBuffers to support FlatGeobuf input and output.

Wagyu and Mapbox Geometry provide polygon clipping and validation for Mapbox Vector Tile output.

Ryu provides round-trip-safe floating-point formatting, and uthash provides in-process hash tables.

The shapefile loader and dumper include code derived from ShapeLib, originally by Frank Warmerdam and Even Rouault. They also include public-domain `getopt` code from AT&T.

`liblwgeom/lookup3.c` contains the public-domain `lookup3` hash function by Bob Jenkins.

Appendix A

Release Notes

A.1 PostGIS 3.7.0

This version requires GEOS 3.10+, PostgreSQL 14-19beta2, Proj 6.1+, libgmp. To take advantage of all features postgis extension features, GEOS 3.15+ is needed. To take advantage of all postgis_sfcgal extension features SFCGAL 2.3+ is needed.

A.1.1 Breaking Changes

#5688, [topology] topology building functions now interpret tolerance 0 (old default) to really mean 0 and want -1 (new default) to mean "use topology precision" (Sandro Santilli)

#6024, Remove support for PostgreSQL 12 and 13 (Regina Obe)

#6040, ST_GeoHash round trip through ST_GeomFromGeoHash (Paul Ramsey)

#6053, [address_standardizer] Extension removed and moved to https://github.com/postgis/address_standardizer (Paul Ramsey)

#6052, [tiger_geocoder] Extension removed and moved to https://git.osgeo.org/postgis/postgis_tiger_geocoder (Regina Obe)

#6079, Remove support for GEOS < 3.10 (Sandro Santilli)

#1793, Remove ST_MemCollect aggregate (Darafei Praliaskouski)

GH-848, CG_AlphaShape now returns a MultiPolygon (Jean Felder)

A.1.2 New Features

#1124, #2935, #3033, #4658, #4659, [loader] Rework loader arguments: shp2pgsql can create UNLOGGED tables and choose the feature id column name, shp2pgsql --drop-table can emit DROP TABLE before prepare output, raster2pgsql/shp2pgsql expose loader actions as long options, both loaders accept long aliases for existing options, and

-if-not-exists makes creation actions idempotent (Darafei Praliaskouski)

#4208, Add single-geometry variants of ST_MaxDistance and ST_LongestLine (Darafei Praliaskouski)

[topology] FindVertexSegmentPairsBelowDistance function (Sandro Santilli)

ST_CoverageEdges, returns MultiLinestring of distinct shared edges in polygonal coverage (Paul Ramsey)

ST_MinimumSpanningTree, window function to calculate MST (Paul Ramsey)

#5993, [topology] Add max_edges parameter to TopoGeo_AddLinestring (Sandro Santilli)

#6001, support MultiLineString in ST_MakeLine (Paul Ramsey)

#2858, ST_MMin and ST_MMax (Paul Ramsey)

#5941, [raster] Support GDT_Float16 pixel type (Darafei Praliaskouski)

#5992, Optimize GiST index for repeated edge subdivision in topology (Darafei Praliaskouski)

#5702, Allow the compiler to detect the parallelism -flto=auto (Darafei Praliaskouski)

#4798, ST_AsGeoJSON warns about duplicate property keys (Darafei Praliaskouski)

#5950, Document POSTGIS_REGRESS_DB_OWNER for sandboxed regression roles (Darafei Praliaskouski)

#4332, Clarify the scope of several Function Reference categories (Darafei Praliaskouski)

#4222, [raster] ST_DumpAsPolygons honours PostgreSQL interrupts (Darafei Praliaskouski)

#5109, Document the meaning of topology.next_left_edge and topology.next_right_edge links (Darafei Praliaskouski)

#5889, [topology] Include representative locations in topology build errors (Darafei Praliaskouski)

#2614, Use GEOSPreparedDistance and caching to accelerate ST_DWithin (Paul Ramsey)

#6022, Document ST_AsMVTGeom tile-coordinate simplification (Darafei Praliaskouski)

GH-839, ST_Multi support for TIN and surfaces (Loïc Bartoletti)

#4398, Add ST_CatmullSmoothing smoothes but retains original vertices (Paul Ramsey)

#4560, ST_3DInterpolatePoint for M interpolation from XYZ inputs (Paul Ramsey)

#1291, ST_MakePolygon support for curved rings (Darafei Praliaskouski)

GT-270, Add NURBSCurve (Loïc Bartoletti)

#2863, Add ST_XSize, ST_YSize, ST_ZSize, and ST_MSize dimension helpers (Darafei Praliaskouski)

[fuzzers] Centralize OSS-Fuzz build/dependency logic in PostGIS scripts, prefer requested CXX over pg_config --cc for internal C++ checks, and make fuzzer linking/runtime dependency packaging portable (Darafei Praliaskouski)

A.1.3 Enhancements

#2045, pgsql2shp query dumps no longer require temporary table privileges (Darafei Praliaskouski)

#1577, [loader] Report unsupported .zip archive input before trying to open shapefile sidecar files (Darafei Praliaskouski)

#3158, Move geometry and geography typmod bit helpers out of the public liblwgeom header (Darafei Praliaskouski)

#4678, [raster] Move st_sum4ma neighborhood callback to C (Darafei Praliaskouski)

#2116, [raster] Add ST_Value nearest-neighbor boundary options (Darafei Praliaskouski)

#2804, #4315, [raster] Support two-argument ST_MapAlgebra callbacks and pass callback call data as actual arguments (Darafei Praliaskouski)

#2898, Document SQL function cost tiers for contributors

#6062, [topology] Stop using recursive snapping, for improved robustness (Sandro Santilli)

#6065, Improved winding order computation robustness for rings having collapsed elements (Sandro Santilli)

#3743, #4385, [raster loader] Document raster2pgsql -s FROM_SRID:SRID reprojection support and clarify index creation timing for append loads (Darafei Praliaskouski)

#2137, [raster] Reuse matching source raster overviews in raster2pgsql -l (Darafei Praliaskouski)

#4749, Use point-in-polygon predicate fast paths for point-only GeometryCollections (Darafei Praliaskouski)

#5532, Validate the manual against DocBook XMLSchema in check-xml (Darafei Praliaskouski)

#6087, Add a repository-owned CI status checker and static dashboard generator (Darafei Praliaskouski)

#2807, [raster] Preserve missing NODATA values in ST_MapAlgebra outputs (Darafei Praliaskouski)

#2832, [raster] Avoid padding single-tile raster2pgsql overviews when padding is not requested (Darafei Praliaskouski)

#2623, Generate a pgsql2shp GID field when exporting geometry-only shapefiles (Darafei Praliaskouski)

#2850, Preserve pgsql2shp numeric precision and scale in DBF metadata (Darafei Praliaskouski)

#2386, Add raster2pgsql man page (Darafei Praliaskouski)

Build PostgreSQL extension modules with `-fno-semantic-interposition` when available so LTO can optimize same-DSO calls to exported PostGIS entry points directly instead of routing through the module PLT; in a local PostgreSQL 18 `pgbench` microbenchmark over 100k four-point groups this reduced `ST_Collect` aggregate latency by about 4% (Darafei Praliaskouski)

#5973, Install PostGIS utilities under `pg_config --bindir` by default and document `pg_config`-derived install paths more accurately (Darafei Praliaskouski)

#2808, Clarify postgis_full_version() output when installed component scripts are newer than core scripts (Darafei Praliaskouski)

#5593, Docs: render manual images with GraphicsMagick without temporary files, enabling parallel builds, keeping ImageMagick fallbacks working, and documenting the generator workflow for contributors (Darafei Praliaskouski)

#5645, Docs: keep code operators ("=>") intact in translated manuals by enforcing verbatim CSS (Darafei Praliaskouski)

#1408, Docs: flatten transparent manual images onto white backgrounds for PDF builds while keeping HTML images transparent (Darafei Praliaskouski)

#2838, Emit valid X3D Coordinate nodes for 2D polygon output (Darafei Praliaskouski)

GH-899, [raster] Honor PostgreSQL interrupts in long-running GDAL progress callbacks (Darafei Praliaskouski)

#2583, Preserve Z/ZM dimensional qualifiers in ST_AsEWKT output for TIN (Darafei Praliaskouski)