Open-Source Databases, Part II: PostgreSQL

Reuven M. Lerner

Abstract

Feature-rich PostgreSQL delivers on database integrity.

If you're developing an application that depends on storing and retrieving large amounts of data, you undoubtedly have discovered how useful a relational database can be. Databases have been around for a while, but for many years, the open-source offerings were inferior to their commercial counterparts. Today, we have several open-source options from which to choose. Last month, I gave an overview of MySQL, which is probably the most popular open-source, client-server relational database.

This month, we look at MySQL's best-known competitor, PostgreSQL. (Next month, I plan to compare the two programs.) PostgreSQL has a smaller community than MySQL, which sometimes leads people to write it off, think of it as unimportant or even believe that it's not as powerful as MySQL. But, PostgreSQL includes nearly all the functionality of MySQL, and it offers a large number of features that MySQL does not currently offer. As this article shows, it's worth considering PostgreSQL any time you need a database.

History

PostgreSQL began as a research project at the University of California, Berkeley, in 1985. Michael Stonebreaker, a professor of computer science, already had developed and released one database product known as Ingres. After commercializing Ingres, he returned to Berkeley and designed a new database (Postgres), meant in part to solve the problems he had seen with Ingres. Despite a growing community of users, Stonebreaker shut down the Postgres project in 1993.

However, Postgres was distributed under the BSD open-source license, which meant users were free to modify and redistribute it. Two Berkeley students, along with a handful of people from elsewhere in North America, decided to see how easily they could modify the code and distribute a working database.

One of the changes these developers made was in the way Postgres communicated to the outside world. Originally, Postgres had used the QUEL query language that Stonebreaker had designed for Ingres. The developers removed QUEL and added an SQL interpreter to be more compatible with other products. To indicate that this version of Postgres used SQL, the database was renamed PostgreSQL.

Today, many of those original developers continue to work with the project, fixing bugs and contributing patches. The effort is coordinated by a volunteer steering committee. Individual PostgreSQL developers may be hired (as employees or contractors) by various companies, but no one company or organization controls the development or direction of PostgreSQL as a whole.

The most recent release of PostgreSQL is 8.2.3, released in early February 2007. This was a bug-fix release for version 8.2, which was released in late 2006. Major releases typically come out once per year,
with additional minor releases for security and other serious bugs.

## Installation

The easiest way to install PostgreSQL is to use a packaging system, such as debs or RPMs. On my Ubuntu system, for example, I was able to install PostgreSQL easily and quickly with apt-get. Note that most packaging systems distinguish between the PostgreSQL client, server and developer libraries, so be sure to retrieve the packages that are most appropriate for your needs. Installing PostgreSQL via apt-get or RPMs should achieve everything you need to get started—from creating a postgres user to initializing a data directory.

If you must install PostgreSQL from source, I suggest reading the documentation that comes with it. Compiling PostgreSQL is not hard, but it does require more description than I have room for in this article. One piece of advice though, for anyone compiling it from scratch, is to unpack the archive, as well as compile and test it, as the postgres user. Trying to compile and test PostgreSQL as the root user is bound to fail, and other users also might not have sufficient privileges for it to work correctly.

Now, we're ready to start up the server. Prebuilt packages generally will include a shell script (to be placed in /etc/init.d/ or the equivalent) that starts the server for you. Even if you have downloaded and installed the source code for PostgreSQL, you will find an appropriate startup script in the contrib directory. I suggest using (or at least modifying) this script rather than writing one from scratch. On my system, I can start PostgreSQL with:

```
/etc/init.d/postgresql start
```

## Simple Connections

If all goes well, our server should now be running. (We can check this by typing `ps aux | grep postgres` at the command line.) The easiest way to access the server is to use the `psql` interactive client, which comes with PostgreSQL. To get a list of databases in the current cluster, use the following syntax:

```
psql -U <username> -l
```

where `<username>` is a PostgreSQL user with sufficient access privileges on the server. The `-U` option lets us indicate the user name, and the `-l` option asks the server to list those databases that are available, such as:

```
List of databases
 Name    |   Owner    | Encoding
----------+------------+-----------
testserver | reuven     | SQL_ASCII
postgres   | postgres   | SQL_ASCII
template0  | postgres   | SQL_ASCII
template1  | postgres   | SQL_ASCII
(4 rows)
```

Like many database systems, PostgreSQL maintains its own list of users and groups separately from the underlying operating system. To create a new user, we use the `createuser` program, which comes with PostgreSQL:

```
createuser -U postgres reuven
```
This invokes createuser as postgres (which has permission to create other users) and then creates a new user named reuven. If we make this new user a superuser, reuven also will be allowed to create new databases and roles.

Now, we can create a new database in this cluster:

```
createdb -U reuven linux
```

Double-check that the database exists with a reuse of psql -l:

```
psql -U reuven -l
```

You might have noticed that we have not given a password any time we have invoked a command that required a user. The default setting for PostgreSQL makes the server available via a local socket (and thus unavailable over the network). Because only local users will be allowed access, we allow connections from any defined user name, even without a password.

We can change this behavior, as well as other security- and connection-related behaviors, in the pg_hba.conf file located at the top of the cluster directory. The file contains extensive documentation and explains how to set up the connection parameters.

### Creating a Table

Let's connect to our database and see what happens:

```
$ psql -U reuven linux
```

Sure enough, we get the psql prompt, inviting us to enter a query. A list of psql comments is available by typing `\?` at the prompt. We also can get help with SQL syntax and commands with `\h`, as in:

```
\h CREATE TABLE
```

Sure enough, let's start by creating a table:

```
CREATE TABLE People (  
  id    SERIAL    NOT NULL,  
  first_name    TEXT    NOT NULL,  
  last_name    TEXT    NOT NULL,  
  email_address    TEXT NOT NULL,  
  added_at    TIMESTAMP  NOT NULL  DEFAULT NOW(),  
  PRIMARY KEY(id),  
  UNIQUE(email_address)
);  
```

This table has five columns and three different data types. The most common type is TEXT, which is the typical way to store textual data. *PostgreSQL has full support for Unicode, and TEXT columns may contain very long strings*. (Until version 8.0, PostgreSQL could store a limited amount of data in each row, so this might come as news to some people.) There is no built-in support for indexing TEXT columns, but an add-on module, called tsearch2, is relatively easy to install and provides such functionality for anyone who needs it.

The added_at column is of type TIMESTAMP. PostgreSQL provides a very sophisticated set of time-
and date-handling routines, thanks to the division between two basic data types, known as TIMESTAMP and INTERVAL. A timestamp indicates when something happened, whereas an interval tells you how long something took. We thus can subtract two timestamps (to get an interval) or add an interval to a timestamp (to get a new timestamp). Moreover, PostgreSQL lets us define intervals using English-like syntax, such as:

```sql
SELECT id FROM People WHERE added_at > NOW ()
  - interval '30 days';
```

The above query shows us all of the people who were added to our contact database in the last 30 days.

The added_at column defines a DEFAULT value as well. This means that if we fail to INSERT an explicit value for added_at, PostgreSQL will use the current time (at the time of insertion).

**The SERIAL Data Type**

The id column, which we define to be our primary key, uses a SERIAL data type. The thing is, SERIAL isn't a data type at all. Rather, it's syntactic sugar that does several things:

- It creates a new sequence object, whose values are integers that start at 1 and increase each time we ask for a value.
- It defines the column type to be INTEGER.
- It sets the DEFAULT clause for our column to be the result of requesting a new value from the sequence.

This might sound like a complicated way of saying, “SERIAL gives us an auto-incrementing column”. And that's true, in a sense, but you can have as many SERIAL columns as you like in a table, and each sequence can have all sorts of properties associated with it, including its starting point and increment.

Finally, by defining id to be a primary key and email_address to be unique, we implicitly ask PostgreSQL to create indexes on these two columns. When we execute the above query, PostgreSQL notifies us what it's doing behind the scenes:

```
NOTICE:  CREATE TABLE will create implicit sequence
"people_id_seq" for serial column "people.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create
implicit index "people_pkey" for table "people"
NOTICE:  CREATE TABLE / UNIQUE will create implicit
index "people_email_address_key" for table "people"
```

If and when you drop the People table, these implicitly defined objects are dropped automatically.

To list all of the tables, sequences and views in our database, we can use the \d command:

```
linux=# \d
List of relations
 Schema | Name         | Type   | Owner
--------+--------------+--------+--------
 public | people       | table  | reuven
 public | people_id_seq| sequence| reuven
(2 rows)
```
We can add a letter to \d to get a list of only tables (t), indexes (i), sequences (s), functions (f) or views (v). For example, here is a list of the indexes that we have created:

**Garrick, small font below.**

```
linux=# \d
List of relations
Schema |           Name           | Type  | Owner  | Table
-------+--------------------------+-------+--------+-------
public | people_email_address_key | index | reuven | people
public | people_pkey              | index | reuven | people
```

(2 rows)

We can also use \d to inspect a particular object more closely. For example, we can look at our People table with \d People:

**Garrick, shrink below.**

```
linux=# \d People
Table "public.people"
Column        |      Type                   |  Modifiers
--------------+-----------------------------+----------
id            | integer                     | not null default nextval('people_id_seq'::regclass)
first_name    | text                        | not null
last_name     | text                        | not null
email_address | text                        | not null
added_at      | timestamp without time zone | not null default now()
Indexes:
"people_pkey" PRIMARY KEY, btree (id)
"people_email_address_key" UNIQUE, btree (email_address)
```

There are several things to notice in the above output:

- First, PostgreSQL sees the table as public.people, not just people. This is because every object must exist inside of a schema, or namespace, and the default schema is called public. We can use schemas to partition the namespace within a particular database or to handle partitions. This means we don't need to split data across two more databases just to deal with conflicting permissions and names.

- The table name, as well as all column names, are displayed in lowercase letters. That's because PostgreSQL tries to adhere to the SQL standard as best as possible, and the standard says that identifiers should be case-insensitive. If you really want case-sensitive names (and you probably don't), use double quotes around the identifiers.

- Our id column has been transformed, as expected, into an integer column with a default value taken from a sequence.

### Constraints

There are some problems with our table definition. Although we have effectively stopped people from storing NULL values in our TEXT columns, we haven't done anything to stop them from entering empty strings. In addition, we might want to ensure that the email_address column looks at least something like an e-mail address.
We can do this by adding constraints to our columns—tiny functions that check the value being inserted or updated. If the new value doesn't fit the definition, PostgreSQL refuses to allow its insertion. Here's a new definition of our table, with some constraints defined:

**Garrick, shrink below.**

```
CREATE TABLE People (
    id       SERIAL    NOT NULL,
    first_name TEXT    NOT NULL  CHECK (first_name <> ''),
    last_name TEXT    NOT NULL CHECK (last_name <> ''),
    email_address TEXT NOT NULL CHECK (email_address ~* '.@.+\\.'),
    added_at TIMESTAMP  NOT NULL  DEFAULT NOW(),

    PRIMARY KEY(id),
    UNIQUE(email_address)
);  
```

If we inspect our table definition, it has changed somewhat, to include the constraints:

**Garrick, shrink below.**

```
linux=# \d people  
Table "public.people"  
Column |            Type             |      Modifiers                     
---------------+-----------------------------+-----------------------------------
id            | integer                     | not null default nextval('people_id_seq'::regclass)  
first_name    | text                        | not null                           
last_name     | text                        | not null                           
email_address | text                        | not null                           
added_at      | timestamp without time zone | not null default now()             

Indexes:  
"people_pkey" PRIMARY KEY, btree (id)  
"people_email_address_key" UNIQUE, btree (email_address)  
Check constraints:  
"people_email_address_check'' CHECK (email_address ~* '.@.+\\.::text)  
"people_first_name_check'' CHECK (first_name <> '::text)  
"people_last_name_check'' CHECK (last_name <> '::text)  
```

Let's see what happens if we violate these constraints:

**Garrick, shrink below.**

```
linux=# insert into people (first_name , last_name, email_address)  
    values ('', 'Lerner', 'reuven@lerner.co.il');  
ERROR:  new row for relation "people" violates check constraint  
"people_first_name_check"
```

```
linux=# insert into people (first_name , last_name, email_address)  
    values ('Reuven2', 'Lerner2', 'reuven');  
ERROR:  new row for relation "people" violates check constraint  
"people_email_address_check"
```

Sure enough, our constraints help to ensure that our database is in order.

The most common type of constraint is a foreign key, in which one table points to another. For example:

**Garrick, small font below.**

```
CREATE TABLE Appointments (  
    person_id    INTEGER     NOT NULL  REFERENCES People,
```
starting_time TIMESTAMP NOT NULL,  
duration INTERVAL NOT NULL,  
notes TEXT NULL
);

If we try to create an appointment that refers to a non-existent person, we will be rejected:

_Garrick, shrink below._

```sql
INSERT INTO Appointments (person_id, starting_time, duration, notes)  
VALUES (5000, '2007-Feb-12 13:00', interval '1 hour', 'Lunch');
```

ERROR: insert or update on table "appointments" violates foreign key  
constraint "appointments_person_id_fkey"  
DETAIL: Key (person_id)=(5000) is not present in table "people".

Foreign-key constraints help in the other direction as well. If you try to drop a row to which a foreign  
key points, PostgreSQL will refuse the request, indicating that you must first delete the foreign key. You  
can adjust the rules for these constraints by setting the ON UPDATE or ON DELETE modifiers to the  
foreign key definition.

**Other Features**

This list of features is just the tip of the iceberg. _And that's part of the magic of PostgreSQL—out of the  
box, it's straightforward and easy to use, but you almost always can redefine and extend existing  
functionality with your own code and data._ The built-in operators, along with the flexible ways in which  
they can be combined and further enhanced with your own functions and definitions, make for a  
powerful combination. I don't often use unions or intersections, but I do often use views.

For example, one of my favorite features is the ability to use subselects just about anywhere you would  
have a value. If you have someone's e-mail address, you can use that to INSERT a row into  
Appointments in a single command:

_Garrick, shrink below._

```sql
INSERT INTO Appointments (person_id, starting_time, duration, notes)  
VALUES ((SELECT id  
FROM People  
WHERE email_address = 'reuven@lerner.co.il'),  
'2007-Feb-12 13:00', interval '1 hour', 'Lunch');
```

If the existing data types aren't enough, we can construct our own. PostgreSQL already comes with a  
number of existing data types, including geometric shapes, IP addresses and even ISBNs.

If we want to create more than one table with similar characteristics, we can take advantage of  
PostgreSQL's object-oriented features. Thus, we could have a People table and a Managers table, in  
which the definition of Manager inherits the characteristics of People and adds its own extensions.

You also can create your own server-side functions, in a variety of different languages—from  
PostgreSQL's own Pl/pgsql to specialized versions of Perl, Python, Tcl, Java, Ruby and the R statistical  
language. These functions can return individual values or entire tables, and can be used in triggers. You  
also can use these functions to rewrite the rules for inserting, updating and deleting data from a table or  
even a view.
But, perhaps the most important feature of all is the built-in support for transactions. Transactions are an essential part of database programming, in that they allow us to combine multiple queries into one all-or-nothing action. The classic example of a transaction is the movement of money from one bank account to another; if the power goes out, you want to be sure that the money was moved, or that it wasn't. It would be unacceptable for the money to disappear altogether or for it to appear in both accounts when the lights come back on.

Recent versions of PostgreSQL have enhanced its transactional capabilities. Not only can you commit or roll back a transaction, but you also can define savepoints inside a transaction. If something goes wrong, you can either roll back the entire transaction or merely go to the previous savepoint. Moreover, PostgreSQL now supports two-phase commits, making it possible to synchronize distributed processes that require communication and coordination.

If anything goes wrong, PostgreSQL also provides a PITR (point-in-time recovery) through a write-ahead log (WAL), ensuring that even if the power is cut off at the most critical moment, transactions will be committed or rolled back, and that as many transactions as possible will be committed.

You might have noticed that I haven't mentioned locking at all. That's because, for the most part, PostgreSQL users don't have to worry about locking. The lack of locking is handled using a system known as MVCC (multiversion concurrency control), which has only one drawback, namely the creation of many unused and cast-off database rows. The traditional way to handle this in PostgreSQL is to VACUUM the database regularly, removing old rows and clearing up space. Recent versions now include an auto-vacuum agent, reducing or even eliminating the need to VACUUM on a regular basis.

Finally, recent versions of PostgreSQL include support for tablespaces. This means you can spread tables across different directories and filesystems, rather than keep everything under the directory defined by your installation. This can boost performance or reliability significantly, particularly on large databases.

**Conclusion**

Don't think of PostgreSQL as a powerful open-source database. Rather, think of it as a powerful database that happens to be released under an open-source license. It has a wealth of features that make it scalable for large systems and needs, but it is easily approachable by novices who want to begin their journey into the world of relational databases.

**Resources**

The main Web site for PostgreSQL is [http://www.postgresql.org](http://www.postgresql.org). This site contains links to software, documentation, FAQs and a host of mailing lists.


The PostgreSQL community mailing lists are also invaluable sources of help and information. It's not unusual for one of the core developers to answer a question that someone has posed or admit that there is a bug that needs fixing.