PostgreSQL 7.0

Open Source Relational Database

Database Administration and Tuning

White Paper

Great Bridge, LLC
October 2000

www.greatbridge.com

Table of Contents

1 Introduction

Great Bridge, LLC was created in May 2000, for the purpose of professionally marketing and supporting open source software solutions based on PostgreSQL. For an introduction to PostgreSQL, the world's most advanced open source database, please see our first white paper, "PostgreSQL 7.0 Open Source Relational Database." The company's mission is to empower today's e-business builders with an enterprise-class database and tools, at a fraction of the cost of proprietary commercial alternatives.

Great Bridge is privately held, well-funded by Landmark Communications, Inc., a media company based in Norfolk, Virginia that also owns The Weather Channel, weather.com, and national and international newspapers, broadcasting, electronic publishing, and specialty media outlets.

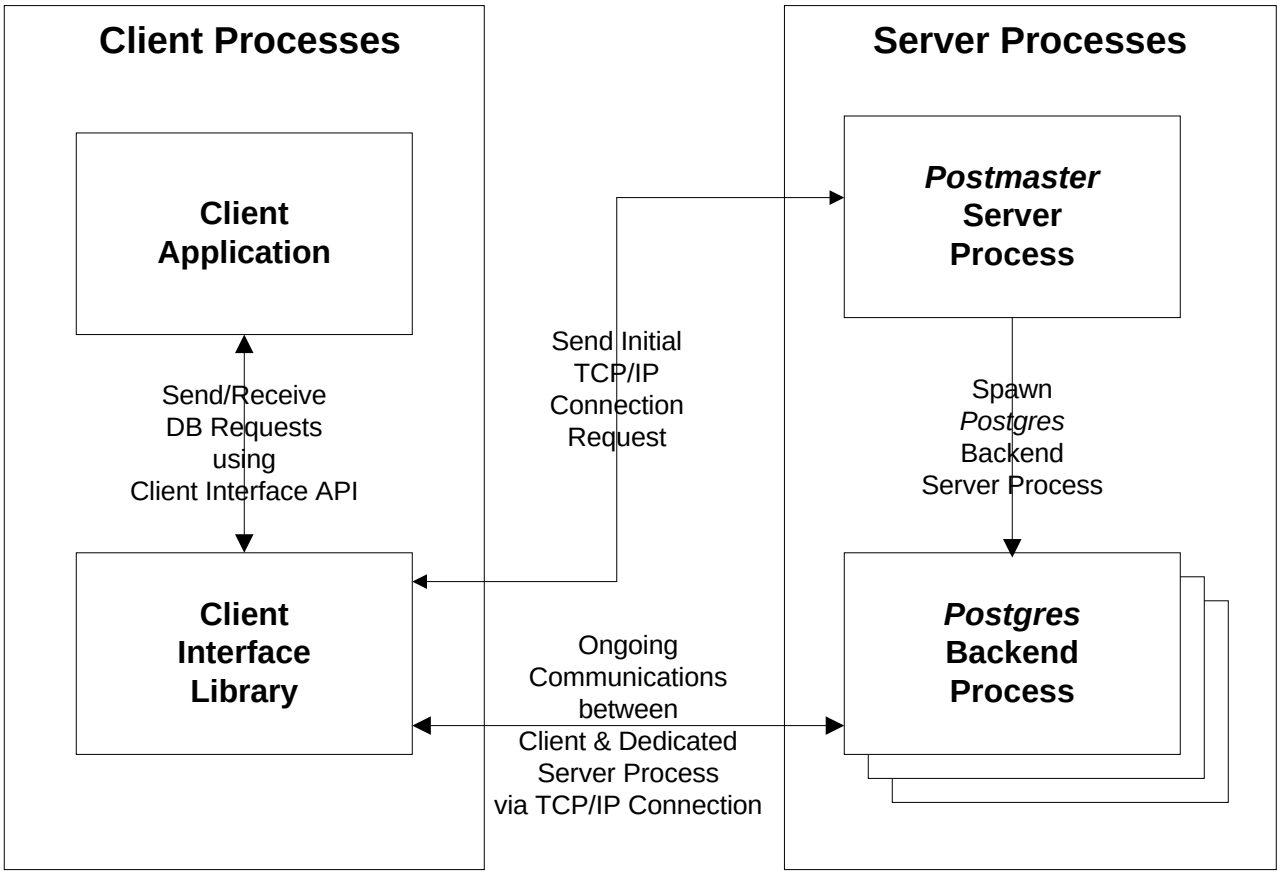
The latest release of the PostgreSQL software is version 7.02, which offers significant transaction throughput, a sophisticated optimizer to streamline complex queries, commercial-grade SQL support, and flexibility for building full-service e-business solutions.

The purpose of this white paper is to provide an overview of the PostgreSQL Database Administration and Performance Tuning features that are available within the product. Specific topics that will be discussed include the PostgreSQL database architecture, unique features available within PostgreSQL, database administrative functions, performance tuning capabilities, and helpful database utilities for managing PostgreSQL environments. The overall goal in presenting these topics is to provide Database Administrators and ITS Managers a better understanding of the advanced technical features available within PostgreSQL.

2 Overall Database Architecture

The PostgreSQL database environment is comprised of three major components, which are outlined in the Database System Model shown below. There can be several PostgreSQL databases running on a single server site. Each server will have one master process called the *Postmaster*, which is responsible for managing all incoming database requests and establishing database connectivity between front-end clients and one or more backend database processes. Once the *Postmaster* confirms the client process has the appropriate credentials to access the requested database(s) running on the server, the *Postmaster* will spawn a server process called the *Postgres Backend Process.* Each client connection will have a dedicated *Postgres* process that will run in the background, servicing database requests submitted by the client.

PostgreSQL Database System Model

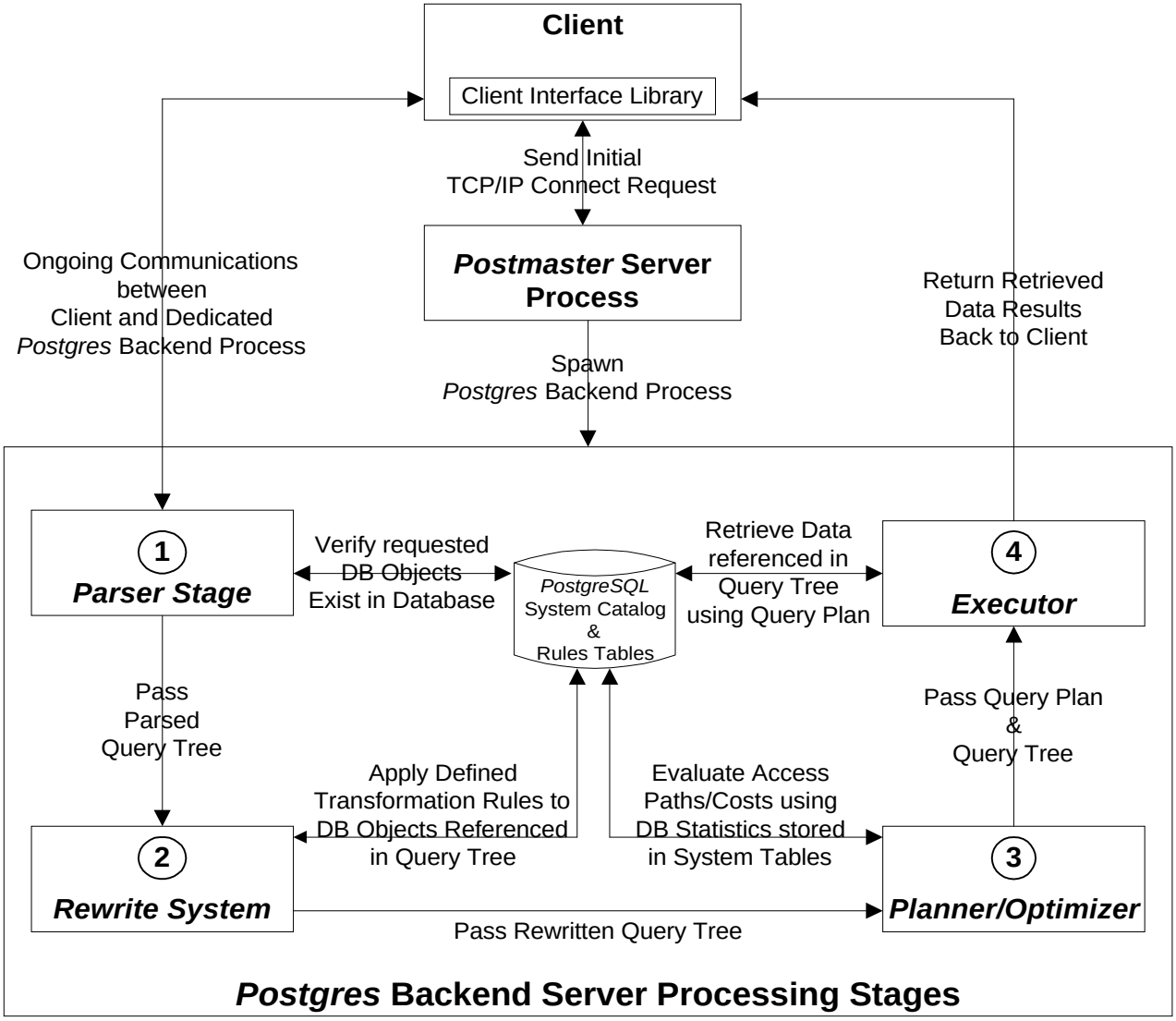


Most client applications communicate with the backend PostgreSQL server using a Client Interface Library, which provides a set of standard Application Programming Interfaces (API), for communicating with PostgreSQL. Although client applications can be written to communicate directly with PostgreSQL using the TCP/IP protocol, the Client Interface Library provides a more convenient mechanism for managing database connections. Several Client Interface Libraries are distributed with the base PostgreSQL software. The most commonly used Client Interface Library is called *libpq*. Client applications that support the Open DataBase Connectivity (ODBC) interface standard can access PostgreSQL using various PostgreSQL ODBC Drivers, which are available within the open source community.

The overall architecture for PostgreSQL is very flexible, allowing database administrators to configure multiple databases with different application topologies to run within the same server environment. In addition, each PostgreSQL installation provides a standard set of configuration files for a generic database called *template1*, which can be modified to reflect site specific standards across all database implementations at a given server site. These environmental standards can be applied to all new databases by using the PostgreSQL *createdb* Utility, which uses the *template1* configuration files to establish a new database on the server.

The *Postgres* backend server process, which services client requests, is comprised of several subcomponents that communicate with each other using shared memory and resources that are available within the server environment. Each client request goes through a series of processing stages to retrieve and deliver the requested data from the PostgreSQL database. The diagram labeled *Postgres* Server Subcomponents illustrates how client requests are serviced through these various processing stages.

Postgres Server Subcomponents



A connection from a client application to a PostgreSQL database server is initially established via the *Postmaster*, which listens for network requests on a specific server port. The default PostgreSQL port setting is 5432. The *Postmaster* validates whether the requesting client is authorized to access the *Postgres* Server and/or the specified database(s), using a secure configuration file called *pg_hba.conf*. If the client is authorized for access, then a dedicated *Postgres* backend server process is spawned, and the client is passed onto the *Postgres* process to be serviced. This initial connection between the client and the *Postgres* backend process only needs to be done once by the *Postmaster* server process. After database connectivity is established, the *Postgres* backend process handles all subsequent communications between the client and the server.

1. The first stage of servicing a client request is handled by the *Postgres Parser*, which checks the client's SQL request to ensure it is using the correct syntax, and builds an internal structure called a query tree. The query tree is a data structure used by the various *Postgres* subcomponents for parsing all of the SQL statements that are passed to the server from the client. As the query tree is built, the *Parser* validates whether the specified database objects and attributes exist within the PostgreSQL database. If any errors are encountered or there are unresolved database references, an error message will be returned to the client and processing will be aborted.
2. If the *Postgres Parser* successfully builds a query tree structure, processing continues onto the *Query Rewrite System*. Within PostgreSQL, there is a powerful *Rule System* that enables processing and transformation rules to be defined within the database engine. The PostgreSQL *Rule System* will be explained in more detail later in this chapter, but one of its primary roles is to resolve queries that access database Views. The *Rewrite System* uses the PostgreSQL *Rule System* to determine if any transformation rules have been defined for any of the objects referenced in the query tree. If so, the *Rewrite System* will apply the transformation rules defined within the PostgreSQL *Rule System* by rewriting the query tree.

   For example if a database View is referenced in a query tree, the Rewrite System will rewrite the query to retrieve the underlying base tables associated with the View definition. All Views are maintained within the PostgreSQL *Rule System*. Once all database rules have been applied to the query tree, the results are passed onto the *Postgres Planner/Optimizer*.

3. The next stage is the *Planner/Optimizer*, which utilizes a sophisticated algorithm to determine the most efficient method for retrieving the data that is referenced in the passed query tree. Access paths for retrieving the data, and their associated resource cost estimates are evaluated by the *Postgres Planner/Optimizer*. The *Planner/Optimizer* selects the most efficient execution plan and builds a query plan, based upon the evaluated resource costs. Both the query plan and the query tree are then passed to the *Executor* to retrieve the data from the database.

4. The *Executor* takes the query plan and query tree passed by the *Postgres Planner/Optimizer* and starts retrieving the data as specified in these internal structures. The *Executor* processes all data sorting and table joins in this processing stage. When all of the specified data is retrieved, the results are returned to the client.

Ongoing communications between the Client and the *Postgres* Server process continue until all Client requests are serviced, and the Client terminates the database session.

Unique Database Features Available within PostgreSQL

There are several advanced features within PostgreSQL that distinguish it from commercially available database products. PostgreSQL supports both relational and object-oriented database models. Although some database vendors claim their products support both models, the reality is most of these databases were originally designed solely for relational use. As these databases matured, their architectures were extended to support object-oriented concepts to meet market demands. What has resulted from such implementations, are hybrids of both relational and objected-oriented models that have restrictions. Fortunately, PostgreSQL does not have any of these restrictions, since it was architecturally designed to support both models from its inception.

Objected-oriented features such as object class inheritance and non-atomic data values (i.e. arrays of base types and set-valued attributes) can be defined and referenced within PostgreSQL. In addition, all PostgreSQL functions and procedures that are embedded within the core database engine are extensible. These extensions allow developers to customize objects to support business-specific requirements. This is possible because PostgreSQL operates in a catalog-driven environment, where all of its internal database objects are stored within its System Catalog. Besides storing basic information about databases, tables, columns, and indices, the Postgres System Catalog also stores metadata about supported class types, functions, access methods, languages, and so on. Thus, the basic concepts of object-oriented programming are fully leveraged within the PostgreSQL architecture.

One example of the type of extensibility PostgreSQL provides within its base software is a mechanism for incorporating user-written code into the core database engine through dynamic loading. In other words, the user can specify an object code file (e.g., a compiled object file or shared library) that implements a new type or function, allowing the *Postgres* Server to dynamically link to it as required. This is particularly useful for development environments where rapid prototyping of applications is needed, or for market-driven e-commerce businesses that demand seamless integration of disparate legacy systems. Thus, PostgreSQL provides a facility for developers to build middleware components that can be referenced from within the database at run-time.

Another powerful feature is the PostgreSQL *Rule System*. As previously discussed, the PostgreSQL *Rule System* is used to resolve database View definitions during the *Query Rewrite* stage of processing a SQL request. Since the *Rule System* is accessed between the *Parser* and *Planner/*Optimizer processing stages, it can be utilized to apply business specific rules or complex operations on data prior to executing the target SQL statement in the *Executor* phase of processing. There are subtle differences between using standard database triggers for such operations, and applying this functionality within PostgreSQL's *Rule System*. However, there are situations where using the *Rule System* can actually improve database performance versus using triggers (See Chapter 8 of the PostgreSQL Programmer's Guide for more details at www.postgresql.org/docs/programmer/rules.htm).

The PostgreSQL database also provides other techniques for enhancing overall performance like its Multi-Version Concurrency Control (MVCC) feature. MVCC is an advanced transactional model that is superior to most commercial database systems. PostgreSQL maintains data consistency using a multi-version model versus using page or row-level locking. In a traditional database system, any row that is modified within a transaction is locked until the transaction is committed or rolled back. This prevents users from reading data that has been altered, but not committed, by other database users. Using PostgreSQL's MVCC model, every database user can read and write consistent data without imposing database locks. Every database user is given a snapshot or version of the data that existed at the time the database session was initiated. Thus, database readers never block database writers from performing their tasks and vice-versa.

Database Administrators can utilize these unique features within PostgreSQL to solve many complex business problems. These features enable applications to be deployed within PostgreSQL that would normally require extensive programming in other database environments. In addition, the open source software community provides a wealth of database tools and applications that can be implemented with the PostgreSQL database to extend its capabilities. Harnessing these capabilities with the power of the PostgreSQL database, Great Bridge intends to provide value-added software and support services for PostgreSQL customers. Great Bridge is committed to providing the level of support for PostgreSQL that Database Administrators and ITS Managers within the industry expect from their database management systems.

3 Physical Database Environment

The PostgreSQL database environment provides a suite of tools for creating and maintaining PostgreSQL databases. Many of these tools are provided as standard utilities within the PostgreSQL environment. The open source community also provides a wealth of software resources that supplement the existing PostgreSQL functionality. These resources come in the form of software tools and helpful technical notes on how to perform specific tasks within PostgreSQL. As these resources mature, Great Bridge intends to incorporate them into the base PostgreSQL product offerings, along with providing technical support for PostgreSQL clients. This section provides an overview of the PostgreSQL database environment, highlighting some of the tools that are available for maintaining its physical infrastructure.

Creating a PostgreSQL Database

When the PostgreSQL software is installed for the first time, a sample database is created and then utilized as a template database for creating all subsequent databases. PostgreSQL provides a utility called *initdb*, which can be used to specify all of the environmental variables associated with the PostgreSQL database environment. The *initdb* utility can also be used when upgrading an existing PostgreSQL environment to a newer release of PostgreSQL. Once the initial PostgreSQL database is created, it will reside in the database directory area under the name *template1*. When any new databases are created on the server, the internal structures from within the *template1* database are all copied to establish the foundation for the new database.

Thus, the PostgreSQL infrastructure provides a means of standardizing database implementations across an entire organization through the use of the template database. Database Administrators can apply environmental standards, and/or define custom objects within the *template1* database. Once these standards are established, they can be incorporated into new PostgreSQL databases that are created on the server, using the *template1* database as a base.

New databases are typically created using another PostgreSQL utility called *createdb*. The *createdb* utility merely takes input from the operating system command line and passes this information onto the psql interface tool to execute a SQL "CREATE DATABASE" statement. The example below shows how to invoke the *createdb* utility.

$ createdb testdb

The above command creates a new PostgreSQL database instance called *testdb*, taking all of the standard defaults based upon predefined PostgreSQL environmental variables. If any customizations are required, parameter options can be specified on the *createdb* command line to install the database in an alternate location, or specify a different database owner, etc. By default, the user account that creates a PostgreSQL database becomes the owner of the database, unless an alternate user is specified. Once the database is successfully created, Database Administrators can create users and other database objects within the new database environment.

In addition to using the *createdb* utility, other database tools are available for free, which provide a Graphical User Interface (GUI) to the PostgreSQL environment. PgAccess and pgAdmin are two GUI tools that provide functionality for creating and maintaining PostgreSQL databases. PgAccess, which was developed by Constantin Teodorescu from Romania, is written in Tcl/Tk and runs on several operating environments including Linux/Unix, Windows, Macintosh, and AS400. PgAccess will be included with the Great Bridge software distribution of the PostgreSQL database in November 2000. In addition, PgAccess can be downloaded from the Internet at www.flex.ro/pgaccess/index.html. The pgAdmin tool, which was developed by Dave Page from the United Kingdom, is a Windows based application that enables remote client desktops to interact with PostgreSQL using Open Database Connectivity (ODBC). The pgAdmin software can be downloaded from the Internet at www.pgadmin.freeserve.co.uk/.

Both the PgAccess and pgAdmin tools are referenced throughout this document as viable alternatives to many of the standard PostgreSQL command line utilities. Most all of the PostgreSQL utilities, including many of these open source tools, execute standard SQL statements to create and maintain PostgreSQL database objects.

PostgreSQL Database Files

There are several environmental variables that must be defined within the PostgreSQL environment. These variables are used to locate where key PostgreSQL components reside within the operating system. These are defined during the PostgreSQL installation, but can be overwritten when invoking any of the PostgreSQL utilities. All of the PostgreSQL utilities accept run-time parameters for specifying custom settings, or overwriting the system defaults. The table below highlights the main environmental variables and file locations that are referenced by the PostgreSQL software. Typically, PostgreSQL executables reside in the ***/usr/bin*** directory, but this is not a requirement.

PostgreSQL Environment Variables	***Purpose of PostgreSQL Variable***
PGDATA	Defines the physical location where the PostgreSQL Data Directory resides on the server. There is a *base* subdirectory established in this area, where all the defined PostgreSQL databases have their own unique subdirectories. Typical location for the data is ***/var/lib/pgsql/data***. In the example of our *testdb*, its physical database files will be located in the following directory: ***/var/lib/pgsql/data/base/testdb***.
PGHOST	References the Database Server Name where PostgreSQL is installed and running.
PGLIB	Defines the location where all the PostgreSQL Run-Time Libraries and Configuration files exist. Typical location for these files is in the following directory: ***/usr/lib/pgsql***.
PGPATH	Defines the default location of where all PostgreSQL command line utilities and tools reside on the server. These files will typically reside in the following area: ***/usr/bin***.
PGPORT	Defines the default communication port that has been assigned to the *Postmaster* process. All PostgreSQL Client requests must reference this port when communicating with a PostgreSQL database. Default port setting is 5432.
System Variables	***Purpose of System Variable***
PATH	Standard system variable used by the operating system to locate executable programs. Any user accounts, which require access to PostgreSQL, should include the directory location where the PostgreSQL executables reside in their PATH setting. Typically, these files reside in ***/usr/bin***.
LD_LIBRARY_PATH	References the location of the shared system libraries that reside on the server. The PGLIB variable must be appended to this variable to allow other system resources to reference PostgreSQL's Run-Time Libraries.

PostgreSQL System Tables

The core database engine within PostgreSQL utilizes a sophisticated set of system tables, which are responsible for cataloging and maintaining all structures that are defined within the database. All of the PostgreSQL system table names are prefixed with the letters "*pg_".* There are over twenty-seven inter-related system tables within a PostgreSQL database. As previously mentioned, the PostgreSQL database was architecturally designed to support both relational and objected-oriented database models. This is obvious when reviewing the primary system table relationships.

All database components such as tables, columns, indices, functions, etc. are all identified as objects, which are subdivided into classes. Each class of object has attributes and access methods that are defined within the internal system tables. The table below provides an alphabetical listing of some of PostgreSQL's system tables, describing how they are referenced within the PostgreSQL database environment.

PostgreSQL System Table Catalog	***Brief Description of System Table***
PG_AGGREGATE	Stores standard and customized Aggregate Functions that have been defined within the database.
PG_AM	Describes Access Methods that are available within the database for storing indices.
PG_AMOP	Describes Access Method Operators.
PG_AMPROC	Stores Access Method Support Functions.
PG_ATTRIBUTE	Stores Attribute Definitions and related Data Integrity Rules associated with attribute.
PG_CLASS	Stores Table Definitions.
PG_DATABASE	Stores information about all existing PostgreSQL databases that reside on the server, including their data directory paths.
PG_GROUP	Stores all defined user groups with references to all the database users that are considered to be members of the group. Used for granting access to designated database objects.
PG_INDEX	Stores Index Definitions that have been created for each Table in the database.
PG_OPCLASS	Describes Access Method Operator Classes.

PostgreSQL System Table Catalog	***Brief Description of System Table***
PG_OPERATOR	Stores all standard operators and custom operators defined within PostgreSQL, along with their associated precedence and functions.
PG_PROC	Stores all Database Procedures and their associated attributes that have been defined within the database.
PG_RELCHECK	Stores all defined data integrity checks/constraints.
PG_REWRITE	Stores Rewrite Rule Definitions used by *the Query Rewrite System* when processing database queries.
PG_SHADOW	Stores all database users and their associated privileges that have been assigned to them.
PG_TRIGGER	Stores information about defined Database Triggers.
PG_STATISTICS	Stores internal statistics about database objects, which is referenced by the PostgreSQL Optimizer.

Many of these system tables can be accessed via standard SQL query statements. However, some of the internal data that is stored in these tables may not make any sense, since the structures are intended to be referenced by the core PostgreSQL database engine. The purpose in presenting this information is to provide Database Administrators and Developers with an overview of where database objects are defined and stored internally within the database.

Database Security and User Access

The PostgreSQL database environment provides several layers of security for protecting both the data and the physical database structures themselves, and for preventing unauthorized access to the environment. Security mechanisms can be enabled or disabled at several points to establish the appropriate level of protection required by your operating environment. Security checks can be enabled for clients accessing PostgreSQL via a remote network or local host connection. In addition, security can be enabled internally within the database by assigning specific database privileges and passwords to database users and/or groups of users.

By default, the person who creates a new PostgreSQL database becomes the owner of the database. This is also true for other database objects like tables, functions, and procedures where the creator becomes the owner of the defined object. Database object owners and users, who are assigned SuperUser privileges, are the only people who can make structural modifications to objects defined within the database and/or delete them.

The concept of SuperUsers within PostgreSQL is similar to how the “root” account works within a Linux/Unix environment. SuperUsers can bypass all database privilege checks that are applied to normal database users. In addition, SuperUsers can create, delete, and modify database user privileges. Typically, SuperUser privileges are reserved for Database Administrators, but they can be granted to other users if deemed necessary. One other privilege, which can be granted to database users, is the right to create databases within the PostgreSQL environment. Basic data manipulation of database objects is controlled by assigning access rights to all database users and/or groups of users.

For example, the database owner or Database Administrator may want to only assign UPDATE capabilities for an *employee* table to designated department managers. This can easily be done by establishing a group called *managers*, and assigning only certain database users to this group granting them the appropriate access privilege. Generic

access can be assigned using the keyword PUBLIC, which applies to all database users that have access to the database.

Four major access types exist within the PostgreSQL database environment. These access types can be assigned to individual users and/or groups of users, dictating what type of operations can be performed on the designated database objects. The following table shows the four access types that are available within PostgreSQL, along with a brief description of what operations can be performed with the associated access permission type.

PostgreSQL Access Permission Types

PostgreSQL Access Type	***Access Type Description***
READ	Read-Only Access, which is shown using a lowercase ("r"), allows users to issue SELECT statements on the designated database object. Granting any other privileges like Append, Write, or Rule to a user will automatically imply the user will have Read Access as well.
APPEND	Append Access enables users to issue INSERT statements on the designated database object. Access permission is shown within the database using a lowercase ("a").
WRITE	Write Access enables users to issue both UPDATE and DELETE statements on the designated database object. Access permission is shown using a lowercase ("w").
RULE	Rule Access is a special access privilege within PostgreSQL that allows users to define customized processing rules within the database engine itself. As previously discussed, the PostgreSQL Rules System is invoked during the Rewrite Phase of processing a database query. Access permission is shown using an uppercase ("R").

If at any time someone wants to view the assigned permissions on a particular database object, they can use the PostgreSQL command line interface *psql* to obtain this information. The *psql* utility provides several user functions or meta-commands that enable users to retrieve information on various database objects. For example, database permissions can be viewed by simply issuing a meta-command (e.g. "\dp database-object-name is a shortcut command meaning "describe permissions" assigned to the specified database object) from within the *psql* utility. All *psql* meta-commands are specified using the

backslash ("\") character. In the example shown below, a *psql* meta-command is invoked, requesting the database to show what access permissions are assigned to the table *employee*, giving the following results.

```
$ psql testdb
testdb=# \dp employee
                Access permissions for database "testdb"
 Relation   |                  Access permissions
------------+-----------------------------------------------------------------
 employee | {"postgres=arwR","group managers=rw"}
```

Since the *testdb* was created by the *postgres* Account, the first access permission shows the *postgres* database user is granted full privileges to perform any database operation on the table *employee*. Granted permissions include READ, APPEND, WRITE, and the ability to define transformation RULES on the *employee* table. Only READ and WRITE privileges have been granted to the group *managers*.

External security measures can be established for PostgreSQL environments using a configuration file called *pg_hba.conf*, which is located in the /var/lib/pgsql/data area. Database and System Administrators can restrict database access to specific remote and local network clients using this configuration file, stipulating how users are authenticated. The *Postmaster* checks the *pg_hba.conf* file to authenticate and authorize database connectivity for all client requests that are submitted to the database server. Several authentication methods are supported by PostgreSQL, including Kerberos V4 and V5, which have become de-facto industry standards for authenticating users within a TCP/IP protocol network.

The *pg_hba.conf* file is comprised of several line entries that contain keywords for defining security settings for all potential network connections to the PostgreSQL environment. Within this structure, three record types can be specified: host, local, and hostssl, which provides authentication of users via Secured-Socket Layers (SSL). Each record type has its own set of arguments, but the basic syntax includes the name of the database that the client can access, the complete or partial IP Address information associated with the client, and the authentication method to be used. As previously mentioned, PostgreSQL supports

several authentication methods. The table below summarizes each authentication method that can be applied within the *pg_hba.conf* file:

Authentication Method	***Description of Authentication Method***
Trust	No authentication is done using this method. It assumes all client connections originating from the designed host IP Address have the authority to use whatever username they specify. This is the least secure of all the authentication methods, and it is not recommended unless you are working in a closed secure network environment.
Password	Expects the client to provide an username and password that is validated against PostgreSQL's Internal Security Files.
Crypt	Expects the client to pass an encrypted username and password over the network that can be validated against PostgreSQL's Internal Security Files.
Ident	Authentication is done by matching a supplied identifier, and mapping the identifier to a designated PostgreSQL database user, which is defined within a map file called *pg_ident.conf*.
Krb4	Provides support for Kerberos V4 Authentication, which uses secret-key cryptography to verify the identity of a client. For detailed discussion using this industry standard authentication method, consult information available on the Internet at www.isi.edu/gost/publications/kerberos-neuman-tso.html or web.mit.edu/kerberos/www.
Krb5	Provides support for Kerberos V5 Authentication, which is an enhanced release of the Kerberos Authentication Protocol that uses secret-key cryptography to verify the identity of a client. Kerberos V5 provides additional support for using multiple IP Addresses and networking protocols within an assigned Kerberos Ticket. Kerberos V5 also provides more encryption algorithms than Kerberos V4. For detailed discussion using this latest version of Kerberos, consult information available on the Internet at www.isi.edu/gost/publications/kerberos-neuman-tso.html or web.mit.edu/kerberos/www.
reject	Rejects any client requests that originate from the specified IP network address.

The most commonly deployed methods used within PostgreSQL environments are *password* and *crypt* authentications. However, many commercial implementations with a strong e-Commerce presence will probably want to deploy one of the Kerberos authentication methods. Depending upon the topology of the user community, multiple authentication methods can be deployed within the same business environment. Thus,

PostgreSQL provides the flexibility to interchange these internal and external security mechanisms to establish the most optimum environment for allowing full access to information, without compromising the integrity of the assets stored within the database.

Database User Setup and Assigning Security Privileges

Most database users are typically setup within the database and assigned access privileges using standard SQL statements like “CREATE USER” and “GRANT…ON…TO USER x”. PostgreSQL fully supports this method of establishing database users, but it also provides a database utility called *createuser* that accomplishes the same task. This utility will accept input parameters from the operating system command line. If no parameters are specified when the *createuser* utility is invoked, it will prompt the user for the required information necessary to establish a PostgreSQL database user. The basic syntax used in the *createuser* utility is demonstrated below.

```
$ createuser -A -D -e -P test_dbuser
Enter password for user "test_dbuser":
Enter it again:
CREATE USER "test_dbuser" WITH  PASSWORD 'test_dbuser' NOCREATEDB
        NOCREATEUSER;
CREATE USER
```

The example above specifies several parameter options for creating a database user by the name of *test_dbuser*. Parameter option (“-A”) restricts the database user from creating other database users, and option (“-D”) prevents them from creating new databases. The (“-e”) option instructs the *createuser* utility to display the generated SQL statement that will be executed by this command line. The last option (“-P”) instructs the *createuser* utility to prompt for the user password that will be initially assigned to the user. The two prompts following the command line are a result of specifying the (“-P”) option in the command line. Both the generated SQL statement and the end results are displayed on the client. As previously stated, the *createuser* would automatically prompt for this information, if no username or parameters were specified on the command line.

In addition to using the PostgreSQL *createuser* utility, the PgAccess tool enables authorized users to create database users using its GUI environment. PgAccess provides the facility to define new PostgreSQL database users, specifying the username, password, whether the user will have privileges to create other users and databases, and the ability to set an expiration date when the user will no longer be valid.

The pgAdmin tool also provides the capability of creating PostgreSQL database users from a desktop client. One additional feature, which the pgAdmin tool provides that is not available within PgAccess, is the ability to define user groups and assign database users as members of a group. This can greatly simplify the administration of granting access privileges amongst a large population of database users.

Both the PgAccess and pgAdmin tools provide the capability of viewing, modifying, and dropping database users all from within the same menu option. Both tools offer the ability for granting access privileges to individual users and/or groups of users on designated database objects, using their respective Table Option Menus. Depending upon the Database Administrator's preferences, any of these database interfaces can be used to establish users within the PostgreSQL environment. In the case of the PgAccess and pgAdmin tools, both provide additional capabilities in assigning database access privileges to users within a GUI environment.

Database Startup and Shutdown

Each database server must have at least one master process running in the background, which is called the *Postmaster*, to establish connectivity to a PostgreSQL database. The *Postmaster* is responsible for managing all incoming database requests and establishing database connectivity between front-end clients and one or more backend database processes. In addition, the *Postmaster* is responsible for allocating shared memory buffer pools and other system resources that are required by the backend *Postgres* database processes. All database startup and shutdown procedures are processed through the *Postmaster*.

Although the *Postmaster* can be directly invoked from the operating system prompt, the PostgreSQL software comes with a utility called *pg_ctl*, which can be used for starting, stopping, restarting, and reporting the status of the *Postmaster*. Both the *Postmaster* and the *pg_ctl* utility accept the same run-time parameters and arguments for specifying shared memory buffers, data directory paths, and backend processing options. However, the pg_ctl utility provides additional mechanisms for displaying the status of the *Postmaster* process, restarting the *Postmaster*, and shutting down the *Postmaster* with up to three different shutdown modes that control how the *Postmaster* handles the shutdown of all active backend *Postgres* processes.

The simplest method of starting a *Postmaster* server process is by issuing the following command using the *pg_ctl* utility.

$ pg_ctl –w start

As previously mentioned, there are a number of run-time parameters that can be specified at run-time. However, most of these are usually defined within a *Postmaster* Configuration File (i.e. postmaster.opts), which is located in the PostgreSQL Data Directory. Any options that are specified at run-time will override the default options established in the postmaster.opts file. In the above example, the parameter (“-w”) instructs the pg_ctl utility to wait for the *Postmaster* server process to come up; verifying a *Postmaster* Process Identification (PID) File is created.

To display the current status of the *Postmaster* process, the following *pg_ctl* command can be issued, giving the following results.

```
$ pg_ctl  status
pg_ctl: postmaster is running (pid: 6401)
options are:
/usr/bin/postmaster
-p 5432
-D /var/lib/pgsql/data
-B 64
-i
-N 32
```

This output shows a listing of all the various parameters that have been specified for the *Postmaster* server process. We can see from this example that the *Postmaster* is running, and has been assign a PID of 6401. The directory path where the *Postmaster* executable is located (i.e. */usr/bin* directory). The option ("-p") indicates the assigned Port for incoming database requests is set to Port 5432, which is the default setting for PostgreSQL. The option ("-D /var/lib/pgsql/data") indicates where the PostgreSQL data directory is located on the server. The remaining parameters affect the overall performance of your PostgreSQL environment and are dependent upon the availability of system resources running on the server.

The option ("-B") shows 64 shared-memory disk buffers are currently allocated for use by the backend *Postgres* processes. Option ("-i") enables "Internet Domain" connectivity for clients to connect to PostgreSQL databases via TCP/IP. Without this option, the *Postmaster* will only accept local socket connections from the host. The last parameter shown in the above example is the ("-N") option, which controls the maximum number of backend *Postgres* server processes that can be started by the *Postmaster*. Here, the current maximum is set to 32, but this can be set as high as 1024, depending upon the size and configuration of your server hardware.

Once the *Postmaster* is up and running, several backend *Postgres* server processes will be spawned and terminated as database requests are processed by the *Postmaster*. If the database is shutdown for any reason, the *Postmaster* is responsible for handling the orderly shutdown of all active *Postgres* backend processes. As previously mentioned, there are three different shutdown modes available via the pg_ctl utility. These are briefly described below:

- *(S)mart Shutdown Mode* – This is the default shutdown mode, which is used, if nothing is specified on the shutdown command line. When using this shutdown mode, the *Postmaster* will restrict any further database connections from being established. It waits for all current connections to logout; along with ensuring active transactions have completed before shutting down the database.

- *(F)ast Shutdown Mode* – Signals all backend *Postgres* processes that the PostgreSQL database is shutting down. All active database transactions are rolled back prior to actually shutting down the database.

- *(I)mmediate Shutdown Mode* - Signals all backend *Postgres* processes that the PostgreSQL database will be immediately shutdown and aborts all active transactions running within the database. This option should rarely be used unless there is a critical situation that requires immediate shutdown of all server processes. In this case, database recovery will probably be required when the database is brought back up.

Sample outputs for these various shutdown modes are demonstrated in the following examples. The shutdown mode parameter is prefixed with the option ("-m"), and followed by the first letter of the selected shutdown mode (i.e. "s" for *Smart Shutdown Mode*).

Sample pg_ctl Shutdown Procedure using Smart Mode:

```
$ pg_ctl -w -m s stop
Smart Shutdown request at Fri Aug 25 13:37:41 2000
Waiting for postmaster shutting down.....000825.13:38:02.549
Data Base System shutting down at Fri Aug 25 13:38:02 2000
done.
postmaster successfully shut down.
```

Prior to executing the Smart Shutdown, one interactive psql session was activated generating a SQL query against the database. The "Waiting for postmaster shutting down…." message appears because the *Postmaster* is waiting for the active database session to complete and logout. Once the session terminated, the *Postmaster* shutdown the database, stating the *Postmaster* was successfully shutdown.

Sample pg_ctl Shutdown Procedure using Fast Mode:

```
$ pg_ctl -w -m f stop
Fast Shutdown request at Fri Aug 25 13:53:37 2000
Data Base System shutting down at Fri Aug 25  13:53:37 2000
Waiting for postmaster shutting down...000825.13:53:37.335
Data Base System shut down at Fri Aug 25 13:53:37 2000
done.
Postmaster successfully shut down.
```

When specifying *Fast Mode Shutdown*, notice the database is immediately shutdown, prior to the *Postmaster* process being shutdown. ROLLBACKs were performed on any active database transactions that were running at the time the Fast Shutdown was issued.

Sample pg_ctl Shutdown Procedure using Immediate Mode:

```
$ pg_ctl -w -m i stop
Immediate Shutdown request at Fri Aug 25 14:14:35 2000
Waiting for postmaster shutting down..done.
postmaster successfully shut down.
```

When specifying *Immediate Mode Shutdown*, the database is immediately shutdown, aborting all active database transactions that were running when the Immediate Shutdown was issued. This shutdown mode is intended for emergency situations that require the database to be quickly shutdown.

At times there may be a need for a Database Administrator to restart the *Postmaster* to reconfigure the PostgreSQL environment, or change some run-time options to adjust the overall performance of the server. In this situation, the Database Administrator can issue a *pg_ctl* command using the *restart* argument. This instructs the *Postmaster* to perform a Smart Shutdown of the database, and immediately start the database again. The following command performs a database restart:

$ pg_ctl –w restart

As demonstrated in this section, PostgreSQL provides several options for performing database startup and shutdown operations, along with reporting the current status of the *Postmaster* server process. The intent behind the *pg_ctl* utility is to provide Database Administrators with a common tool for standardizing startup and shutdown operations within a PostgreSQL database environment. Many other database vendors require Database Administrators to write their own custom scripts for performing these types of functions. This often leads to inconsistent operating procedures within large ITS organizations and extends the learning curve for new DBA staff members. Thus, PostgreSQL alleviates these problems by providing a standard utility that is well documented regardless of the operating environment, where the database is installed.

PostgreSQL Backup and Restore

There are two primary mechanisms for backing up a PostgreSQL database. The recommended method is to use a PostgreSQL utility called *pg_dump*, or another variation called *pg_dumpall*. The intent behind both utilities is to dump all database object definitions including tables, indices, triggers, data, etc. into a flat file that can be used to restore the database to its original state at a given point in time. Multiple parameters can be specified with these utilities to accommodate unique backup requirements for a given database environment. Both "Hot" Backups, which allow backup operations to run while the database is running, and "Cold" Backups can be performed using these PostgreSQL backup utilities.

An alternative method to using one of these PostgreSQL utilities is to backup the physical database files, using the operating system level backup facility. However, file system backups have some inherit restrictions that must be considered when backing up PostgreSQL databases. Since file system backups do not interact with the internals of PostgreSQL like the *pg_dump* or *pg_dumpall* utilities do, the database server process must be shut down prior to performing file system backups. This must be done to ensure the integrity of the database structures is maintained internally within PostgreSQL. Consequently, the same is true when restoring physical database files from a file system backup.

Another consideration when selecting a system level backup approach is the file size of the backups. System level backups will be significantly larger than the *pg_dump*/*pg_dumpall* files because the data for both the tables and their associated indices must be backed up. In contrast, *pg_dump* and *pg_dumpall* utilities only dump the table data and the index definitions. The index data content is not backed up, since indices are rebuilt during the restore operation when using either *pg_dump* or *pg_dumpall*. Thus, the effort required to restore a database using a system level backup file versus a dump file produced by *pg_dump* or *pg_dumpall* can be significantly more involved.

Although database backups can be done using either approach, most business environments demand the type of flexibility offered by the *pg_dump* and *pg_dumpall* utilities. The distinction between using *pg_dump* versus *pg_dumpall* is whether you want to backup one single PostgreSQL database or an entire cluster of PostgreSQL databases that run on a given server. Depending upon your database backup strategy, one or both of these utilities may be appropriate for your environment.

Both utilities accept the same parameter options, which can be used to customize the output format of the generated dump files. The only difference between the two utilities is how they are invoked from the command line, and what kinds of privileges are required to run each utility. Since *pg_dumpall* assumes that it will dump all of the PostgreSQL databases that are running on the server, there are no arguments for specifying a database name like you have with the *pg_dump* utility. As previously mentioned, you can selectively backup specific tables when using the *pg_dump* utility. If you are performing a selective backup of a database table, the *pg_dump* utility requires both the name of the PostgreSQL database to be backed up, and the name of a specific table.

Since *pg_dumpall* must have access to every PostgreSQL database running on a given server, this utility does require "SuperUser" privileges like those assigned to the *Postgres* Account to perform a *pg_dumpall* operation. However, the *pg_dump* utility only requires a user to have *READ* access privileges to the database and/or objects being backed up. The examples below show the syntax for invoking these two utilities and the various options that are available for customizing your backups. The first example shows how to invoke the *pg_dumpall* utility, directing the output to be written to a file named dbserver_backups.sql:

$ pg_dumpall –v > /home/postgres/dbserver_backups.sql

The above parameter ("-v") instructs the *pg_dumpall* utility to operate in "verbose" mode, which enables the display of informational messages to the console during the dump operation. All related database objects and data are written to the specified file */home/postgres/dbserver_backups.sql* via the Linux/Unix pipe symbol (">"). The generated output from *pg_dumpall* is intended to provide a SQL based text file that can be used to restore all of the databases to their original state at the point of the backup. The dump file

contains SQL Data Definition Language (DDL) statements for recreating the database objects during a restore operation, and PostgreSQL COPY commands for loading the text-formatted data back into the database.

An example of how to invoke the *pg_dump* utility is shown below, which follows the same syntax as the *pg_dumpall* command with the added argument of a specific database to be backed up.

$ pg_dump –v testdb > /home/postgres/testdb_backup.sql

Here, the database *testdb* will be backed up using the *pg_dump* utility, and all of the related database objects that reside in this database will be dumped to an output file. The generated output will be written to the file *testdb_backup.sql*.

If a selective backup of one particular table within the database was needed, the above *pg_dump* command could be modified to only dump the contents of one table residing in the *testdb* database. This type of DBA operation is very common in environments where critical tables must be backed up for special processing or periodically archived after performing batch updates. The example below instructs the *pg_dump* utility to only dump the table *sales* to a dump file from the *testdb* database, using the parameter option ("-t") and the name of the table to dump (i.e. *sales*):

$ pg_dump –tv sales testdb > /home/postgres/testdb_sales_backup.sql

A sample of the output generated from the above *pg_dump* command is shown below. The output format of the dump file is identical to all the other examples previously mentioned for both the *pg_dump* and *pg_dumpall* utilities. In the case of the *pg_dumpall* utility, there would be a separate database connection statement embedded in the dump file for every database that was backed up on the server.

Generated Output written to testdb_sales_backup.sql:

```
\connect - postgres
CREATE TABLE "sales" (
        "stor_id" character varying(4) NOT NULL,
        "ord_num" character varying(20) NOT NULL,
        "ord_date" date NOT NULL,
        "qty" int4 NOT NULL,
        "payterms" character varying(12) NOT NULL,
        "title_id" character varying(6) NOT NULL,
        PRIMARY KEY ("stor_id", "ord_num", "title_id")
);
GRANT SELECT on "sales" to PUBLIC;
GRANT ALL on "sales" to "postgres";

COPY "sales" FROM stdin;
6380    6871    1994-09-14      5       Net 60  BU1032
6380    722a    1994-09-13      3       Net 60  PS2091
7066    A2976   1993-05-24      50      Net 30  PC8888
7066    QA7442.3        1994-09-13      75      ON invoice      PS2091
7067    D4482   1994-09-14      10      Net 60  PS2091
7067    P2121   1992-06-15      40      Net 30  TC3218
7067    P2121   1992-06-15      20      Net 30  TC4203
7067    P2121   1992-06-15      20      Net 30  TC7777
7131    N914008 1994-09-14      20      Net 30  PS2091
7131    N914014 1994-09-14      25      Net 30  MC3021
7131    P3087a  1993-05-29      20      Net 60  PS1372
7131    P3087a  1993-05-29      25      Net 60  PS2106
7131    P3087a  1993-05-29      15      Net 60  PS3333
7896    QQ2299  1993-10-28      15      Net 60  BU7832
7896    TQ456   1993-12-12      10      Net 60  MC2222
7896    X999    1993-02-21      35      ON invoice      BU2075
8042    423LL930        1994-09-14      10      ON invoice      BU1032
8042    P723    1993-03-11      25      Net 30  BU1111
8042    QA879.1 1993-05-22      30      Net 30  PC1035
\.

CREATE UNIQUE INDEX "sales_pk" on "sales" using btree ( "stor_id" "varchar_ops",
 "ord_num" "varchar_ops", "title_id" "varchar_ops" );
CREATE  INDEX "ord_num" on "sales" using btree ( "ord_num" "varchar_ops" );
```

Generated Output written to testdb_sales_backup.sql (Continued)

```
CREATE  INDEX "sales_by_dte_idx" on "sales" using btree ( "ord_date" "date_ops",
 "stor_id" "varchar_ops", "ord_num" "varchar_ops" );
CREATE CONSTRAINT TRIGGER "fk_on_sales_titleid" AFTER INSERT OR UPDATE
ON "sales"  NOT DEFERRABLE INITIALLY IMMEDIATE FOR EACH ROW
EXECUTE PROCEDURE "RI_FKey_check_ins" ('fk_on_sales_titleid', 'sales', 'titles',
'UNSPECIFIED', 'title_id', 'title_id');
CREATE CONSTRAINT TRIGGER "fk_on_sales_storid" AFTER INSERT OR UPDATE
ON "sales" NOT DEFERRABLE INITIALLY IMMEDIATE FOR EACH ROW
EXECUTE PROCEDURE "RI_FKey_check_ins" ('fk_on_sales_storid', 'sales', 'stores',
'UNSPECIFIED', 'stor_id', 'stor_id');
```

Other parameter options are available to both the *pg_dump* and *pg_dumpall* utilities for customizing the generated output. For example, a DBA may only want to backup the data rather than backup both the database structures and their associated data contents. This is easily handled by specifying a run-time parameter ("-a"), which flags the utility to dump the data without the schema definitions. Another parameter option ("-s") will dump just the schema definitions. Consequently, multiple *pg_dump* and *pg_dumpall* operations can be configured to accommodate many different backup strategies.

Database restores are handled by invoking the PostgreSQL Interactive SQL Tool called psql, passing the generated output from either *pg_dump* or *pg_dumpall* as input to this database interface. Using our previous example from the *pg_dump* of the *testdb* database, we can restore this database by issuing the following command at the operating system prompt:

$ psql –e testdb < /home/postgres/testdb_backup.sql

The parameter option ("-e") is used to instruct the psql tool to display all of the commands that are executed within the psql session. In this example, we are connecting to the *testdb* database, and the generated dump file *testdb_backup.sql* is being read as standard input to restore the database. Once the dump file is processed within the psql session, the *testdb* database will be restored to its original state at the point when the backup operation was performed.

Import and Export Tools

The PostgreSQL Database provides several options for importing and exporting data. The most common means of importing data is using the PostgreSQL COPY command, which accepts input from a flat file or the operating system's standard input device (i.e. stdin). The format of the input file can be either text or binary, using any type of field delimiters that can be specified within the COPY command. The End-Of-File (EOF) marker within the data file is indicated by using a special terminator symbol ("\.") followed by a new line at the end of the file. The COPY command is executed from within the *psql* utility. Thus, data can be easily loaded directly into a PostgreSQL database without having to write complex scripts or programming code. An example of importing data with the COPY command is shown below:

Data Import Sample using COPY Command

```
$ psql testdb
testdb=> COPY jobs FROM '/home/postgres/jobs_data.txt' USING DELIMITERS '|';
1|New Hire - Job not specified|10|10
2|Chief Executive Officer|200|250
3|Business Operations Manager|175|225
4|Chief Financial Officier|175|250
5|Publisher|150|250
6|Managing Editor|140|225
7|Marketing Manager|120|200
8|Public Relations Manager|100|175
9|Acquisitions Manager|75|175
10|Productions Manager|75|165
11|Operations Manager|75|150
12|Editor|25|100
13|Sales Representative|25|100
14|Designer|25|100
\.
```

In this example, we have a table called *jobs* that already exists within the database. The table is comprised of four columns: job_id, job_desc, min_lvl, and max_lvl. As specified in the COPY statement, the data is delimited by a pipe symbol ("|"). Similarly, data can also be exported from PostgreSQL using the COPY command. When exporting data, the COPY command syntax changes by substituting the "FROM" with a "TO". All standard data integrity checks are enforced using defined database triggers and constraints.

Another option for importing and exporting data into PostgreSQL is to use the PgAccess GUI Tool. PgAccess provides both import and export capabilities within a Linux/Unix environment. Using the PgAccess Import Table function, data can be imported into a PostgreSQL Table from a flat file that resides on the file system. The tool actually mimics the functionality of the COPY command by enabling users to specify the target table that will receive the imported data, the import filename, and the column/field delimiter values. Then, PgAccess parses this information and formulates a COPY command to be executed by the backend *Postgres* process (Refer to Section 6 – Database Utilities for more details regarding other PgAccess features).

The windows-based tool pgAdmin can also import and export data into PostgreSQL via its ODBC interface. Using pgAdmin's IMPORT Tool, data can be loaded into PostgreSQL tables directly from delimited flat files that reside on other operating platforms other than Linux/Unix. The tool provides some additional formatting capabilities, such as specifying the column order of the input data and mapping the input to the target table columns. In addition, this tool gives status summaries of the import operation reporting the number of records imported, and the number of records that encountered errors. All data errors are written to a log file, along with details about the errors that were encountered (See Section 6 – Database Utilities for a more detailed discussion of pgAdmin's capabilities).

As was previously mentioned, data exports from PostgreSQL can be accomplished using the COPY command from within the psql interface, PgAccess, or pgAdmin. However, pgAdmin also provides Exporter Tools that can export data into HTML formatted files for display on the Web, or Excel using Microsoft's Object Linking and Embedding (OLE) technology. These Exporters are incorporated into several pgAdmin functions, such as the Quick SQL Tool and the Data Viewer, which is a tool that displays the contents of selected PostgreSQL Tables.

The pgAdmin Quick SQL Tool is useful for customizing queries to extract data based upon some selection criteria. Then, the query results can be exported to either an HTML Web Page or Excel Spreadsheet. This latter technique is commonly used in the business

community for producing Ad Hoc Reports required for business analysts and management. The HTML Exporter is helpful for data warehousing environments that require monthly and quarterly snapshots of data to be viewable corporate-wide across an established Intranet environment.

In addition to these useful third-party interfaces to PostgreSQL, there is an integral feature within the *pg_dump* Backup utility that can be used for exporting data from a PostgreSQL database. Although *pg_dump* is typically used for performing database backups, it can also serve as a data exporter. By specifying various *pg_dump* parameters, data can be extracted from an entire database or a select set of tables. For example, data from one PostgreSQL table can be extracted to a flat file by issuing the following *pg_dump* command:

$ pg_dump –at jobs testdb > /home/postgres/dump_jobs_data

The above parameters ("-at") instruct the *pg_dump* utility to dump only the data from the table *jobs* from within the *testdb* database. The data output is written to the specified file */home/postgres/dump_jobs_data* via the Linux/Unix pipe symbol (">"). The output format of the data will be tab-delimited by default. The output file can then be imported into another PostgreSQL database, using the COPY command example that was previously shown without the "USING DELIMITERS '|'" clause.

One extensible option, which the *pg_dump* utility provides, is the facility for extracting data into a flat file that is formatted with ANSI SQL92 INSERT statements. Thus, output formatted in this matter via the *pg_dump* utility can be loaded into any other SQL92 compliant database, using standard SQL scripts. The example below shows how this can be done using the parameter option ("-d"), which instructs *pg_dump* to extract the data, encapsulating the data with SQL92 "INSERT…VALUE" statements.

$ pg_dump –adt jobs testdb > /home/postgres/insert_jobs_data.sql

Sample output generated from this pg_dump command is shown below:

```
INSERT INTO "jobs" VALUES (1,'New Hire - Job not specified',10,10);
INSERT INTO "jobs" VALUES (2,'Chief Executive Officer',200,250);
INSERT INTO "jobs" VALUES (3,'Business Operations Manager',175,225);
INSERT INTO "jobs" VALUES (4,'Chief Financial Officier',175,250);
INSERT INTO "jobs" VALUES (5,'Publisher',150,250);
INSERT INTO "jobs" VALUES (6,'Managing Editor',140,225);
INSERT INTO "jobs" VALUES (7,'Marketing Manager',120,200);
INSERT INTO "jobs" VALUES (8,'Public Relations Manager',100,175);
INSERT INTO "jobs" VALUES (9,'Acquisitions Manager',75,175);
INSERT INTO "jobs" VALUES (10,'Productions Manager',75,165);
INSERT INTO "jobs" VALUES (11,'Operations Manager',75,150);
INSERT INTO "jobs" VALUES (12,'Editor',25,100);
INSERT INTO "jobs" VALUES (13,'Sales Representative',25,100);
INSERT INTO "jobs" VALUES (14,'Designer',25,100);
```

This capability further demonstrates how PostgreSQL can be integrated into existing database environments, allowing data to be easily shared across multiple database platforms. For additional details on the *pg_dump* utility, please refer to the PostgreSQL Backup and Restore Section in Chapter 3 – Physical Database Environment.

4 Database Management and Database Monitoring

The PostgreSQL database provides several tools for assisting in the day-to-day management and monitoring of the database environment. These tools come in the form of run-time parameters that can be set to write specific information to the database log files and operating system utilities that display information about active server processes. Generally, both System Managers and Database Administrators actively participate in monitoring activity on the database server. Potential bottlenecks that impact performance may be detected from one of several different sources, which may require more detailed information to be collected. Once this information is captured, it can be reviewed by your ITS staff to evaluate the overall performance of the database server.

User Access Monitoring Tools

Two operating system utilities, which are commonly used to monitor user access in a Linux/Unix environment, are the *Process Status (ps)* utility and the *top* utility. The *ps* utility provides a snapshot of information about current processes that are running on the server. The *top* utility is an interactive tool that dynamically gathers process statistics about the most CPU intensive processes that are running on the server. Both utilities accept options for specifying particular information to display in their output listings. Since the PostgreSQL environment is comprised of at least one primary process, which is the *Postmaster* process, and one to many backend *Postgres* processes, Database Administrators can selectively target processes to monitor by referencing one of these process names.

The example below shows how to monitor all of the *Postgres* backend processes that are running on the server using the *ps* utility. Since we are primarily interested in monitoring only the *Postgres* backend processes, the option ("U process-name") is used to instruct the *ps* utility to only display processes that have the process name of *postgres*.

```
$ ps U postgres
  PID TTY      STAT   TIME COMMAND
 2481 ?        S      0:00 /usr/bin/postgres 216.54.52.152 smith pubs idle
 2899 ?        R      0:00 /usr/bin/postgres localhost dbuser test SELECT
 3104 ?        S      0:00 /usr/bin/postgres 216.54.52.147 dbamgr pubs idle
 3105 ?        D      0:00 /usr/bin/postgres 216.54.52.147 dbamgr test commit
```

Based upon the results displayed by the *ps* utility, we can see there are four active user connections running on the server, and only two PostgreSQL databases are being accessed. Reading the results from left to right, we see a process, which is assigned a process identifier (PID) of *2481*, is connected via a remote TCP/IP connection, originating from a remote computer with a TCP/IP address of *216.54.52.152*. This user is logged into the server under the username of *smith*, and is accessing the *pubs* database. Currently, the process is idle. The second process under *PID 2899* is connected via a local host connection under the username of *dbuser*, and is accessing the *test* database. Here, the *ps* utility shows us *dbuser* is executing a SQL SELECT statement against the *test* database. Finally, we see two processes (i.e. *PID 3104 and 3105*) that are both assigned

to the same user (*dbamgr*), but this user is accessing two separate databases (i.e. *pubs* and *test*).

The *STAT* or state code shows the current state of the process. In the above example, all of the processes are (S)leeping with the exception of *PID 2899*, which has a process state of (R)unning because it is processing a SQL SELECT statement running in the *test* database. In addition, *PID 3105* has a process state of (D), which indicates the process is in an uninterruptible sleep state. This state usually reflects some type of I/O activity, which is confirmed by the fact that the *ps* utility shows *PID 3105* is in the process of committing a transaction to the database. Other types of state codes, which may appear in the *ps* results, are (T) indicating the process is in a traced or stopped state, and (Z) indicating the process is in a "zombie" state. Several options can be specified on the *ps* utility command line to display different information and output formats in the results. For further details, please refer to your operating system documentation for other available *ps* options.

To get more specific information about the system resources that each process is using, the *top* utility can be used. The *top* utility provides an interactive display of the activity that is running on the server in real time. The results are periodically refreshed based upon a specified interval to reflect changes occurring on various processes. For demonstration purposes, let's say we want to get more information about the four processes shown in our *ps* results example. The *top* utility enables us to selectively monitor these processes by specifying their assigned process identifiers using the option ("-p"). This instructs the *top* utility to only display information related to the specified process identifiers.

The following example shows how to use the *top* utility for monitoring the four process identifiers listed in the previous example.

```
$ top –p 2481 –p 2899 –p 3104 –p 3105
6:54pm  up 5 days, 10:20,  4 users,  load average: 0.37, 0.11, 0.03
4 processes: 3 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  5.5% user,  1.9% system,  0.0% nice, 92.5% idle
Mem:   517120K av,  269540K used,  247580K free,   92600K shrd,  182352K buff
Swap:  530104K av,          0K used,  530104K free    31692K cached

 PID  USER     PRI  NI  SIZE  RSS SHARE STAT LIB  %CPU %MEM   TIME  COMMAND
2481  postgres   0    0  2856  2856  2268       S    0   0.0       0.5   0:00  postmaster
2899  postgres   0    0  2868  2868  2212       R    0   0.0       0.5   0:00  postmaster
3104  postgres   0    0  2736  2736  2220       S    0   0.0       0.5   0:01  postmaster
3105  postgres   6    0  2708  2708  2196       D    0   2.3       0.5   0:00  postmaster
```

Reading the above results from the *top* utility, we can see the four processes referenced in the *ps* example are still active. The *top* utility also shows the status of all the system resources that are currently being utilized. Some of the resources shown include the amount of available memory with the average amount of memory used, available shared memory, amount of allocated memory buffers, and the swap file statistics. Although the results indicate the system is running about 92.5% idle, the process assigned *PID 3105* is using approximately 2.3% of the CPU. Using the results from both the *ps* and *top* utilities, we know *PID 3105* is obviously performing the most intensive database activity compared to the other active users that are currently running.

Once Database Administrators have isolated the various PostgreSQL processes that they want to monitor on the server, more database specific information can be obtained using PostgreSQL's database logging and debugging parameters. PostgreSQL provides several levels of logging information depending upon the amount of detail that is needed for monitoring activity. Both the *Postmaster* and *Postgres* processes will accept run-time parameters that specify the level of logging desired. Debug logging levels for PostgreSQL are controlled by the run-time parameter option ("-d integer-value"), which can be passed to the PostgreSQL server processes. Zero is the default value assigned to this option, which means no debugging output is generated to the PostgreSQL log file.

The higher the debug level value is set, the more “debugging” information that is generated. Generally, there is no need to specify any higher setting than four when debugging or monitoring most database situations. Logging information is written to the system log file, unless otherwise specified. The example below shows how to specify several debug options using the *Postmaster* command line, and directs the output to a database log file located in the home directory assigned to the *postgres* user account (e.g. */home/postgres*).

$ postmaster –i -d 2 -o "-d 3 -tpa -te" >>/home/postgres/postgresql_db.log 2>&1 &

Reading from left to right, the *Postmaster* process option (“-i”) is used to enable TCP/IP connectivity to the backend database, and the debug level option ("-d") is set to a value of two. The *Postmaster* option (“-o”) specifies processing parameters that will be passed onto any *Postgres* backend processes. All backend options must be enclosed within double quotes. Here, the *Postgres* backend process debug level is set to a value of three. In addition, two run-time options are specified that include the *show_parser_stats* and the *show_executor_stats,* which are both set to on using options (“-tpa” and “-te” respectively).

The command syntax (“*>>/home/postgres/postgresql_db.log*”) tells the *Postmaster* to redirect standard output messages to the designated database log file (e.g. */home/postgres/postgresql_db.log).* The phrase (“2>&1”) instructs the *Postmaster* to also send any error messages to the same file where standard output information is being written (i.e. */home/postgres/postgresql_db.log)*. Normally, output redirections are specified using only one (“>”) symbol. The double (“>>”) symbol syntax is used here because both standard output messages and error messages are being redirected to the same file (i.e. The double (“>>”) symbol is used to append standard error messages to the same file that is being used for logging standard output messages). The debug levels for both the *Postmaster* process and the *Postgres* backend processes are set to zero by default, which means logging is turned off. Finally, the ampersand (“&”) at the end of the command line instructs the *Postmaster* to startup as a background process.

Continuing with our scenario from the *ps* and *top* results, the Database Administrator can search for specific PIDs within the database log file to monitor their transaction activity. Below is an excerpt of the database log information that was generated from the debugging options specified above. Only activity relating to *PID 3105* is shown in this example, since we know from the *top* output results that this process is performing a CPU intensive operation. Using the log information, we can isolate what type of SQL statement *PID* 3105 is performing.

```
000919.15:13:20.597 [3105] started: host=216.54.52.147 user=dbamgr database=test
000919.15:13:20.597 [3105] InitPostgres
000919.15:13:20.694 [3105] reset_client_encoding()..
000919.15:13:20.695 [3105] reset_client_encoding() done.
000919.15:13:20.695 [3105] StartTransactionCommand
000919.15:13:20.695 [3105] query: select getdatabaseencoding()
000919.15:13:20.699 [3105] ProcessQuery
000919.15:13:20.699 [3105] CommitTransactionCommand
000919.15:13:25.425 [3105] StartTransactionCommand
000919.15:13:25.425 [3105] query: DROP TABLE "roysched";
000919.15:13:25.425 [3105] ProcessUtility: DROP TABLE "roysched";
000919.15:13:25.442 [3105] CommitTransactionCommand
000919.15:13:25.567 [3105] StartTransactionCommand
000919.15:13:25.567 [3105] query: CREATE TABLE "roysched" (
        "title_id" character varying(6) NOT NULL,
        "royalty" int4 NOT NULL,
        "lorange" int4,
        "hirange" int4,
        PRIMARY KEY ("title_id", "royalty"));
000919.15:13:28.601 [3105] CommitTransactionCommand
000919.15:13:28.603 [3105] StartTransactionCommand
000919.15:13:28.604 [3105] query: INSERT INTO "roysched" ("title_id","royalty",
"lorange","hirange") VALUES ('BU1032',0,5000,10);
000919.15:13:28.606 [3105] CommitTransactionCommand
000919.15:13:28.606 [3105] StartTransactionCommand
000919.15:13:28.607 [3105] query: INSERT INTO "roysched" ("title_id","royalty",
"lorange","hirange") VALUES ('BU1032',5001,5,12);
```

Reading from the top, we can see that *PID 3105* connected to the *test* database on September 19th at 3:13 PM, logging in as user *dbamgr*. Jumping down nine lines, we see that the user issued a SQL “DROP TABLE” statement iagainst the *roysched* table. This is followed by a SQL “CREATE TABLE” statement for lines later, which restores the *roysched* table structure. After the *roysched* table is restored, the log shows a series of SQL INSERT

statements. Based upon the above log information, it appears the *dbamgr* is performing some type of database restore operation, which explains why this process is using significant portions of the CPU resources.

Although this is only a small example of the type of logging mechanisms available within PostgreSQL, the scenario presented in this section demonstrates how Database Administrators can use the monitoring tools that are available within PostgreSQL.

5 Database Performance Tuning

Several techniques can be applied to the PostgreSQL environment to fine-tune database performance. Of course, overall database performance is impacted by several factors including the operating environment where the database is running, number of users accessing the database, and whether other applications must share the available server resources. The topics addressed in this section will demonstrate the various tools and techniques that are available within PostgreSQL for tuning potential bottleneck issues and maintaining good overall database performance.

In addition, there are some general practices that can be deployed to ensure PostgreSQL is being used effectively. For example, large data imports and/or loads are best done using the PostgreSQL *COPY* command rather than using standard SQL *INSERT* statements. The *COPY* command bypasses a lot of database constraints that enable the database to load data extremely quickly into PostgreSQL tables. Performance can also be improved when using either *COPY* and/or *INSERTS* if the target table indices are dropped, and reapplied after the bulk load operation completes.

Another technique for improving data access performance is to cluster data into groups. This is accomplished by using the *CLUSTER* command, which reorders the data that is stored in tables to match the ordering of an index. This command is useful in situations where there are duplicate values for columns that are used to retrieve data, based upon some type of selection criteria. For example, a book publisher may have a list of books indexed by author. If all the books written by the same author are grouped together on the same database page, disk I/O could be reduced by implementing clustering on the author index. The syntax used to cluster data in this fashion is outlined below:

CLUSTER index-name ON table-name

In our example above, the index-name that references the column *authors* in the table *titleauthors* would be substituted in the *CLUSTER* statement to reorder the data to match the index order. This technique is also helpful in eliminating the need for sort operations, whenever *ORDER BY* clauses are used in database queries.

Run-time Configuration Settings for PostgreSQL Servers

Many factors must be considered when configuring a PostgreSQL server environment for optimum performance. The only limitations imposed on PostgreSQL are the amount of system resources that are physically available on the system. System resources include the amount of physical memory available on the system, physical disk space, the number of CPU processors running on the server, and the speed of these processors. In addition, one must consider whether PostgreSQL will be running on a dedicated server, or whether it will share the system resources with other applications and/or packaged solutions. In all cases, tuning the database server will involve balancing the performance requirements of the database with the amount of system resources that can be allocated for use by the PostgreSQL software.

The standard default settings for PostgreSQL are set to minimum levels to ensure the database will use the smallest amount of system resources when it is installed. For most database implementations, some of these run-time settings will need to be increased. However, it is best to initially startup the PostgreSQL database using the defined defaults, and then monitor over time the overall performance of the PostgreSQL environment. Prior to making any changes to the PostgreSQL run-time parameters, it is best to collect some statistics on both the operating system and the PostgreSQL database processes that are running on the server. This information is invaluable when making determinations about what environmental settings may need to be adjusted. The gathering of this information is typically a collaborative effort between Database Administrators and System Managers who are responsible for the operating environment.

As mentioned above, the overall performance of a PostgreSQL database server will be dependent upon its operating system resources and the applications that are running on the server. PostgreSQL will utilize all the allocated resources that are assigned to it during the startup process, if they are needed. In addition, some run-time parameters can be dynamically passed to the backend *Postgres* processes, which are spawned by the PostgreSQL *Postmaster* process. Consequently, it is the responsibility of the Database Administrators to ensure that the run-time configuration settings do not exceed the

available system resources. The more information Database Administrators have about their respective databases, the better equipped they are to properly configure PostgreSQL.

Although several run-time parameters exist within PostgreSQL, only subsets of these directly affect the overall performance of the database environment. All the other parameters can be applied on an as needed basis to customize the database's behavior for specific application needs. The table below highlights the PostgreSQL run-time parameters that have the most impact on database performance. The specified configuration guidelines are intended as "rules of thumb." Other considerations, which must be factored into these guidelines, include the number of concurrent users that are expected to access the database server and the types of concurrent database transactions that will be running on the server.

PostgreSQL Run-time Parameters	***Parameter Description***	***General Configuration Guidelines***
MaxBackends	Maximum Backend *Postgres* Processes that can be started by the *Postmaster*. This parameter should be based upon the anticipated number of concurrent users that will be accessing the database server. One *Postgres* backend process will be spawned for every database connection to the server. Option is specified at database startup on the *Postmaster* command line using parameter option ("-N").	Generally, this parameter setting should be set to the maximum number of anticipated concurrent database users that will access the system at one time, plus a safety margin. This parameter impacts other PostgreSQL run-time parameters like Shared Memory Disk Buffers and Sort Buffers. Thus, all dependent options must be adjusted whenever this parameter is changed. Default setting is 32 with a maximum setting of 1024. The maximum limit can be overwritten if the PostgreSQL software is recompiled with higher ceiling limits.

PostgreSQL Run-time Parameters	***Parameter Description***	***General Configuration Guidelines***
NBuffers	Specifies the amount of Shared Memory Disk Buffers that are allocated to PostgreSQL for database operations. Each buffer is typically 8192 bytes. The parameter setting is dependent upon the run-time value setting specified in the MaxBackends parameter. The Shared Memory Disk Buffers value cannot exceed the amount of physical Random Access Memory (RAM) that is available on the server. Option is specified at database startup on the *Postmaster* command line using parameter option ("-B").	Generally, this parameter setting should be minimally set to twice the value specified in the MaxBackends. If possible, larger settings should be specified ensuring the total amount of buffer space allocated does not exceed any more than half of the physical RAM that is available on the server. The intent is to allocate at least two separate buffers for every backend *Postgres* process that will be spawned by the *Postmaster*. For large installations with a lot of concurrent users, it is beneficial to have as much memory as possible. Note this parameter must be increased whenever the MaxBackends parameter is modified. Default setting is 64 Buffers. Maximum limits are only constrained by the amount of available RAM on server.
SortMem	Specifies the amount of memory that is allocated to backend *Postgres* processes to perform internal sorts and database hashes, before processes resort to using temporary disk storage. Value is specified in kilobytes, and this run-time parameter is passed to all the backend *Postgres* processes when they are spawned. Special consideration should be given to databases that process a lot of complex queries, since multiple sorts and merge operations can be invoked from one complex query. Option is specified at database startup on the *Postmaster* command line as part of the options list ('-o "…"') that is passed onto the *Postgres* backend processes. The SortMem option ("-S xxx") is enclosed in quotes when specifying it in the options list (e.g. '-o "-S 4096"').	Recommend increasing the value settings for this run-time parameter whenever complex queries are running slowly, or there are a significant number of temporary disk files being created, while queries are processing (e.g. sort file names to look for on system are *pg_tempnnn.nn*). Default setting is 512 Kb. Setting cannot exceed the amount of physical memory available for all server processes. Recommend starting with a minimum value of 4096 Kb.

PostgreSQL Run-time Parameters	***Parameter Description***	***General Configuration Guidelines***
FSYNC (boolean)	Parameter used to disable or enable whether PostgreSQL will automatically flush database write buffers, which are cached in memory, to the physical disk after every update transaction successfully completes. The name refers to the "File Synchronization" operation that PostgreSQL must perform to write all of the cached database transactions in memory to the physical PostgreSQL database disk files. If the parameter is disabled, the operating system controls the transaction buffering, sorting, and delaying disk writes as it sees fit. If the operating system controls when write caches are flushed to disk, then there is a potential risk that incomplete database transactions could result during power failures or system crashes. Having this parameter enabled forces database transactions to wait whenever the PostgreSQL database engine flushes the memory cache to disk. Thus, significant performance improvements can be achieved by disabling this feature. Option is enabled by default, but it can be disabled at database startup by passing the parameter as part of the options list ('-o "…"'), which passes the parameters onto the *Postgres* backend processes. The FSYNCH parameter is enclosed in quotes when specifying it in the options list (e.g. '-o "-F"'), which flags both the *Postmaster* and any *Postgres* backend processes to disable the FSYNC feature.	There is significant controversy surrounding the benefits and risks involved with disabling the FSYNC feature. However, some alternative solutions may be implemented to reduce some of the potential risk of disabling this option. For example, significant performance gains can be achieved during large load operations if FSYNC is temporarily disabled for bulk loads. Another alternative for reducing the risk is to invest in a reliable Uninterruptible Power Supply (UPS) to prevent power failures. Assuming you have a stable operating system combined with a reliable UPS backup, the FSYNC parameter could be disabled with minimal risks. Unless your database environment cannot tolerate any risk at all, the performance gains far outweigh the risks of disabling this parameter.

Prior to setting any of the above run-time parameters, Database Administrators should confirm what system resources are available on the server. This information combined with the recommended guidelines outlined in the above table will assist Database Administrators in tuning their PostgreSQL environments.

When parameter adjustments need to be made, they can be specified during database startup on either the *Postmaster* command line or by passing these settings via the *pg_ctl* utility. In addition, there is a PostgreSQL initialization file where run-time parameter settings can be permanently specified. The examples below show how the PostgreSQL parameters can be modified using one of these methods.

```
$ pg_ctl -o '-i -B 300 -N 150 -o "-F -S 8192"'  start
postmaster successfully started up.

$ pg_ctl status
pg_ctl: postmaster is running (pid: 2708)
options are:
/usr/bin/postmaster
-p 5432
-D /var/lib/pgsql/data
-B 300
-b /usr/bin/postgres
-i
-N 150
-o '-F -S 8192'
```

The above example uses the *pg_ctl* utility to startup the PostgreSQL database. Please note the use of the parameter options ("-o") that appear to be referenced twice. The first ("-o") option is a *pg_ctl* parameter, which instructs the *pg_ctl* utility to pass the specified run-time parameters onto the *Postmaster* process. The second ("-o") option is a *Postmaster* parameter, which is used by the *Postmaster*, to specify "backend" processing options that need to be passed onto all *Postgres* backend processes. All run-time parameters that are passed via the *pg_ctl* utility must be enclosed in quotes. Since one of the *pg_ctl utility* options ("-o") also requires quotes, the entire run-time parameter list is enclosed with single quotes, and the *Postmaster* options ("-o") list is enclosed with double quotes.

Reading from left to right, option ("-i") enables TCP/IP connectivity to the backend database, option ("-B") sets the Shared Memory Disk Buffers to 300, and option ("-N") sets the Maximum Backend Processes to 150. The FSYNCH feature is disabled with option ("-F"), and the Sort Buffer Size option ("-S") is set to 8192 Kb. The second *pg_ctl* command

requests the status of the *Postmaster* after the database is started. This command confirms the current PostgreSQL database settings that are running on the server.

An alternative method for starting up PostgreSQL databases with customized run-time settings is to invoke the *Postmaster* using the following command at the operating system prompt. The same run-time parameter settings, which we used in the *pg_ctl* example above, are specified here using the *Postmaster* command. The only additional syntax used with the *Postmaster* command is the ampersand (&), which is an operating system symbol that indicates the process should be started up as a background process.

```
$ postmaster –i -B 300 -N 150 -o "-F -S 8192" &
[1] 2752

$ pg_ctl status
pg_ctl: postmaster is running (pid: 2752)
options are:
postmaster
-p 5432
-D /var/lib/pgsql/data
-B 300
-b /usr/bin/postgres
-i
-N 150
-o '-F -S 8192'
```

As previously mentioned, run-time parameters can also be specified in an initialization file, which is typically located in the ***/etc/rc.d/init.d*** area. The name of the file is ***postgresql.init***. Whenever the server boots up, the *Postmaster* is started using this file. Thus, run-time parameters can be automatically assigned when the server reboots by editing the *postgresql.init* file and appending the appropriate parameter options to all of the command lines that reference the *Postmaster* command. The syntax for specifying these parameters in the initialization file is exactly the same as the example shown above, where the *Postmaster* is called directly.

Database Administrators must monitor and evaluate server performance on a regular basis to assess whether database needs and/or server demands have changed. The configuration guidelines outlined in this section demonstrate how PostgreSQL can be fine-

tuned to achieve optimal performance for a given server environment. Thus, PostgreSQL enables Database Administrators to proactively manage database performance.

Database Compression Tool (vacuum)

The PostgreSQL *vacuum* utility serves two primary purposes for ensuring a PostgreSQL database will perform at optimum levels. First, the *vacuum* utility is responsible for reclaiming storage space on disk that has not been completely freed up by the database engine. Part of this housekeeping function involves cleaning out ROLLBACK transaction segments, and/or table rows that have been marked for deletion. The second function involves collecting database statistics, which are used by the PostgreSQL Optimizer to determine retrieval paths for resolving queries. Database performance can be significantly improved by keeping these statistics current. The *vacuum* utility should be run on a nightly basis, and preferably after performing database backups.

Although the *vacuum* utility can be run without specifying any parameters, it does provide a facility for running selective *vacuum* operations on more dynamic database objects. For example, several volatile tables within the database may warrant running this utility more often than other objects, if they are constantly being modified or frequently accessed. The table below shows what parameter options are available within the *vacuum* utility and how they can be used.

VACUUM Run-Time Parameters	***Purpose of Parameter Option***
Verbose	Instructs the PostgreSQL *vacuum* utility to printout a detailed activity report for actions performed on each database object.
Analyze	Collects and updates table/column/index statistics stored within the System Catalog for use by PostgreSQL Optimizer. In addition, other housekeeping functions are performed to reclaim storage space on disk that has not been freed up by the database engine.
table-name	Performs *vacuum* operations on specified Table. Default setting is to perform *vacuum* operations on all database objects.

VACUUM Run-Time Parameters	***Purpose of Parameter Option***
table-name (column,.)	Performs *vacuum analyze* operations on specified column, and/or list of columns. *Special Note:* Using this syntax instructs the *vacuum* utility to only collect statistics for the columns which are listed, skipping other columns that exist within the specified table.

The PostgreSQL *vacuum* utility is invoked from within the psql environment. The *vacuum* operation targets the current database that was specified when launching the psql tool. The utility should be run when all database users are logged out of the database, or during slow processing periods to get the most benefit from performing a *vacuum* operation. In addition, only the database/table owner is authorized to run the *vacuum* utility. An example of how to execute the *vacuum* command from within psql is shown below:

```
$ psql testdb
testdb=> vacuum verbose;
```

The above example will perform both housekeeping functions and collect statistics on the *testdb* database, giving detailed activity reports on what was done during the operation. If a Database Administrator wanted to perform a selective *vacuum* operation on a group of columns that comprised a concatenated key for a table to update statistics, the following *vacuum* command could be invoked:

```
$ psql testdb
testdb=> vacuum verbose analyze authors (au_lname, au_fname);
```

In this example, the *vacuum* operation will analyze the specified columns *au_lname* and *au_fname*, which reside within the table *authors*. Both columns are part of a concatenated key that is used for performing name lookups within the database. The statistics gathered during this operation will be updated to the associated objects within the System Catalog. The PostgreSQL Optimizer will utilize these statistics to determine which path to use when retrieving data from the table *authors*.

Tuning Index Structures using Explain Plans

One of the most common tasks a Database Administrator will perform is to analyze why certain database queries take longer to run than other queries that appear to be similar in structure. At first glance, the Database Administrator will probably breakdown the SQL statements into manageable pieces to determine how the database engine might process the query to return the desired data results. Explain plans help to eliminate the guesswork involved with finding the root cause of why queries are not performing optimally. The intent behind an explain plan is to map out the data retrieval paths that the database engine will take to find the requested data.

Explain plans also provide a mathematical estimation of the resource costs involved with traversing a data structure. Resource costs include CPU utilization, disk I/O, sort operations, memory usage, and processing time involved with satisfying a database query. These cost estimates are similar to the way database optimizers find the best retrieval method for getting data out of a database. The PostgreSQL database provides a utility called *EXPLAIN*, which can estimate the costs involved with processing any SQL statement that is created within the database.

The EXPLAIN utility tabulates the startup costs associated with sorting data before it is used for scanning data values. It provides a ballpark estimate of the number of rows targeted for output, based upon database statistics stored within the PostgreSQL System Tables. In addition, it will provide an average size in bytes of the rows estimated to be returned, which helps the Database Administrator determine whether network buffer sizes may be exceeded. Finally, the utility will provide a total cost estimate for executing the specified query, giving its associated access paths.

Database Administrators can derive a wealth of information about how to tune and/or structure the data, based upon the output that is generated by the *EXPLAIN* utility. For example, the *EXPLAIN* utility can help Database Administrators determine whether a SQL statement will sequentially scan a table versus using a defined index. The utility shows what indices are being used, if any, along with whether data must be sorted/merged before

it can be processed. Thus, Database Administrators can experiment with making index changes and adjusting run-time parameters to see if the explain plan results are impacted by these changes. In some situations, adding or restructuring an index can make significant improvements on the overall performance of the database. Once the appropriate remedy is determined, it can be implemented in the base system environment.

The *EXPLAIN* utility is an invaluable tool in helping Database Administrators to monitor and debug database performance issues. The following example demonstrates how to invoke the *EXPLAIN* utility and the explain plans that it generates. In this example, the *EXPLAIN* utility is used to analyze a query that merges information from two tables, *titles* and *publishers*, which are linked together by a common data element named *pub_id*, and displays the results. The *pub_id* uniquely identifies publishers that have published various book titles. The query performs a simple JOIN between two tables, *titles* and *publishers*, using a foreign key containing the column *pub_id*.

```
$ psql testdb
testdb=# EXPLAIN select t.title_id, t.title, p.pub_id, p.pub_name, p.city, p.state,
testdb=#      p.country from titles t, publishers p where t.pub_id = p.pub_id;
NOTICE:  QUERY PLAN:

Hash Join  (cost=1.10..5.00 rows=36 width=96)
  ->  Seq Scan on titles t  (cost=0.00..2.36 rows=36 width=36)
  ->  Hash  (cost=1.08..1.08 rows=8 width=60)
        ->  Seq Scan on publishers p  (cost=0.00..1.08 rows=8 width=60)

EXPLAIN
testdb=#
```

The output from the *EXPLAIN* utility depicts a tree structure of processing steps that should be read from the bottom to the upper portions of the tree. The upper processing steps are the last operations to be executed during the query process. Thus, we will start at the first processing step, which performs a sequential scan of the *publishers* table. The *EXPLAIN* utility estimates this step will take up to 1.08 cost units, which means one disk page is fetched plus a little CPU time to perform the operation. The utility also estimates the scan will return eight rows with an averaging size of 60 bytes per row. Since there is no "WHERE" clause that independently restricts the rows retrieved from the *publishers* table,

we can infer only eight rows exist within this table. This is also consistent with the fact that the *publishers* table size fits within one database page.

The *EXPLAIN* utility shows the PostgreSQL Optimizer chose to perform a sequential scan of the *publishers* table rather than using its primary key index, which contains the column *pub_id*. If there were significantly more rows populated in the *publishers* table, the index would probably have been used to avoid the sequential scan. Moving on to the next processing step, we see the rows retrieved from the *publishers* table are placed into a "hash" table in memory, which is signified by the "Hash" statement. The next step shows a sequential scan of the *titles* table, which is estimated to return 36 rows within an average size of 36 bytes per row.

The final step performs a "Hash Join" operation that matches all of the rows returned from the *titles* table scan with the rows of *publishers* that reside in the "hash" table in memory. The *pub_id* values from the two tables are used as "hash" keys. For each match the "Hash Join" step will generate one joined row. The *EXPLAIN* utility estimates 36 joined rows will result. The total cost of this retrieval plan is estimated to be five cost units, or five disk pages fetched with an estimated 1.1 units expended before any output rows can result. The expended cost estimate accounts for processing time required to setup the "hash" table join. The PostgreSQL Optimizer estimates startup times separately from the total time so that it can make intelligent choices for queries using "LIMIT" clauses, since this would restrict the amount of rows needed in the query results.

Based upon the output generated from the above example, a Database Administrator may want to adjust some run-time parameters that are available within the PostgreSQL database to avoid some of the sequential scans highlighted by the *EXPLAIN* utility. As previously referenced early in this section, the PostgreSQL database provides several run-time parameters that can be adjusted to address a particular performance issue. Many of these parameters can be specified when invoking various database utilities. This enables Developers and Database Administrators to experiment with run-time configurations to achieve the optimum performance for a given application environment.

For example, there is a parameter that controls whether the PostgreSQL Optimizer will perform sequential scans when resolving a SQL query. Database Administrators can sometimes force the Optimizer to use an index by turning this feature on or off accordingly. This can be done from within the *psql* utility by issuing a SET command, referencing the appropriate run-time parameter. Continuing with our earlier example of joining the two tables *titles* and *publishers*, a Database Administrator could set a run-time parameter to disable sequential scans performed by the PostgreSQL Optimizer, and force it to use the primary key index. The example below disables sequential scans, wherever possible, during the current *psql* session, giving the following revised explain plan results.

```
$ psql testdb
testdb=# SET ENABLE_SEQSCAN TO OFF
testdb=# EXPLAIN select t.title_id, t.title, p.pub_id, p.pub_name, p.city, p.state,
testdb=#      p.country from titles t, publishers p where t.pub_id = p.pub_id;

NOTICE:  QUERY PLAN:

Merge Join  (cost=100000003.29..100000008.93 rows=36 width=96)
  ->  Index Scan using publishers_pk on publishers p  (cost=0.00..5.09 rows=8 width=60)
  ->  Sort  (cost=100000003.29..100000003.29 rows=36 width=36)
        ->  Seq Scan on titles t  (cost=100000000.00..100000002.36 rows=36 width=36)

EXPLAIN
testdb=#
```

Setting the run-time parameter ENABLE_SEQSCAN to off did force the use of the primary key on *publishers*. Again reading the generated output from the bottom, the processing step shows the data was retrieved differently by going to the *titles* table first. Since no suitable index is available within the *titles* table, the table is sequentially scanned and then sorted by *pub_id* to order the rows for the join. The next step shows an index scan of the *publishers* table, which uses the primary key (*publishers_pk*). The leveling of the tree structure within the explain plan also reveals the index scan of the *publishers* table and the sort/scan operation performed on the *titles* table are done in parallel. The matching rows between the two tables are joined by their *pub_id* values and merged into a “hash” table in memory, giving the final results of the query.

The estimated costs reveal how the ENABLE_SEQSCAN run-time parameter really works. Setting the ENABLE_SEQSCAN to off adds a large constant to the estimated costs, so that the query planner will not choose the sequential scan, unless no other alternative exists. The low-cost "Hash Join", which was previously used in the first example, is now rejected because its estimated costs include two costly sequential scans of the *publishers* and *titles* tables.

If an index existed on the column *pub_id* in the *titles* table, the explicit sort could be eliminated reducing the overhead costs reflected in the above explain plan. Since there may be other queries on the *titles* table that select by *pub_id*, the Database Administrator can experiment with adding a new index to see if the overhead costs can be reduced. The following explain plan shows the impact of adding this new index on the *titles* table:

```
NOTICE:  QUERY PLAN:

Merge Join  (cost=0.00..14.03 rows=36 width=96)
  ->  Index Scan using publishers_pk on publishers p  (cost=0.00..5.09 rows=8 width=60)
  ->  Index Scan using titles_pubid_idx on titles t  (cost=0.00..8.40 rows=36 width=36)

EXPLAIN
```

Notice in the above explain plan output, the query planner believes the "Merge Join" will be almost three times slower than the original "Hash Join" from the first example (i.e. 14.03 cost units versus 5 cost units). If the ENABLE_SEQSCAN is reset to on, the query planner would go back to using the "Hash Join" retrieval method shown in the first example, despite the presence of the new index. This demonstrates why Database Administrators and Developers need to carefully analyze the impact of making database changes, using a tool like the *EXPLAIN* utility. However, the query planner's cost estimates are based on statistical "guesses" and simple cost models that may not always correspond to the actual run-time conditions of your processing environment. Thus, the end results of conducting an experiment like this may determine the "Merge Join" in the last example is actually the faster retrieval method for satisfying the query.

Other run-time parameters that can be adjusted are outlined in the following table. Although this is not an exhaustive list, it does reflect the most significant run-time parameters that impact how the PostgreSQL Optimizer processes database queries.

PostgreSQL Run-Time Parameters	***Description of Run-Time Parameter Usage***
ENABLE_HASHJOIN	Enables or disables the use of hash-join plan types. Default setting is set to ON.
ENABLE_INDEXSCAN	Enables or disables the use of index scans. Default setting is set to ON.
ENABLE_MERGEJOIN	Enables or disables the use of merge-join plan types. Default setting is set to ON.
ENABLE_NESTLOOP	Enables or disables the use of nested loop joins when processing a query. Disabling nested loop joins will not entirely suppress this occurrence, but it will force the Optimizer to use other methods when they are available. Default setting is set to ON.
ENABLE_SEQSCAN	Enables or disables the use of sequential scans when processing a query. This will not eliminate sequential scans, but it will force the Optimizer to consider other approaches like using an index or rewriting the query, as shown in the above examples.
ENABLE_SORT	Enables or disables the use of sorts. This is helpful if you know data is already indexed in sorted order. Thus, using an index may circumvent a sort. Default setting is set to ON.
GEQO	Special feature within PostgreSQL that eliminates the need for the Planner to perform an exhaustive search for a viable retrieval path, based upon a mathematical algorithm. The parameter name stands for *Genetic Query Optimization*. Default setting is set to ON. Refer to PostgreSQL Developer's Guide for further details.
GEQO_RELS	Integer specifying a threshold value for when to use the *Genetic Query Optimization* (GEQO) method. The value instructs the Optimizer to only use the GEQO method, when processing the same number of relations as specified by this parameter. If value is set to 5, then the GEQO method will only be used when there are five or more tables/relations involved in a complex query.
KSQO	Parameter stands for *Key Set Query Optimizer* (KSQO). It is used to instruct the Optimizer to convert queries that contain multiple AND's along with OR's into UNION queries. This can significantly improve ODBC queries generated from tools like Microsoft Access, which tend to execute queries of this form.

The examples shown in this section demonstrate how PostgreSQL can be adjusted using a number of techniques to get the optimum performance from the database. However, it also shows why Database Administrators must use tools like the *EXPLAIN* utility to thoroughly analyze what is the right solution for a database performance problem. The information gathered from explain plans and monitoring the impact of adjusting the PostgreSQL run-time parameters provide invaluable feedback for Database Administrators. This feedback will also assist your staff in configuring the right database environment for running your business applications using PostgreSQL. The key point here is that the PostgreSQL database provides flexibility and scalability to accommodate diverse business environments.

6 Database Tools and Utilities

The open source community has made significant contributions in developing various tools that supplement the features available within the PostgreSQL database software. Since the PostgreSQL environment supports many different application interfaces, there are several choices in the type of tools one can use. PostgreSQL provides API libraries for C, C++, Tool Command Language/Graphical User Interface Toolkit (Tcl/Tk). In addition, PostgreSQL provides fully compliant ODBC and JDBC drivers for building GUI applications written in commercial languages like Delphi, Visual Basic, PowerBuilder, and Java. Other third-parties have built API extensions for accessing PostgreSQL using Perl, PHP, and Python. Depending upon your server and desktop operating environments, one or a combination of these tools may be appropriate for your organization.

Although this section does not cover all of the PostgreSQL tools that are available, it does feature some of the more widely adopted ones. As these tools mature and developer alliances are formed with Great Bridge, the intent will be to incorporate a suite of tools in the PostgreSQL product offerings from Great Bridge.

PostgreSQL Command Line SQL Interface Tool (psql)

The standard tool for accessing PostgreSQL databases from within a server environment is the *psql* utility. It is a command line interface, which allows users to execute SQL statements against a PostgreSQL database. In addition, it provides several user functions for obtaining information about database objects defined within the database. The *psql* utility supports several text editors for editing commands entered during a *psql* session. These include vi, emacs, pico, jed, and kedit. The default editor is vi, but this can be overwritten by specifying another editor using one of the following environment variables: PSQL_EDITOR, EDITOR, and VISUAL.

The *psql* executable is located in the */usr/bin* directory. It can be invoked from the operating system prompt, passing the database name of the PostgreSQL database that you want to access. The example below shows how to invoke the *psql* tool and what is displayed when first entering the *psql* environment.

```
$ psql testdb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=#
```

The *psql* prompt will always reflect the name of the database that is being accessed for the current session. Informational messages are provided to advise users of the commands that can be executed within the *psql* environment. The "\h" command provides online help for all the SQL commands that are available from within PostgreSQL. For example, you can view the SQL syntax for creating a table by typing *"\h CREATE TABLE"* at the *psql* prompt. To exit out of a *psql* session, simply type the *"\q"* command.

The *psql* tool provides several built-in functions that can be invoked using a meta-command, which is prefixed by a backslash (i.e. "\"). Meta-commands provide easy

shortcuts for accessing information about the database environment, using simple keystrokes comprised of a backslash ("\") and an alphabetic character. A complete list of all these built-in functions can be retrieved by typing the command "\?" at the *psql* prompt. Some of the most useful meta-commands are listed in the following table.

Psql Meta-Commands	***Functional Description of Meta-Command***
\c or \connect	Enables user to connect to another PostgreSQL Database from within the *psql* tool (i.e. \c sales_db).
\d relation-name	Describes all of the related columns for the specified relation, which could be a table, view, or index. If no relation is specified, this command displays all of the defined database objects that exist within the current database.
\df [pattern]	Lists all of the functions that have been defined within the current database. The [pattern] is an optional argument, which you can pass as an argument to the command. This will search all defined functions that contain the letters specified (e.g. \df abs – Displays all functions containing abs in their names).
\distvS	This command is a variation on the "\d" command above. Any of the following letters can be specified to show a listing of all (i)ndices, database (s)equences, (t)ables, (v)iews, or (S)ystem Tables. To list only defined tables, you would type \d at the *psql* prompt.
\e or \edit	Invokes the default text editor allowing the user to edit a specified file, or the last SQL statement issued within the *psql* session.
\i filename	Instructs *psql* to read the specified file as input, executing the statements as if the user entered them. Useful for executing SQL scripts that have been previously saved in a text file.
\l	Lists all the PostgreSQL Databases that are running on the server. Helpful tool for Database Administrators and developers who want to know what databases exist on a server.

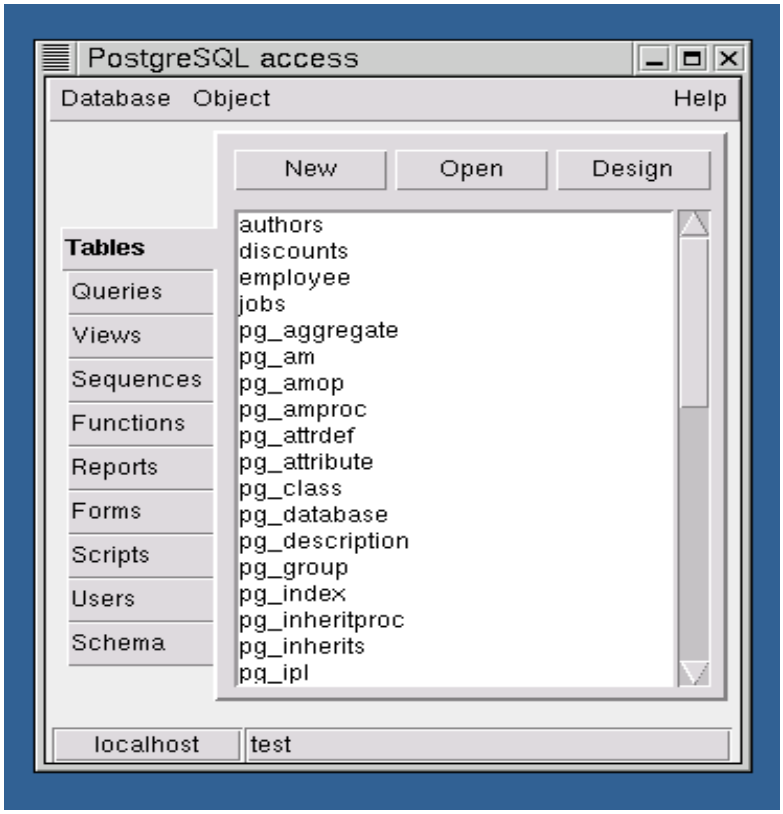
Psql Meta-Commands	***Functional Description of Meta-Command***
\o filename	Directs future query results to an output file, formatting the output by separating columns with the pipe ("|") symbol.
\z [pattern]	Lists all tables defined within the current database, along with their assigned permissions. If arguments are passed to this command, it will show any table that contains the specified letters. This is extremely helpful if you only know part of the table name, but cannot recall the exact spelling of it. For example, there may be two tables called *titles* and *title_authors*. These could be listed if you entered the command "\z tit".
\! [command]	Allows user to execute an operating system command from within the *psql* environment, or escape out to the operating system environment. If no command is specified, you are temporarily spawned to an operating system prompt. You may type "exit" to return to the *psql* environment.

When comparing the *psql* tool to other database interfaces like Oracle's SQL*Plus utility, PostgreSQL's *psql* interface offers significantly more features like the built-in meta-commands and online help facility. Both of these features enable users to easily obtain information about the PostgreSQL database environment, along with getting assistance on standard SQL statement syntax. Thus, the *psql* tool accommodates both novice and expert users, allowing organizations to quickly become productive in a PostgreSQL environment.

PgAccess Database Tool

PgAccess provides a graphical interface for accessing and maintaining PostgreSQL databases from multiple platforms. The tool is available from the open source community and was originally developed by Constantin Teodorescu who lives in Romania. The *PgAccess* source code and executables can be downloaded off the Internet by referencing the following URL address: www.flex.ro/pgaccess/index.html. PgAccess is written in Tcl/Tk, which is an open source programming language that runs on several operating environments including Linux/Unix, Windows, Macintosh, and AS400. As a result, *PgAccess* can be deployed in most any business environments that require access to PostgreSQL backend databases.

PgAccess was primarily designed to be an access tool for PostgreSQL, as its name implies. However, the tool also provides a host of administrative functions that may be useful to Database Administrators and/or Developers, who are responsible for defining and maintaining PostgreSQL database objects. The *PgAccess* application environment provides ten menu tabs along the left-hand side of its display Window as shown below.



These menu tabs correspond to specific database objects, which can be accessed and modified using the *PgAccess* tool. Along the top of the application window, there are three buttons labeled New, Open, and Design that serve as launching points for accessing the various database objects listed along the menu tabs. The *PgAccess* menu options are outlined below:

- *Table Menu Tab* – Provides the capability of displaying all of the tables, which are defined within the current PostgreSQL database. When this menu tab is selected, a list of all the database table objects is displayed, allowing the user to specify the type of operation they want to perform (e.g. Create New table, Open table for viewing the data, or Design table for modifying the selected table structure). The user can hit the New Button if they want to create a new table, which invokes a pop-up data entry window for entering various attributes associated with the new table. The Open Button launches a data viewer window, which executes a SQL SELECT statement against the specified table. If the user decides to hit the Design Button, the user can make minor modifications to the exiting table. Thus, both maintenance and query activities can be performed on database tables, using this *PgAccess* Menu Tab.
- *Query Menu Tab* – Enables users to create new database queries, open existing queries that have previously been saved, and/or modify an existing database query. This is the primary function used within *PgAccess* for accessing data from a PostgreSQL database. Within the *Query* function, there is a GUI *Query Wizard* that can be used to generate database SQL queries. Once queries are designed and built, they can be saved within the *PgAccess* environment for later use. In addition, database Views can be created, which are based upon queries defined within this menu option.

- *View Menu Tab* - Provides the capability of accessing database Views, which are defined within the current PostgreSQL database. When this menu tab is selected, a list of defined Views is displayed. This allows users to specify the type of operation they want to perform (e.g. Create New View, Open View to display the data that is returned from the underlying SQL statement, or Design View for modifying the View's underlying SQL statement). The user can hit the New Button if they want to create a new View, or use an existing database query saved within *PgAccess* to define a View. The Open Button launches a data viewer window, which executes the underlying SQL statement defined within the View. If the user decides to hit the Design Button, the user can make minor modifications to the underlying SQL statement defined within the View. Thus, both maintenance and query activities can be performed on database Views, using the *PgAccess* View menu option.
- *Sequence Menu Tab* – Provides the capability of defining, viewing, and modifying sequence objects within the current PostgreSQL database. Sequence objects are database structures that maintain and generate random number sequences used within a PostgreSQL application environment. For example, a table may be defined with an attribute that uniquely identifies the existence of a table row like a transaction-id. A sequence object can be used to generate random numbers, which will be used to populate the transaction-id column whenever a new row is inserted into the table.
- *Function Menu Tab* – Provides a utility for building and maintaining database functions that are referenced within the PostgreSQL database. This *PgAccess* utility supports the writing of functions in one of several programming languages including SQL, PL/pgSQL which is a procedural language built within PostgreSQL, PL/Tcl which is a Tcl/Tk to PostgreSQL API, and ANSI Standard C. All user-defined functions defined within this utility are stored within the PostgreSQL database.
- *Report Menu Tab* - Provides a facility for defining and maintaining reports, which can be used for browsing select PostgreSQL database objects. This utility is based upon the Tcl/Tk open source programming language. Thus, users must have some working knowledge of Tcl/Tk to successfully build reports using this menu option.

- *Form Menu Tab* – Provides a facility for defining and maintaining data entry forms that can be used to insert data into PostgreSQL database tables. This utility is based upon the Tcl/Tk open source programming language, which is an excellent tool for building applications within the PostgreSQL environment. Users must have some working knowledge of Tcl/Tk to successfully build forms using this menu option.
- *Script Menu Tab* – Provides a facility for defining and maintaining user-defined SQL scripts that can be called to perform various database operations.
- *User Menu Tab* – Provides a utility for creating and modifying database users that can access the PostgreSQL database. Database user passwords can also be reset from within this menu option.
- *Schema Menu Tab* – Provides a utility for defining a new database instance within the PostgreSQL environment. In addition, this utility provides the ability to copy existing database table structures into the new database schema environment.

In addition to the above menu options, the *PgAccess* File Menu provides three administrative utilities that can be used to perform several database operations like a PostgreSQL *vacuum*, data imports, and data exports. The import and export utilities enable users to migrate data to and from PostgreSQL databases, using external flat files as input and/or output. Both utilities enable users to specify the delimiter characters that are used to separate data fields. All of these utilities, including the main menu tabs within *PgAccess*, generate standard SQL statements to perform these functions within the PostgreSQL backend database.

The *PgAccess* tool is an excellent client interface for managing and accessing PostgreSQL databases within multi-platform environments. It provides a portable tool for businesses to easily integrate PostgreSQL into their existing operating environments. Both Database Administrators and Developers will appreciate the flexibility offered by *PgAccess* for building and maintaining PostgreSQL applications using its various database tools. *PgAccess* is also an excellent prototyping tool for quickly generating database applications.

PgAdmin Database Administration and Query Tool

The *pgAdmin* tool is a Windows based application that offers an entire suite of functionality for accessing and maintaining PostgreSQL database objects. The tool was originally developed by Dave Page who lives in the United Kingdom and is available for download at no charge using the following URL address: www.pgadmin.freeserve.co.uk/. The *pgAdmin* application was written in Visual Basic V6.0 using an ODBC interface to PostgreSQL. The latest PostgreSQL ODBC driver can be downloaded from the PostgreSQL Website using the following URL address: ftp://ftp.postgresql.org/pub/odbc/index.html. The application was primarily designed to be a database administration tool. However, it can also be extremely useful for the average user who just wants to perform ad hoc queries against the database.

The *pgAdmin* functionality is subdivided into five basic menu options, which are outlined below:

- *Schema Menu* – Provides the ability to view information about all of the PostgreSQL Databases that are running on a specific server, and their associated objects including Tables, Indices, Views, Triggers, Functions, Languages defined within the database, and Privileges. The *pgAdmin* tool enables users to create, rename, add attributes, delete, and query all of the database objects mentioned above.
- *System Menu* - Provides the capability to perform several administrative tasks such as running the PostgreSQL *vacuum* utility, adjusting database tuning parameters, creating new database users, creating database groups, and assigning users to groups.
- *Tools Menu* – Provides several tools that enable users to execute ad hoc SQL queries against a PostgreSQL database, perform selective imports of data from an external file into an existing database table, and perform selective exports of data from a table to an external flat file. In addition, it provides a database migration tool for selectively moving data from other data sources into a PostgreSQL database environment. It supports migrating data from any ODBC compliant data source into PostgreSQL, including popular desktop tools like Microsoft Excel and Microsoft Access databases.

- *Utilities Menu* – Provides the capability for launching the Windows ODBC Data Source Administration Tool for changing ODBC driver configurations, and/or other data source parameters. In addition, it provides a *psql* window that launches the interactive PostgreSQL *psql* utility running on the database server, and a Remote Procedure Call (RPC) utility for executing server operating system commands on the database server. As previously mentioned, the *pgAdmin* application can export data from PostgreSQL into flat files, but it also can export data generating output in either HTML or Excel formats. Other export utilities can be loaded into the *pgAdmin* environment by using a tool called the Export Manager. The Export Manager installs or uninstalls data exporters that are available using Dynamically-Linked Library (DLL) Plug-Ins.
- *Reports Menu* – Provides several different report formats for displaying and printing out information about various database objects that exist within a PostgreSQL database. Reports are designed to accept selection criteria for specifying database objects and/or objects owned by particular database owners. All of the reports can be viewed online, or sent to a printer device.

Like some of the other PostgreSQL GUI Tools, the *pgAdmin* application generates standard SQL statements to perform the various functions outlined above. However, it provides a display window at the bottom of the application window, which shows each SQL statement that is sent to the PostgreSQL database server. Any SQL statements that are generated by *pgAdmin* can be copied, saved, and reused in the *pgAdmin* SQL utility.

The *pgAdmin* tool is an excellent client interface for managing and accessing PostgreSQL databases within a Windows environment. It enables businesses that have already made investments in Windows platforms to utilize their existing infrastructure to access PostgreSQL database environments. PostgreSQL tools like *pgAdmin* also demonstrate how simple it can be to integrate PostgreSQL into existing operating environments.

PostgreSQL Support for Data Modeling Tools

PostgreSQL provides both Open DataBase Connectivity (ODBC) and Java DataBase Connectivity (JDBC) API Interfaces to PostgreSQL, which are compliant with the latest industry standard specifications for both interfaces. As a result, any data modeling tool that is ODBC and/or JDBC compliant should be able to access the PostgreSQL environment. One common data-modeling tool that uses this type of technology is ERwin, which is commercial product offered by Computer Associates, Incorporated (Additional information on ERwin is available on the Internet at http://www.cai.com/products/alm/erwin.htm). Many Database Administrators and Data Architects use ERwin to analyze and graphically design relational databases.

Using the PostgreSQL ODBC Interface with the ERwin tool enables Database Administrators to access PostgreSQL databases and retrieve information about their data structures. There are several features within ERwin that enable database structures to be reversed-engineered from a physical database, generating a graphical model of the database. Data models can be modified by adding, changing, or deleting various data structures using ERwin. Once these modifications are complete, ERwin provides a facility for generating standard SQL Data Definition Language (DDL) scripts that reflect the modified structures defined within the data models. The generated scripts can be written to an output file for later use, or ERwin can interactively reapply the database changes within PostgreSQL, using the PostgreSQL ODBC Interface.

Capitalizing on the open architecture of the PostgreSQL environment and the features available within ERwin, ITS staff members can integrate many disparate environments into one comprehensive repository of information. For example, personnel data residing in a third-party package running Oracle can be shared with a departmental application housing information in a Microsoft SQL Server environment. Both of these data sources can be integrated into a single data warehousing application that runs on a PostgreSQL database. The key to this type of integration is to have tools and database environments that are both flexible and architecturally open. PostgreSQL possesses many of these characteristics, making it a viable integration solution for complex business environments.

PostgreSQL Upgrade Tool

The PostgreSQL software provides a utility called *pg_upgrade*, which will assist in upgrading existing PostgreSQL databases to newer releases of the database, without reloading data. The *pg_upgrade* utility requires the Database Administrator to perform two preliminary operations prior to running the utility. The first requirement is to establish a backup file of your existing PostgreSQL database that only contains the schema definitions with no data. This can be accomplished by running the *pg_dump* or *pg_dumpall* utilities, using the ("-s") option (See Chapter 3 - Database Backup and Restore of this document for more details). The second requirement involves renaming the existing database directory where the physical database files reside to an alternate area (i.e. mv $PGDATA/db-directory $PGDATA/old-db-directory).

Once a backup file of the existing database is created and the physical database area is renamed, the new PostgreSQL software can be installed and configured. The upgrade step will involve launching the *pg_upgrade* utility, passing the backup file as an input argument, and specifying the renamed database directory. The example below shows how to execute the *pg_upgrade* utility passing these input arguments.

$ pg_upgrade –f $PGDATA/backups/db_backups.out $PGDATA/old-db-directory

The *pg_upgrade* utility will read the backup file *db_backups.out* to restore all of the databases and their associated structures into the new PostgreSQL environment without the data. The final task will be to copy the physical database files containing the data to the new target database. When the upgrade operation completes, all of the existing databases should be upgraded to the new version of PostgreSQL. The renamed directory can be deleted, once the Database Administrator confirms the databases have been successfully upgraded.

Database upgrades often involve tedious planning and testing before actually performing an upgrade. The intent behind the *pg_upgrade* utility is to assist the Database Administrators in transitioning their existing databases to a newer release of PostgreSQL.

This will also ensure organizations can take advantage of the latest features of PostgreSQL when they become available for distribution.

7 Conclusions

PostgreSQL is a mature open source database that is the product of 23 years of development by some of the best software developers in the industry. The quality and functionality of the PostgreSQL database makes it the most advanced open source database server available today. As demonstrated throughout our discussions of the PostgreSQL administrative features, the database provides both flexibility and scalability in deploying applications within the PostgreSQL environment. There are many tools and utilities available within PostgreSQL that help Database Administrators and System Managers manage and tune PostgreSQL. As new releases of PostgreSQL become available, Great Bridge will provide the infrastructure necessary to support commercial businesses and their ITS staff with integrating PostgreSQL into their organizations.

Great Bridge is committed to making open source software viable and being the leading commercial provider of PostgreSQL database products, services, and support. By combining the advanced features of PostgreSQL with this support infrastructure, Great Bridge is confident of its ability to demonstrate the business advantages of deploying PostgreSQL database solutions within the commercial marketplace. Both the open source community and Great Bridge truly believe PostgreSQL can fundamentally change how database solutions are implemented in the future, giving businesses a competitive edge. This belief is best summarized in Great Bridge's mission statement:

> *"Great Bridge, LLC will be the leading commercial provider of open source solutions powered by PostgreSQL, the world's most advanced open source database. Great Bridge will deliver value-added open source software and support services based on PostgreSQL, empowering e-Business builders with an enterprise-class database and tools at a fraction of the cost of closed, proprietary alternatives."*

Appendix A - PostgreSQL Reference Web Sites

The following list provides a summary of online references that are available on the Internet, covering topics related to PostgreSQL and Linux/Unix operation systems. Recommend visiting the Great Bridge Website at www.greatbridge.com for the latest news regarding PostgreSQL software.

PostgreSQL Database and Related Product References

Friberg, P., Hellman, S., Templeton, M. (1999, September 21). Managing Earthquake Data With PASSCAL Database Tools – Appendix B PostgreSQL Installation Instructions. [Online]. Available: http://www.passcal.nmt.edu/manuals/smallfont/apb.html

Owen, Lamar (1999, November 5). Map of a RPM PostgreSQL Installation. [Online]. Available: http://www.ramifordistat.net/postgres/redhat_map.html

FAQs on PostgreSQL and Related Access Tools. [Online]. Available: http://www.postgresql.org/users-lounge/docs/faq.html
http://laplace.snu.ac.kr/~pos7hink/Interests/Database/Database-HOWTO.html#toc34
http://laplace.snu.ac.kr/~pos7hink/Interests/Database/Database-HOWTO-34.html

Layered Software Components for PostgreSQL related Tools. [Online]. Available:
http://gatekeeper.pa.dec.com/pub/BSD/NetBSD/packages/pkgsrc/databases/pgaccess/
http://gatekeeper.pa.dec.com/pub/BSD/NetBSD/packages/pkgsrc/databases/postgresql/

Open Source and PostgreSQL Mailing List Archives (Indexed by Topic). [Online]. Available:
http://www.geocrawler.com/lists/3/Databases
http://www.geocrawler.com/lists/3/Gnome
http://www.geocrawler.com/lists/3/Red-Hat-Linux

PgAccess Configuration Help. [Online]. Available:
http://www.postgresql.org/mhonarc/pgsql-docs/1999-08/msg00036.html

PostgreSQL Documentation Resources. [Online]. Available:
http://www.postgresql.org/docs/index.html
http://www.postgresql.org/docs/postgres/part-developer.htm

PostgreSQL V7.0.2 RPM File References. [Online]. Available:
ftp://ftp.postgresql.org/pub/binary/v7.0.2/RPM/
ftp://ftp.postgresql.org/pub/binary/v7.0.2/RPM/README

Zeos Library of Interactive Database Tools for PostgreSQL. [Online]. Available:
http://www.zeos.dn.ua/eng/index.html

Linux/Unix Operating System Tips & Configuration References

GNOME Project References. [Online]. Available: http://www.gnome.org/

Linux HOWTOs Index – Linux & PostgreSQL Quick Installation Instructions. [Online]. Available:
http://garbo.uwasa.fi/ldp/HOWTO/HOWTO-INDEX/index.html
http://garbo.uwasa.fi/ldp/HOWTO/PostgreSQL-HOWTO.html
http://garbo.uwasa.fi/ldp/HOWTO/Config-HOWTO.html#toc4

RedHat Linux Support Database. [Online]. Available:
http://www.redhat.com/apps/support/

RedHat Support – Using RPM. [Online]. Available:
http://www.redhat.com/support/manuals/RHL-6.0-Manual/install-guide/manual/doc074.html

Miscellaneous Resources on Related Linux/Unix & PostgreSQL Topics:

Linux World References to Open Source & PostgreSQL. [Online]. Available:
http://www.linuxworld.com/linuxworld/lw-2000-05/lw-05-database_p.html
http://www.linuxworld.com/linuxworld/lw-1998-12/lw-12-linux101_p.html

SEVA Incorporated – Porting Access97 to PostgreSQL. [Online]. Available:
http://www.sevainc.com/Access/index.html