pgUnitTest

Help

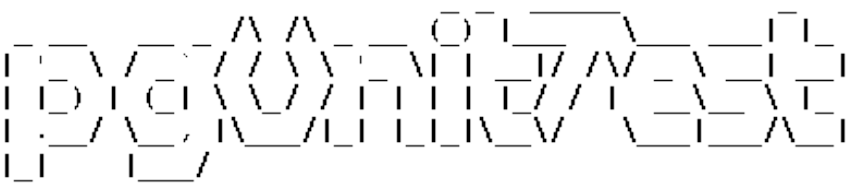
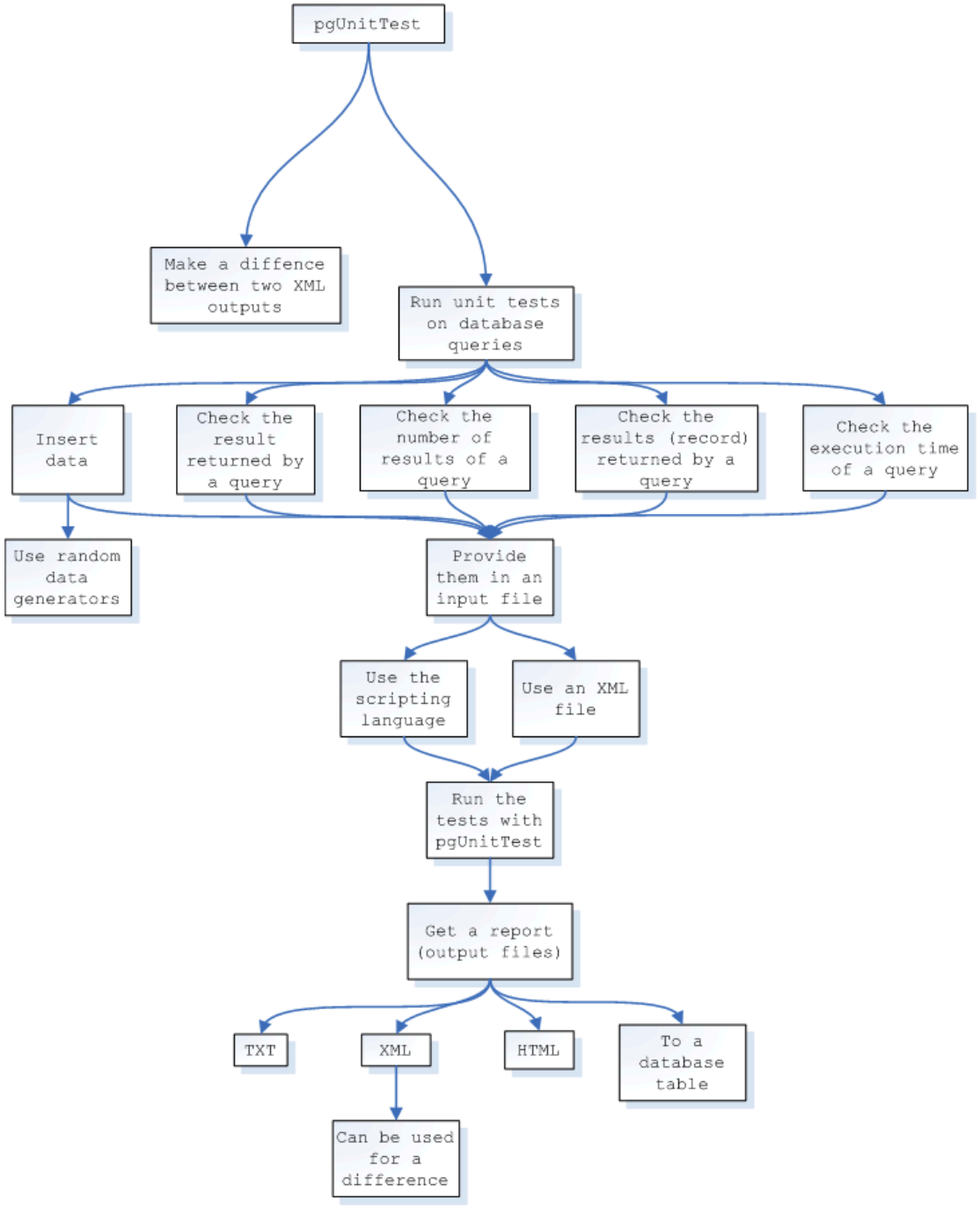


TABLE OF CONTENTS

Overview


pgUnitTest
Make a diffence between two XML outputs
Run unit tests on database queries
Insert data
Check the result returned by a query
Check the number of results of a query
Check the results (record) returned by a query
Check the execution time of a query
Use random data generators
Provide them in an input file
Use the scripting language
Use an XML file
Run the tests with pgUnitTest
Get a report (output files)
TXT
XML
HTML
To a database table
Can be used for a difference

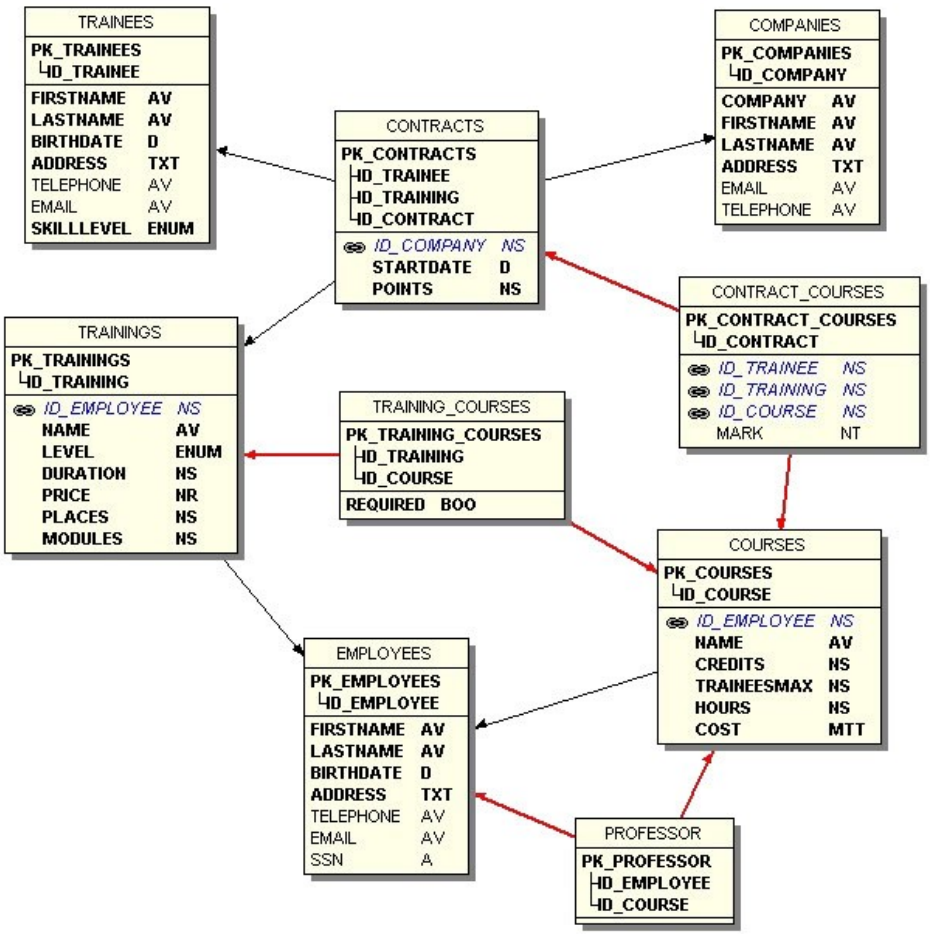
Example of use

With `pgUnitTest` you can run unit tests on database queries. There are two types of unit tests:

- `insert` unit tests meant to insert data or randomly generated data,
- `select` unit tests to check queries, execution time...

We consider that the `PostgreSQL` database is on `localhost`, the user name is `postgres`, and the password is `postgres` too.

Example database



Create the database

The table structure is in the help folder of `pgUnitTest`: `help/example/training.sql`.

On the command line use `psql`:

```
psql -U postgres
```

Create the database:

```
CREATE DATABASE training;
```

Exit from the interactive terminal:

```
\q
```

Import the data from `training.sql` to create the tables:

```
psql -U postgres -d training -f help/example/training.sql
```

The tables should be ready for the following tests.

Run unit tests

The unit tests can be provided with a script file or with a XML file. Let's see how we can test our `trainees` table in the `training` database:

- Insert 10,000 trainees with randomly generated data,
- Check the percentage of null telephone numbers,
- Check that the skill level is included in the expected range,
- Check if two trainees have the same identity (first name and last name),
- Check if the telephone and the email of the first two trainees are the ones expected.

In order to do that it is advised to, first insert the 10,000 trainees, and then check the results.

Insert the trainees

Put in a text file (`data.pgu`):

```
ROLLBACK { FALSE }

UNITTEST ('Insert trainees') {
    EXECUTE
        FOR 1 TO 10000 {
            INSERT INTO trainees
                (firstname, lastname, birthdate, address, telephone,
                 email, skilllevel)
            VALUES (
                -- id_trainee serial NOT NULL => automatic
                -- firstname character varying(255) NOT NULL
                '%{name(0)}%',
                -- lastname character varying(255) NOT NULL
                '%{name(0)}%',
                -- birthdate date NOT NULL
                '%{date(1970-01-01, 1980-01-01, 0)}%',
                -- address text NOT NULL
                '%{address(0)}%',
                -- telephone character varying(255) => 20% of null values
                '%{string("[0-9]{10}", 20, 12345)}%',
                -- email character varying(255) => 20% of null values
                '%{string("[a-z0-9]{6,12}@[a-z]{6,12}.com", 20)}%',
                -- skilllevel smallint NOT NULL between 0 and 8
                '%{smallint(0, 8, 0)}%'
            );
        }
    WITH
        TEST_INSERT;
}
```

We consider that `data.pgu` is in the `pgUnitTest` directory.

Run the test (the `insert`) with the following command line, in the `pgUnitTest` directory:

```
java -jar pgUnitTest.jar -s data.pgu -f html -o data.html -b
"//localhost/training/postgres/postgres"
```

This test should succeed, as you can see in `data.html` (the output file in HTML format).

Check some results

Put this in a text file (`test.pgu`):

```
ROLLBACK { FALSE }

UNITTEST ('Percentage of null values for telephone column') {
```

```
    EXECUTE
        SELECT (count(*) * 100 / 10000) as mean
        FROM trainees
        WHERE telephone IS NULL;
    WITH
        TEST_RESULT;
    CHECK
        20;
}

UNITTEST ('Skill level in range') {
    EXECUTE
        SELECT skilllevel
        FROM trainees
        WHERE skilllevel < 0 OR skilllevel > 8;
    WITH
        TEST_ROWSCOUNT;
    CHECK
        0;
}

UNITTEST ('Two trainees with the same identity (firstname, lastname)') {
    EXECUTE
        SELECT * FROM (
            SELECT firstname, lastname, count(*) AS cnt
            FROM trainees
            GROUP BY firstname, lastname
            HAVING count(*) >= 2
        ) c;
    WITH
        TEST_ROWSCOUNT;
    CHECK
        0;
}

UNITTEST ('Complex record: the test fails') {
    EXECUTE
        SELECT telephone, email FROM trainees WHERE id_trainee <= 3;
    WITH
        TEST_RESULTS;
    CHECK
        telephone, email
        ('8732346179', NULL)
        ('7436831518', 'futxzs@cefcqgar.com');
}
```

Run the tests (the `select`) with the following command line, in the `pgUnitTest` directory:

```
java -jar pgUnitTest.jar -s test.pgu -f html -o test.html -b
"//localhost/training/postgres/postgres"
```

You can see the report in `test.html` (the output file in HTML format). The first two tests should succeed but not the last two ones.

More tests

A `data.pgu` and a `test.pgu` files are bundled with `pgUnitTest` in the `help/example` directory. These tests are meant to be run on the complete `training` database. But, first you need to recreate the database in order not to have any data in it.

Scripting language

There are five types of tests, as shown on the diagram in the overview. The tests are performed on database queries. To run tests `pgUnitTest` uses a scripting language.

The scripting language is **case-insensitive**. The `pgu` extension is advised. To edit a script you can use any text editor. SQL syntax highlighting is good as the major parts of keywords are SQL ones.

Like in SQL, `--` can be used to comment a line.

The structure of a `pgu` file is this one:

```
ROLLBACK { TRUE|FALSE }

UNITTEST [('Name of the test')] {
          EXECUTE
                    INSERT OR SELECT STATEMENT;
          WITH
                    TEST TYPE;
          [ CHECK
                    RESULT EXPECTED; ]
} *
```

Where `*` means that you can put as many tests as you want.

For each type of test:

- The expected result is in the `check` section,
- The test type is in the `with` section,
- The query is in the `execute` section,

Each section ends with a semicolon `;`.

Rollback option

When preparing the tests it is useful not to insert the data. You can set the `rollback` option, at the beginning of the file to `true`. Then, every insert operation will be canceled and no data will be inserted.

This option is **required**.

To really insert data, this option must be set to `false`.

Test name option

The name of the test is between quotes. It is **optional**.

Check the result unit test

```
UNITTEST ('This test checks a result') {
          EXECUTE
             SELECT a FROM example WHERE a = "qwerty";
          WITH
             TEST_RESULT;
          CHECK
             'qwerty';
}
```

This test checks whether a query returns an expected result or not.

There can be a modifier after the check in order to be case-insensitive:

```
UNITTEST ('This test checks a result') {
          EXECUTE
             SELECT a FROM example WHERE a = "qwerty";
          WITH
             TEST_RESULT;
          CHECK -i
             'Qwerty';
}
```

The expected result can be:

- An integer **number**,
- A floating number,
- A **string** between quotes (use `''` or `\'` to escape quotes).

The expected string are **trimmed**: `'  qwerty  '` is equal to `'querty'`.

Check the number of rows unit test

```
UNITTEST {
```

```
          EXECUTE
             SELECT * FROM example;
          WITH
             TEST_ROWSCOUNT;
          CHECK
             10;
}
```

This test checks whether the query returns a specific number of rows results or not. The expected number of rows must be an **integer**.

There can be one modifier after the `check`:

- `-g` means the number specified is greater than the expected number of rows,
- `-l` means the opposite.

```
UNITTEST {
          EXECUTE
             SELECT * FROM example;
          WITH
             TEST_ROWSCOUNT;
          CHECK -l
             10;
}
```

This test checks whether the query returns more than `10` results or not.

Check the execution time unit test

```
UNITTEST {
          EXECUTE
             SELECT * FROM example;
          WITH
             TEST_EXECTIME;
          CHECK
             100;
}
```

This test checks the execution time of a query. This time in milliseconds must be lower or equal to the expected one. The expected time must be an **integer** and it represents a time in **milliseconds**.

There is no modifier for this test.

Check the results (records) unit test

This test is similar to another one but deals with complex results (records).

```
UNITTEST {
          EXECUTE
             SELECT * FROM example;
          WITH
             TEST_RESULTS;
          CHECK
             a, b, c
             (1, 'qwerty', 'US')
             (2, 'azerty', 'FR');
}
```

In the test above we get each row and each column from the `example` table. We provide the name of the columns we want to check: `a, b, c`. Then we provide each expected row. In this example there are two rows. It means:

- For the first row `a` must be equal to `1`, `b` must be equal to `qwerty`, `c` must be equal to `US`,
- For the second row `a` must be equal to `2`, `b` must be equal to `azerty`, `c` must be equal to `FR`.

There are three possible modifiers:

- `-i` means that the comparison is case-insensitive (see *Check the result unit test section*),
- `-l` means that the expected rows are included in all the rows returned by the query,
- `-g` means the opposite.

For a null value put `NULL`. Use `''` or `\'` to escape quotes in strings.

Insert unit test

This test is special. There is **no `check` section** (no expected result except that the query must be successful and the data inserted).

An insert unit test can use a single `insert` statement or use a `for` statement to insert a lot of data.

```
UNITTEST {
          EXECUTE
             INSERT INTO example VALUES('1234');
          WITH
```

```
            TEST_INSERT;
}
```

The test above does a simple `insert` into the table `example`.

```
UNITTEST {
          EXECUTE
             FOR 1 TO 1000 {
                INSERT INTO example VALUES('%{string(10,20,5)}%');
             }
          WITH
             TEST_INSERT;
}
```

The test above does a complex `insert`. It loops 1,000 times to insert a randomly generated string into the table `example`.

Go the *random data generators* section to learn more about the random data generators.

Random data generators

How to use the generators

The way to use the generators is to put something like this in the insert statement for the concerned element (the one that has to be filled up with random data):

```
'%{integer(10,20,5)}%'
```

It starts with `'%{` then it continues with the generator name, the parameters between parenthesis and the sequence `}%'` to finish (see example in the *insert unit test* section).

Common parameters

`unique` is `true` or `false` and means whether each value in the provided range must be generated once or not.

`seed` is a long number used to initialize a random generator. Specifying a `seed` means that every time this random generator will be called it will generate the same sequence of data.

`null` means the percentage of null values that have to be generated, between `0` and `100`.

All `min` and `max` parameters are **inclusive**.

Number generators

```
short(min, max, null, seed, unique)
short(min, max, null, unique)
short(min, max, null)
integer(min, max, null, seed, unique)
integer(min, max, null, unique)
integer(min, max, null)
bigint(min, max, null, seed, unique)
bigint(min, max, null, unique)
bigint(min, max, null)
real(min, max, null, seed, unique)
real(min, max, null, unique)
real(min, max, null)
```

```
double(min, max, null, seed, unique)
double(min, max, null, unique)
double(min, max, null)
```

You must specify at least:

- The minimum of the range;
- The maximum of the range;
- The percentage of null values;

The other parameters are optional.

```
serial(start)
serial()
```

A serial generator is a generator which starts with a value and increment this value each time. It is of course unique and cannot have a seed, just a start value, which is `1` if not specified.

```
numeric(precision, scale, null, seed)
numeric(precision, scale, null)
```

- `precision` is the number of digits;
- `scale` is the number of digits after the decimal point (`scale <= precision`);
- Cannot be unique.

Date generators

```
timestamp(min, max, null, seed, unique)
timestamp(min, max, null, unique)
timestamp(min, max, null)
time(min, max, null, seed, unique)
time(min, max, null, unique)
time(min, max, null)
date(min, max, null, seed, unique)
date(min, max, null, unique)
date(min, max, null)
```

The parameters are the same as for the number generators.

- Date: `YYYY-MM-DD` (ex: 1970-11-01 for the 1st of November, 1970);
- Time : `hh:mm:ss` (ex: 14:05:00 for 2:05 pm);

- Time stamp: `YYYY-MM-DD hh:mm:ss`.

Do not quote the parameters.

String generators

```
string(min, max, null, seed)
string("regex", null, seed)
string(min, max, null)
string("regex", null)
```

With the `min` and `max` parameters, it uses the default character set: `[a-zA-Z]` with the specified length (between `min` and `max`).

The regular expression `regex` lets the user specify the pattern of the string to generate. They must be between double quotes.

For example:

```
[a-zABCD]{5,10}
```

You put the list of possible characters between brackets `[ ]`. Either you list the characters `ABCD` or you can put a range `a-z` (meaning `abcdefghijklmnopqrstuvwxyz`).

Then you put the number of characters between accolades `{ }`. On the left it is the minimum length of the string, and on the right the maximum length. To have a string with a specified length you can either put the same value for the minimum and for the maximum or just one value:

`[a-z]{10}` for 10 characters

The characters can be **escaped** with `\`. `[ab\]]` is correct since the first `]` is escaped.

You can create complex regular expressions: `[ab]{1,5}@[cd]{1,5}`.

In the last example you see that you can put a **single character** without a range. The expression above is equivalent to: `[ab]{1,5}[@]{1}[cd]{1,5}`.

Note that you cannot escape a single character. You cannot have `[ab]{1}\][ab]{1}`. Instead you must put `[ab]{1}[\]][ab]{1}` (*or* *`[ab]{1}[\]]{1}[ab]{1}` which is equivalent, the `{1}` is optional*). See the `\]` between brackets.

Another example:

`[a-zA-Z\-_]{10}@[a-zA-Z\-_]{8,10}.com` to generate an email address.

It is impossible to generate unique strings.

Text generators

```
text(min, max, words, null, seed)
text(min, max, words, null)
```

This generator can be considered as a `string(min, max, null[, seed])` with the number of words to generate: `words`. Therefore the strings are composed of `[a-zA-Z]`.

Dictionary generators

```
dictionary(dictionary, null, seed, unique)
dictionary(dictionary, null, unique)
dictionary(dictionary, null)
```

This is a special string generator: `dictionary` is the path to the dictionary where to pick words. So, there must be one word per line. `unique` means each line will be returned once.

In case of a non-null constraint error, check that the dictionary file does not contain an empty line.

Internal generators

```
zip(null, seed)
zip(null)
name(null, seed, unique)
name(null, unique)
name(null)
address(null, seed)
address(null)
```

For:

- A zip code (a special string/regex generator);
- A name (a special dictionary generator);
- An address (a combo of dictionary, regex and number generators).

Reference generators

```
reference(table, column, null, seed, unique)
reference(table, column, null, unique)
reference(table, column, null)
```

They let the user pick the value in a column in a specific table. It is very useful to fill in foreign key constraints. The values will be the ones of `table.column`. The table must have at least one row for he generator to work.

Summary

```
short(min, max, null, seed, unique)
short(min, max, null, unique)
short(min, max, null)
integer(min, max, null, seed, unique)
integer(min, max, null, unique)
integer(min, max, null)
bigint(min, max, null, seed, unique)
bigint(min, max, null, unique)
bigint(min, max, null)
real(min, max, null, seed, unique)
real(min, max, null, unique)
real(min, max, null)
double(min, max, null, seed, unique)
double(min, max, null, unique)
double(min, max, null)
serial(start)
serial()
numeric(precision, scale, null, seed)
numeric(precision, scale, null)
timestamp(min, max, null, seed, unique)
timestamp(min, max, null, unique)
timestamp(min, max, null)
time(min, max, null, seed, unique)
time(min, max, null, unique)
time(min, max, null)
date(min, max, null, seed, unique)
date(min, max, null, unique)
date(min, max, null)
string(min, max, null, seed)
string("regex", null, seed)
string(min, max, null)
string("regex", null)
```

```
text(min, max, words, null, seed)
text(min, max, words, null)
dictionary(dictionary, null, seed, unique)
dictionary(dictionary, null, unique)
dictionary(dictionary, null)
zip(null, seed)
zip(null)
name(null, seed, unique)
name(null, unique)
name(null)
address(null, seed)
address(null)
reference(table, column, null, seed, unique)
reference(table, column, null, unique)
reference(table, column, null)
```

Export formats

The export format corresponds to the `-f` parameter on the command line. It can be one of { `html`, `xml`, `txt`, `db` }.

When `html`, `xml` or `txt` is used, the path to the output file is required for the `-o` parameter on the command line.

When `db` is used it is more complex. `pgUnitTest` is bundled with `pgunitresults.sql` which is a file that you can import into a database. It creates the table `pgunitresults`.

You can export the results of unit tests to this table. For the `-o` parameter, instead of the file path you put a connection string to that table `-o "//host/database/user/password/table"`. `table` should be the table created before: `pgunitresults` as default.

```
java -jar pgUnitTest.jar
         -f db -o "//host/database/user/password/pgunitresults"
         -s input_file.pgu -b "connection_string"
```

Command line parameters

Run pgUnitTest

pgUnitTest uses the Java Runtime Environment. It requires the JRE 1.5 (aka 5.0) but the JRE 1.6 (aka 6.0) is advised for a better output.

To run pgUnitTest, you must be in its directory and use the command line:

```
java -jar pgUnitTest.jar
```

To display the help, enter:

```
java -jar pgUnitTest.jar -h
```

Run unit tests

To run unit tests from an input script:

```
java -jar pgUnitTest.jar -s input.pgu -f html -o data.html -b
"//localhost:5432/database/postgres/postgres"
```

-f is the output format and must be one of {xml, html, txt, db}.

-o is the output file for {xml, html, txt} and is the connection string to the export table for db.

-s in the input script.

-b is the connection string to the PostgreSQL database to test: this includes the host, the database name, the user name and the password to connect to the database.

The **port** in the connection string is **optional**.

Make a difference

You put the parameters:

- -o with the **output HTML file**;

- `-d` with the list of files to compare;
- That's all...

The files to compare must be `pgUnitTest` outputs in XML format.

For example:

```
java -jar pgUnitTest.jar -d input1.xml input2.xml -o out/diff.html
```

Frequently asked questions

Are the unique values really unique?

Not exactly: **they cycle**. If you give an integer range [1;10] and you try to generate 20 values, each value will be generated twice. If you had given [1;20] you would not have had any problem.

It is the same thing for a dictionary: if you try to generate more values than the number of lines in the dictionary it is normal that there will be values that are generated more than once.

I am trying to use the unique values for a real or a double and they seem to repeat!

In this version of `pgUnitTest`, the **values cycle every 524,288 value** generated for a reasonable range. A reasonable range is something like [1.6;1.8]. A not reasonable range is something like [1.6;1.6000000001] where the precision of the internal number is not enough to generate the required number of values (524,288 values).

Why is there no possibility to generate unique random strings?

Imagine the regular expression: `[a-zA-Z0-9_]{10}@[a-zA-Z0-9_]{10}.com`. It can be used to generate an email address.

There are approximately `9.7x10^35` possibilities which is:

- Impossible to generate and pick randomly;
- Difficult to count as the biggest long integer is *only* 92233720368547758071.

However `pgUnitTest` is an Open Source program and **any help or suggestion is really welcome**.