PostgreSQL Tutorial

Welcome to the PostgreSQL Tutorial. The following few chapters are intended to give a simple introduction to PostgreSQL, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the PostgreSQL system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading **Part II** to gain a more formal knowledge of the SQL language, or **Part IV** for information about developing applications for PostgreSQL. Those who set up and manage their own server should also read **Part III**.

Table of contents

1 What Is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on **POSTGRES, Version 4.2**, developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- updatable views
- transactional integrity
- multiversion concurrency control

Also, PostgreSQL can be extended by the user in many ways, for example by adding new

- data types
- functions
- operators
- aggregate functions
- index methods
- procedural languages

And because of the liberal license, PostgreSQL can be used, modified, and distributed by anyone free of charge for any purpose, be it private, commercial, or academic.

2 A Brief History of PostgreSQL

The object-relational database management system now known as PostgreSQL is derived from the POSTGRES package written at the University of California at Berkeley. With over two decades of development behind it, PostgreSQL is now the most advanced open-source database available anywhere.

2.1 The Berkeley POSTGRES Project

The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc. The implementation of POSTGRES began in 1986. The initial concepts for the system were presented in **[ston86]**, and the definition of the initial data model appeared in **[rowe87]**. The design of the rule system at that time was described in **[ston87a]**. The rationale and architecture of the storage manager were detailed in **[ston87b]**.

POSTGRES has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in **[ston90a]**, was released to a few external users in June 1989. In response to a critique of the first rule system (**[ston89]**), the rule system was redesigned (**[ston90b]**), and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rule system. For the most part, subsequent releases until Postgres95 (see below) focused on portability and reliability.

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into **Informix**, which is now owned by **IBM**) picked up the code and commercialized it. In

late 1992, POSTGRES became the primary data manager for the **Sequoia 2000 scientific computing project**.

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been devoted to database research. In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

2.2 Postgres95

In 1994, Andrew Yu and Jolly Chen added an SQL language interpreter to POSTGRES. Under a new name, Postgres95 was subsequently released to the web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 release 1.0.x ran about 30–50% faster on the Wisconsin Benchmark compared to POSTGRES, Version 4.2. Apart from bug fixes, the following were the major enhancements:

- The query language PostQUEL was replaced with SQL (implemented in the server). (Interface library **libpq** was named after PostQUEL.) Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregate functions were re-implemented. Support for the `GROUP BY` query clause was also added.
- A new program (psql) was provided for interactive SQL queries, which used GNU Readline. This largely superseded the old monitor program.
- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, `pgtclsh`, provided new Tcl commands to interface Tcl programs with the Postgres95 server.

- The large-object interface was overhauled. The inversion large objects were the only mechanism for storing large objects. (The inversion file system was removed.)
- The instance-level rule system was removed. Rules were still available as rewrite rules.
- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code
- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched GCC (data alignment of doubles was fixed).

2.3 PostgreSQL

By 1996, it became clear that the name "Postgres95" would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

Many people continue to refer to PostgreSQL as "Postgres" (now rarely in all capital letters) because of tradition or because it is easier to pronounce. This usage is widely accepted as a nickname or alias.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the server code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Details about what has happened in PostgreSQL since then can be found in **Appendix E**.

3 Further Information

Besides the documentation, that is, this book, there are other resources about PostgreSQL:

Wiki

The PostgreSQL **wiki** contains the project's **FAQ** (Frequently Asked Questions) list, **TODO** list, and detailed information about many more topics.

Web Site

The PostgreSQL **web site** carries details on the latest release and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open-source project. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. Read the mailing lists and answer questions. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

1 Chapter 1. Getting Started

1.1 Installation

Before you can use PostgreSQL you need to install it, of course. It is possible that PostgreSQL is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access PostgreSQL.

If you are not sure whether PostgreSQL is already available or whether you can use it for your experimentation then you can install it yourself. Doing so is not hard and it can be a good exercise. PostgreSQL can be installed by any unprivileged user; no superuser (root) access is required.

If you are installing PostgreSQL yourself, then refer to **Chapter 17** for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate environment variables.

If your site administrator has not set things up in the default way, you might have some more work to do. For example, if the database server machine is a remote machine, you will need to set the PGHOST environment variable to the name of the database server machine. The environment variable PGPORT might also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the database, you should consult your site administrator or, if that is you, the documentation to make sure that your environment is properly set up. If you did not understand the preceding paragraph then read the next section.

1.2 Architectural Fundamentals

Before we proceed, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make this chapter somewhat clearer.

In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server program is called postgres.
- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. To achieve this it starts (“forks”) a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original postgres process. Thus, the supervisor server process is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

1.3 Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. In that case you can omit this step and skip ahead to the next section.

To create a new database, in this example named mydb, you use the following command:

```
$ createdb mydb
```

If this produces no response then this step was successful and you can skip over the remainder of this section.

If you see a message similar to:

```
createdb: command not found
```

then PostgreSQL was not installed properly. Either it was not installed at all or your shell's search path was not set to include it. Try calling the command with an absolute path instead:

```
$ /usr/local/pgsql/bin/createdb mydb
```

The path at your site might be different. Contact your site administrator or check the installation instructions to correct the situation.

Another response could be this:

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: No such file or directory
        Is the server running locally and accepting connections on that socket?
```

This means that the server was not started, or it is not listening where createdb expects to contact it. Again, check the installation instructions or consult the administrator.

Another response could be this:

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:  role "joe" does not exist
```

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL user account for you. (PostgreSQL user accounts are distinct from operating system user accounts.) If you are the administrator, see **Chapter 22** for help creating accounts. You will need to become the operating system user under which PostgreSQL was installed (usually postgres) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your operating system user name; in that case you need to use the -U switch or set the PGUSER environment variable to specify your PostgreSQL user name.

If you have a user account but it does not have the privileges required to create a database, you will see the following:

```
createdb: error: database creation failed: ERROR:  permission denied to create database
```

Not every user has authorization to create new databases. If PostgreSQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed PostgreSQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. [1]

You can also create databases with other names. PostgreSQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 bytes in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type:

```
$ createdb
```

If you do not want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database mydb, you can destroy it using the following command:

```
$ dropdb mydb
```

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

More about createdb and dropdb can be found in **createdb** and **dropdb** respectively.

---

[1] As an explanation for why this works: PostgreSQL user names are separate from operating system user accounts. When you connect to a database, you can choose what PostgreSQL user name to connect as; if you don't, it will default to the same name as your current operating system account. As it happens, there will always be a PostgreSQL user account that has the same name as the operating system user that started the server, and it also happens that that user always has permission to create databases. Instead of logging in as that user you can also specify the -U option everywhere to select a PostgreSQL user name to connect as.

1.4 Accessing a Database

Once you have created a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC or JDBC support to create and manipulate a database. These possibilities are not covered in this tutorial.

- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in **Part IV**.

You probably want to start up psql to try the examples in this tutorial. It can be activated for the mydb database by typing the command:

```
$ psql mydb
```

If you do not supply the database name then it will default to your user account name. You already discovered this scheme in the previous section using createdb.

In psql, you will be greeted with the following message:

```
psql (14.3)
Type "help" for help.


mydb=>
```

The last line could also be:

```
mydb=#
```

That would mean you are a database superuser, which is most likely the case if you installed the PostgreSQL instance yourself. Being a superuser means that you are not subject to access controls. For the purposes of this tutorial that is not important.

If you encounter problems starting psql then go back to the previous section. The diagnostics of createdb and psql are similar, and if the former worked the latter should work as well.

The last line printed out by psql is the prompt, and it indicates that psql is listening to you and that you can type SQL queries into a work space maintained by psql. Try out these commands:

```
mydb=> SELECT version();
```

```
                                version
-------------------------------------------------------------------------------------------
 PostgreSQL 14.3 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10) 4.9.2, 64-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

The psql program has a number of internal commands that are not SQL commands. They begin with the backslash character, "\". For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

To get out of psql, type:

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more internal commands, type \? at the psql prompt.) The full capabilities of psql are documented in **psql**. In this tutorial we will not use these features explicitly, but you can use them yourself when it is helpful.

2 Chapter 2. The SQL Language

2.1 Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including **[melt93]** and **[date97]**. You should be aware that some PostgreSQL language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named mydb, as described in the previous chapter, and have been able to start psql.

Examples in this manual can also be found in the PostgreSQL source distribution in the directory src/tutorial/. (Binary distributions of PostgreSQL might not provide those files.) To use those files, first change to that directory and run make:

```
$ cd .../src/tutorial
$ make
```

This creates the scripts and compiles the C files containing user-defined functions and types. Then, to start the tutorial, do the following:

```
$ psql -s mydb

...

mydb=> \i basics.sql
```

The \i command reads in commands from the specified file. psql's -s option puts you in single step mode which pauses before sending each statement to the server. The commands used in this section are in the file basics.sql.

2.2 Concepts

PostgreSQL is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. Relation is essentially a mathematical term for *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single PostgreSQL server instance constitutes a database *cluster*.

2.3 Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

You can enter this into psql with the line breaks. psql will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes ("--") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

varchar(80) specifies a data type that can store arbitrary character strings up to 80 characters in length. int is the normal integer type. real is a type for storing single precision floating-point numbers. date should be self-explanatory. (Yes, the column of type date is also named date. This might be convenient or confusing — you choose.)

PostgreSQL supports the standard SQL types int, smallint, real, double precision, char(***N***), varchar(***N***), date, time, timestamp, and interval, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not key words in the syntax, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

The point type is an example of a PostgreSQL-specific data type.

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

2.4 Populating a Table With Rows

The INSERT statement is used to populate a table with rows:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes ('), as in the example. The date type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The point type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

You could also have used COPY to load large amounts of data from flat-text files. This is usually faster because the COPY command is optimized for this application while allowing less flexibility than INSERT. An example would be:

```
COPY weather FROM '/home/user/weather.txt';
```

where the file name for the source file must be available on the machine running the backend process, not the client, since the backend process reads the file directly. You can read more about the COPY command in **COPY**.

2.5 Querying a Table

To retrieve data from a table, the table is *queried*. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT * FROM weather;
```

Here * is a shorthand for “all columns”. [2] So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

```
    city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 San Francisco |     46 |     50 | 0.25 | 1994-11-27
 San Francisco |     43 |     57 |    0 | 1994-11-29
 Hayward       |     37 |     54 |      | 1994-11-29
(3 rows)
```

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

```
    city      | temp_avg |    date
--------------+----------+------------
```

```
San Francisco |      48 | 1994-11-27
San Francisco |      50 | 1994-11-29
Hayward       |      45 | 1994-11-29
(3 rows)
```

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be "qualified" by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
   WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

```
    city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

You can request that the results of a query be returned in sorted order:

```
SELECT * FROM weather
   ORDER BY city;
    city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do:

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
    FROM weather;
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

Here again, the result row ordering might vary. You can ensure consistent results by using DISTINCT and ORDER BY together: [3]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

---

[2] While SELECT * is useful for off-the-cuff queries, it is widely considered bad style in production code, since adding a column to the table would change the results.

[3] In some database systems, including older versions of PostgreSQL, the implementation of DISTINCT automatically orders the rows and so ORDER BY is unnecessary. But this is not required by the SQL standard, and current PostgreSQL does not guarantee that DISTINCT causes the rows to be ordered.

2.6 Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. Queries that access multiple tables (or multiple instances of the same table) at one time are called *join* queries. They combine rows from one table with rows from a second table, with an expression specifying which rows are to be paired. For example, to return all the weather records together with the location of the associated city, the database needs to compare the city column of each row of the weather table with the name column of all rows in the cities table, and select the pairs of rows where these values match.[4] This would be accomplished by the following query:

```
SELECT * FROM weather JOIN cities ON city = name;
    city     | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the cities table for Hayward, so the join ignores the unmatched rows in the weather table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns from the weather and cities tables are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using *:

```
- SELECT city, temp_lo, temp_hi, prcp, date, location
-     FROM weather JOIN cities ON city = name;
```

Since the columns all had different names, the parser automatically found which table they belong to. If there were duplicate column names in the two tables you'd need to *qualify* the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
    weather.prcp, weather.date, cities.location
  FROM weather JOIN cities ON weather.city = cities.name;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this form:

```
SELECT *
  FROM weather, cities
  WHERE city = name;
```

This syntax pre-dates the JOIN/ON syntax, which was introduced in SQL-92. The tables are simply listed in the FROM clause, and the comparison expression is added to the WHERE clause. The results from this older implicit syntax and the newer explicit JOIN/ON syntax are identical. But for a reader of the query, the explicit syntax makes its meaning easier to understand: The join condition is introduced by its own key word whereas previously the condition was mixed into the WHERE clause together with other conditions.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the weather table and for each row to find the matching cities row(s). If no matching row is found we want some "empty values" to be substituted for the cities table's columns. This kind of query is called an *outer join*. (The joins we have seen so far are *inner joins*.) The command looks like this:

```
SELECT *
  FROM weather LEFT OUTER JOIN cities ON weather.city = cities.name;
```

```
    city      | temp_lo | temp_hi | prcp |    date    |     name      | location
--------------+---------+---------+------+------------+---------------+-----------
 Hayward      |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

This query is called a *left outer join* because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

Exercise: There are also right outer joins and full outer joins. Try to find out what those do.

We can also join a table against itself. This is called a *self join*. As an example, suppose we wish to find all the weather records that are in the temperature range of other weather records. So we need to compare the temp_lo and temp_hi columns of each weather row to the temp_lo and temp_hi columns of all other weather rows. We can do this with the following query:

```
SELECT w1.city, w1.temp_lo AS low, w1.temp_hi AS high,
      w2.city, w2.temp_lo AS low, w2.temp_hi AS high
   FROM weather w1 JOIN weather w2
      ON w1.temp_lo < w2.temp_lo AND w1.temp_hi > w2.temp_hi;
    city      | low | high |     city      | low | high
--------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Here we have relabeled the weather table as w1 and w2 to be able to distinguish the left and right side of the join. You can also use these kinds of aliases in other queries to save some typing, e.g.:

```
SELECT *
    FROM weather w JOIN cities c ON w.city = c.name;
```

You will encounter this style of abbreviating quite frequently.

---

[4] This is only a conceptual model. The join is usually performed in a more efficient manner than actually comparing each possible pair of rows, but this is invisible to the user.

2.7 Aggregate Functions

Like most other relational database products, PostgreSQL supports *aggregate functions*. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the count, sum, avg (average), max (maximum) and min (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with:

```
SELECT max(temp_lo) FROM weather;
 max
-----
  46
(1 row)
```

If we wanted to know what city (or cities) that reading occurred in, we might try:

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     WRONG
```

but this will not work since the aggregate max cannot be used in the WHERE clause. (This restriction exists because the WHERE clause determines which rows will be included in

the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a *subquery*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
     city
---------------
 San Francisco
(1 row)
```

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with `GROUP BY` clauses. For example, we can get the maximum low temperature observed in each city with:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using `HAVING`:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
```

```
   HAVING max(temp_lo) < 40;
 city   | max
---------+-----
 Hayward |  37
(1 row)
```

which gives us the same results for only the cities that have all temp_lo values below 40. Finally, if we only care about cities whose names begin with "S", we might do:

```
SELECT city, max(temp_lo)
   FROM weather
   WHERE city LIKE 'S%'          -- (1)
   GROUP BY city
   HAVING max(temp_lo) < 40;
```

(1) The LIKE operator does pattern matching and is explained in **Section 9.7**.

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions. (Strictly speaking, you are allowed to write a HAVING clause that doesn't use aggregates, but it's seldom useful. The same condition could be used more efficiently at the WHERE stage.)

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

2.8 Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT * FROM weather;


     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9 Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

```
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

One should be wary of statements of the form

```
DELETE FROM tablename;
```

Without a qualification, DELETE will remove *all* rows from the given table, leaving it empty. The system will not request confirmation before doing this!

3 Chapter 3. Advanced Features

3.1 Introduction

In the previous chapter we have covered the basics of using SQL to store and access your data in PostgreSQL. We will now discuss some more advanced features of SQL that simplify management and prevent loss or corruption of your data. Finally, we will look at some PostgreSQL extensions.

This chapter will on occasion refer to examples found in **Chapter 2** to change or improve them, so it will be useful to have read that chapter. Some examples from this chapter can also be found in advanced.sql in the tutorial directory. This file also contains some sample data to load, which is not repeated here. (Refer to **Section 2.1** for how to use the file.)

3.2 Views

Refer back to the queries in **Section 2.6**. Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the query each time you need it. You can create a *view* over the query, which gives a name to the query that you can refer to like an ordinary table:

```
CREATE VIEW myview AS
    SELECT name, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which might change as your application evolves, behind consistent interfaces.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

3.3 Foreign Keys

Recall the weather and cities tables from **Chapter 2**. Consider the following problem: You want to make sure that no one can insert rows in the weather table that do not have a matching entry in the cities table. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the cities table to check if a matching record exists, and then inserting or rejecting the new weather records. This approach has a number of problems and is very inconvenient, so PostgreSQL can do this for you.

The new declaration of the tables would look like this:

```
CREATE TABLE cities (
        name     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(name),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

Now try inserting an invalid record:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR:  insert or update on table "weather" violates foreign key constraint
"weather_city_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

The behavior of foreign keys can be finely tuned to your application. We will not go beyond this simple example in this tutorial, but just refer you to **Chapter 5** for more information. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn about them.

3.4 Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like:

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's

officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a *transaction* gives us this guarantee. A transaction is said to be *atomic*: from the point of view of other transactions, it either happens completely or not at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with `BEGIN` and `COMMIT` commands. So our banking transaction would actually look like:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
COMMIT;
```

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed that Alice's balance went negative), we can issue the command ROLLBACK instead of COMMIT, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a BEGIN command, then each individual statement has an implicit BEGIN and (if successful) COMMIT wrapped around it. A group of statements surrounded by BEGIN and COMMIT is sometimes called a *transaction block*.

Note

Some client libraries issue BEGIN and COMMIT commands automatically, so that you might get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

It's possible to control the statements in a transaction in a more granular fashion through the use of *savepoints*. Savepoints allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with SAVEPOINT, you can if needed roll back to the savepoint with ROLLBACK TO. All the transaction's database changes between defining the savepoint and rolling back to it are discarded, but changes earlier than the savepoint are kept.

After rolling back to a savepoint, it continues to be defined, so you can roll back to it several times. Conversely, if you are sure you won't need to roll back to a particular savepoint again, it can be released, so the system can free some resources. Keep in mind that either releasing or rolling back to a savepoint will automatically release all savepoints that were defined after it.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.

Remembering the bank database, suppose we debit $100.00 from Alice's account, and credit Bob's account, only to find later that we should have credited Wally's account. We could do it using savepoints like this:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

This example is, of course, oversimplified, but there's a lot of control possible in a transaction block through the use of savepoints. Moreover, `ROLLBACK TO` is the only way to regain control of a transaction block that was put in aborted state by the system due to an error, short of rolling it back completely and starting again.

3.5 Window Functions

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. However, window functions do not cause rows to become grouped into a single output row like non-window aggregate calls would. Instead, the rows

retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Here is an example that shows how to compare each employee's salary with the average salary in his or her department:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
  depname  | empno | salary |          avg
-----------+-------+--------+----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

The first three output columns come directly from the table `empsalary`, and there is one output row for each row in the table. The fourth column represents an average taken across all the table rows that have the same `depname` value as the current row. (This actually is the same function as the non-window `avg` aggregate, but the `OVER` clause causes it to be treated as a window function and computed across the window frame.)

A window function call always contains an `OVER` clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a normal function or non-window aggregate. The `OVER` clause determines exactly how the rows of

the query are split up for processing by the window function. The PARTITION BY clause within OVER divides the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

You can also control the order in which rows are processed by window functions using ORDER BY within OVER. (The window ORDER BY does not even have to match the order in which the rows are output.) Here is an example:

```
SELECT depname, empno, salary,
      rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

As shown here, the rank function produces a numerical rank for each distinct ORDER BY value in the current row's partition, using the order defined by the ORDER BY clause. rank needs no explicit parameter, because its behavior is entirely determined by the OVER clause.

The rows considered by a window function are those of the “virtual table” produced by the query's FROM clause as filtered by its WHERE, GROUP BY, and HAVING clauses if any. For example, a row removed because it does not meet the WHERE condition is not seen by any window function. A query can contain multiple window functions that slice up the data in different ways using different OVER clauses, but they all act on the same collection of rows defined by this virtual table.

We already saw that ORDER BY can be omitted if the ordering of rows is not important. It is also possible to omit PARTITION BY, in which case there is a single partition containing all rows.

There is another important concept associated with window functions: for each row, there is a set of rows within its partition called its *window frame*. Some window functions act only on the rows of the window frame, rather than of the whole partition. By default, if ORDER BY is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the ORDER BY clause. When ORDER BY is omitted the default frame consists of all rows in the partition. [5] Here is an example using sum:

```
SELECT salary, sum(salary) OVER () FROM empsalary;
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
```

```
  6000 | 47100
  5200 | 47100
(10 rows)
```

Above, since there is no ORDER BY in the OVER clause, the window frame is the same as the partition, which for lack of PARTITION BY is the whole table; in other words each sum is taken over the whole table and so we get the same result for each output row. But if we add an ORDER BY clause, we get very different results:

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
 salary |  sum
--------+-------
  3500 |  3500
  3900 |  7400
  4200 | 11600
  4500 | 16100
  4800 | 25700
  4800 | 25700
  5000 | 30700
  5200 | 41100
  5200 | 41100
  6000 | 47100
(10 rows)
```

Here the sum is taken from the first (lowest) salary up through the current one, including any duplicates of the current one (notice the results for the duplicated salaries).

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically execute after the processing of those clauses. Also, window functions execute after non-window aggregate

functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

If there is a need to filter or group rows after the window calculations are performed, you can use a sub-select. For example:

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
          rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

The above query only shows the rows from the inner query having `rank` less than 3.

When a query involves multiple window functions, it is possible to write out each one with a separate `OVER` clause, but this is duplicative and error-prone if the same windowing behavior is wanted for several functions. Instead, each windowing behavior can be named in a `WINDOW` clause and then referenced in `OVER`. For example:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

More details about window functions can be found in **Section 4.2.8**, **Section 9.22**, **Section 7.2.5**, and the **SELECT** reference page.

---

[5] There are options to define the window frame in other ways, but this tutorial does not cover them. See **Section 4.2.8** for details.

3.6 Inheritance

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let's create two tables: A table `cities` and a table `capitals`. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you're really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
  name       text,
  population real,
  elevation  int,    -- (in ft)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  elevation  int     -- (in ft)
);

CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
    UNION
  SELECT name, population, elevation FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, for one thing.

A better solution is this:

```
CREATE TABLE cities (
  name       text,
  population real,
  elevation  int     -- (in ft)
);

CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all columns (name, population, and elevation) from its *parent*, cities. The type of the column name is text, a native PostgreSQL type for variable length character strings. The capitals table has an additional column, state, which shows its state abbreviation. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an elevation over 500 feet:

```
SELECT name, elevation
  FROM cities
  WHERE elevation > 500;
```

which returns:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an elevation over 500 feet:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

Here the ONLY before cities indicates that the query should be run over only the cities table, and not tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed — SELECT, UPDATE, and DELETE — support this ONLY notation.

Note

Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness. See **Section 5.10** for more detail.

3.7 Conclusion

PostgreSQL has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These features are discussed in more detail in the remainder of this book.

If you feel you need more introductory material, please visit the PostgreSQL **web site** for links to more resources.