CHAPTER 28

From Databases to Datatypes

Taking time to properly design your project's data model is key to its success. Neglecting to do so can have dire consequences not only on storage requirements, but also on application performance, maintainability, and data integrity. In this chapter, you'll become better acquainted with the many facets of the hierarchy of objects within PostgreSQL. By its conclusion, you will be familiar with the following topics:

- The difference between the various levels of the PostgreSQL hierarchy, including clusters, databases, schemas, and tables.
- The purpose and range of PostgreSQL's supported datatypes. To facilitate reference, these datatypes are broken into four categories: date and time, numeric, textual, and Boolean.
- PostgreSQL's table attributes, which serve to further modify the behavior of tables and their columns.
- How to use advanced concepts, such as constraints and domains, to help further enforce data integrity.

Working with Databases

While most people think of a database as a single entity, the truth is that a single installation of PostgreSQL can handle many unique databases at the same time. This collection of databases is technically referred to as a *cluster*. In this section, we look at how to manipulate databases within a cluster.

Default Databases

By default, a PostgreSQL cluster comes with two template databases, `template0` and `template1`. These databases contain all of the basic information that is needed to create new databases on the system. When you initially connect to a new installation of PostgreSQL, you'll want to connect to the `template1` database and use that to create a new database. If there are schema objects or extensions that you need to load into PostgreSQL that you want all future databases to have access to, you can load them into the `template1` database. The `template0` database is mainly provided as a backup in case you manage to modify your `template1` database in a manner that cannot be corrected.

Creating a Database

There are two common ways to create a database. Perhaps the easiest is to create it using the `CREATE DATABASE` command from within the `psql` client:

```
template1=# CREATE DATABASE company;
CREATE DATABASE
```

You can also create a database via the `createdb` command-line tool:

```
]$ createdb company
CREATE DATABASE
]$
```

Common problems that lead to failed database creation include insufficient or incorrect permissions, or an attempt to create a database that already exists.

Connecting to a Database

Once the database has been created, you can connect to it with the `\c` `psql` command:

```
template1=# \c company
You are now connected to database "company".
company=#
```

Alternatively, you can connect directly into that database when logging in via the `psql` client by passing its name on the command line, like so:

```
]$ psql company
```

In both cases, you'll immediately have the database tables and data at your disposal upon executing each command.

Deleting a Database

You delete a database in much the same fashion as you create one. You can delete it from within the `psql` client with the `DROP DATABASE` command:

```
company=# DROP DATABASE company;
ERROR:  cannot drop the currently open database
```

You should be aware that you cannot drop a database that is currently being accessed. If you are connected to the database, you must first connect to another database before the `DROP` command will work:

```
company=# \c template1
You are now connected to database "template1".
template1=# DROP DATABASE company;
DROP DATABASE
template1=#
```

Alternatively, you can delete it with the `dropdb` command-line tool:

```
]$ dropdb company
DROP DATABASE
]$
```

Modifying Existing Databases

You can also modify certain aspects of a database by using the ALTER DATABASE command. One such example would be that of renaming an existing database:

```
template1=# ALTER DATABASE company RENAME TO testing;
ALTER DATABASE
template1=#
```

As with the DROP DATABASE command, you cannot rename a database that has any active connections. Although you can modify other attributes of a database, the ALTER DATABASE command has contained different options in every release since it was added in 7.3, and there will be additional changes in 8.1 as well, so we will refer you to the documentation for your specific version for a complete list of options.

■ **Tip** You may have noticed that this text often uses all uppercase text for SQL keywords such as ALTER, DATABASE, and RENAME. This is not mandatory; you could accomplish all of the examples in this book using lowercase commands. However, using all uppercase is fairly common practice, and your code will be much more readable if you follow this convention.

Working with Schemas

Schemas contain a collection of tables, views, functions, and other types of objects, within a single database. Unlike with multiple databases, multiple schemas within a database are designed to allow any user to easily access any of the objects within any of the schemas in the database, as long as they have the proper permissions. A few of the reasons you might want to use schemas include:

- To organize database objects into logical groups to make them more manageable
- To allow multiple users to work within one database without interfering with each other
- To put third-party applications into separate schemas so that they do not collide with the names of existing objects in your database

The commands discussed in this section will help you get started using schemas.

Creating Schemas

You can use the CREATE SCHEMA command to create new schemas:

```
CREATE SCHEMA rob;
```

Altering Schemas

You can change the name of a schema by using the `ALTER SCHEMA` command:

```
ALTER SCHEMA rob RENAME TO robert;
```

Dropping Schemas

Dropping a schema is done through the `DROP SCHEMA` command. By default, you cannot drop a schema that contains any objects. You can control this by using the `CASCADE` or `RESTRICT` keywords:

```
DROP SCHEMA robert CASCADE;
```

The Schema Search Path

Once you begin adding schemas into your database, you will quickly begin to realize that working with multiple schemas can be a pain when you have to reference every object with a fully qualified *`schemaname.tablename`* notation. To get around this problem, PostgreSQL supports a schema search path setting akin to the search paths used for executables and libraries in most operating systems. In order for the operating system to find an executable or library, you first have to tell it where to look by giving it a list of directories that could contain the item of interest. Then, you have to place the item into one of these directories. The same applies to the PostgreSQL search path.

When you reference a table with an unqualified name, PostgreSQL searches through the schemas listed in the search path until it finds a matching table. You can view the current search path with the following command:

```
rob=# show search_path;
```

Running this command will show the search path:

```
 search_path
--------------
 $user,public
(1 row)
```

The default search path is equivalent to a schema with the same name as the current user, and then the public schema, which is the default schema created for all databases. You can change the search path by issuing a `set` command, like so:

```
set search_path="$user",public,mynewschema;
```

This would add the schema `mynewschema` into the search path, and allow any tables, views, or other system objects to be referenced unqualified. Consider the following command that lists all `customer` tables in the search path:

```
  company=# \dt *.customer
```

As you can see, the new schema is included in the results:

```
           List of relations
   Schema    |   Name   | Type  | Owner
-------------+----------+-------+-------
 mynewschema | customer | table | rob
 public      | customer | table | rob
(2 rows)
```

This example shows two tables named `customer` located in the `company` database. The first table is in the schema we created called `mynewschema`, and the second table is in the default schema called `public`. Remember that the `public` schema is automatically created for you and that, by default, all tables will be created within that schema unless you designate otherwise.

Working with Tables

This section demonstrates how to create, list, review, delete, and alter tables in PostgreSQL.

Creating a Table

A table is created using the `CREATE TABLE` statement. A vast number of options and clauses specific to this statement are available, but it seems a bit impractical to introduce them all in what is an otherwise informal introduction. Instead, we'll introduce various features of this statement as they become relevant in future sections. The purpose of this section is to demonstrate general usage. As an example, let's create an `employee` table for the `company` database:

```
company=# CREATE TABLE employee (
company(#    empid SERIAL UNIQUE NOT NULL,
company(#    firstname VARCHAR(40) NOT NULL,
company(#    lastname VARCHAR(40) NOT NULL,
company(#    email VARCHAR(80) NOT NULL,
company(#    phone VARCHAR(25) NOT NULL,
company(#    PRIMARY KEY(empid)
company(# );
NOTICE:  CREATE TABLE will create implicit sequence "employee_empid_seq"
for serial column "employee.empid"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey"
for table "employee"
CREATE TABLE
```

You can always go back and alter a table structure after it has been created. Later in the chapter, the section "Altering a Table Structure" demonstrates how this is accomplished via the `ALTER TABLE` statement. You will notice that creating this table produces several notices about things like sequences and indexes. Don't worry about these for now—the meaning of `SERIAL`, `UNIQUE`, `NOT NULL`, and so on will be described later in the chapter.

■ **Tip** You can choose whatever naming convention you prefer when declaring PostgreSQL tables. However, you should choose one format and stick with it (for example, all lowercase and singular). Take it from experience, constantly having to look up the exact format of table names because a set format was never agreed upon can be quite annoying.

As you read earlier in the discussion of schemas, you can also create a table in a schema other than the default schema. To do so, simply prepend the table name with the desired schema name, like so: *schemaname.tablename*.

Copying a Table

Creating a new table based on an existing one is a trivial task. The following query produces a copy of the employee table, naming it employee2:

```
CREATE TABLE employee2 AS SELECT * FROM employee;
```

The new table, employee2, will be added to the database. Be aware that while the new table may look like an exact copy of the employee table, it will not contain any default values, triggers, or constraints that may have existed in the original table (these are covered in more detail later in this chapter as well as in Chapter 34).

Sometimes you might be interested in creating a table based on just a few columns found in an existing table. You can do so by simply specifying the columns within the CREATE SELECT statement:

```
CREATE TABLE employee3 AS SELECT firstname,lastname FROM employee;
```

Creating a Temporary Table

Sometimes it's useful to create tables that have a lifetime that is only as long as the current session. For example, you might need to perform several queries on a subset of a particularly large table. Rather than repeatedly run those queries against the entire table, you can create a temporary table for that subset and then run the queries against the smaller temporary table instead. This is accomplished by using the TEMPORARY keyword (or just TEMP) in conjunction with the CREATE TABLE statement:

```
CREATE TEMPORARY TABLE emp_temp AS SELECT firstname,lastname FROM employee;
```

Temporary tables are created in the same way as any other table would be, except that they're stored in a temporary schema, typically something like pg_temp_1. This handling of the temporary schema is done automatically by the database, and is mostly transparent to the end user.

By default, temporary tables last until the end of the current user session; that is, until you disconnect from the database. Sometimes, however, it can be handy to keep a temporary table around only until the end of the current transaction. (Well go into more detail on transactions in Chapter 36; for now, you can think of them as a grouped set of operations, designated by the BEGIN and COMMIT keywords.) You can do this by using the ON COMMIT DROP syntax:

```
CREATE TEMPORARY TABLE emp_temp2 (
   firstname VARCHAR(25) NOT NULL,
   lastname VARCHAR(25) NOT NULL,
   email VARCHAR(45) NOT NULL
)   ON COMMIT DROP ;
```

Remember that this is only useful when used within `BEGIN` and `COMMIT` commands; otherwise, the table will be silently dropped as soon as it is created.

■ **Note** In PostgreSQL, ownership of the `TEMPORARY` privilege is required to create temporary tables. See Chapter 29 for more details about PostgreSQL's privilege system.

Viewing a Database's Available Tables

You can view a list of the tables made available to a database with the `\dt` command:

```
company=# \dt
```

The result of running this command would look something like this:

```
        List of relations
 Schema |   Name    | Type  | Owner
--------+----------+-------+-------
 public | employee | table | rob
(1 row)
```

Viewing Table Structure

You can view a table structure by using the `\d` command along with the table name:

```
company=# \d employee
```

This produces results similar to the following:

```
            Table "public.employee"
  Column   |         Type          |   Modifiers
-----------+-----------------------+-----------------
 empid     | integer               | not null
 firstname | character varying(25) | not null
 lastname  | character varying(25) | not null
 email     | character varying(45) | not null
 phone     | character varying(10) | not null
Indexes:
    "employee_pkey" PRIMARY KEY, btree (empid)
```

Deleting a Table

Deleting, or dropping, a table is accomplished via the `DROP TABLE` statement. Its syntax follows:

```
DROP TABLE tbl_name [, tbl_name...] [ CASCADE | RESTRICT ]
```

For example, you could delete your `employee` table as follows:

```
DROP TABLE employee;
```

You could also simultaneously drop the `employee2` and `employee3` tables created in previous examples like so:

```
DROP TABLE employee2 employee3;
```

By default, dropping a table removes any constraints, indexes, rules, and triggers that exist for the table specified. However, to drop a table that is referenced by a foreign key (see the "REFERENCES" section later in the chapter for more information) in another table, or by a view, you must specify the `CASCADE` parameter, which removes any dependent views entirely. However, it removes only the foreign-key constraint in the other tables, not the tables entirely.

Altering a Table Structure

You'll find yourself often revising and improving your table structures, particularly in the early stages of development. However, you don't have to go through the hassle of deleting and re-creating the table every time you'd like to make a change. Rather, you can alter the table's structure with the `ALTER` statement. With this statement, you can delete, modify, and add columns as you deem necessary. Like `CREATE TABLE`, the `ALTER TABLE` statement offers a vast number of clauses, keywords, and options. You can look up the gory details in the PostgreSQL manual on your own. This section offers several examples intended to get you started quickly.

Let's begin with adding a column. Suppose you want to track each employee's birth date with the `employee` table:

```
ALTER TABLE employee ADD COLUMN birthday TIMESTAMPTZ;
```

Whoops! You forgot the `NOT NULL` clause. You can modify the new column as follows:

```
ALTER TABLE employee ALTER COLUMN birthday SET NOT NULL;
```

Most people don't know what time they were born, so changing the datatype to a `DATE` would be more appropriate. In previous versions of PostgreSQL, this would have meant going through the trouble of creating a new column, updating it, dropping the old column, and then renaming the new column. Fortunately, as of PostgreSQL 8.0, you can now do it simply, using the `ALTER TYPE` command:

```
ALTER TABLE employee ALTER COLUMN birthday TYPE DATE;
```

Of course, now that it is a date column, maybe it would be better served to change the name of the column to `birthdate`. This is done with the `RENAME` command:

```
ALTER TABLE employee RENAME COLUMN birthday TO birthdate;
```

Finally, after all that, you decide that it really isn't necessary to track the employee's birth date. Go ahead and delete the column:

```
ALTER TABLE employee DROP COLUMN birthdate;
```

Working with Sequences

Sequences are special database objects created for the purpose of assigning unique numbers for input into a table. Sequences are typically used for generating primary key values, especially in cases where you need to do multiple concurrent inserts but need the keys to remain unique. Let's now look at how to work with sequences.

Creating a Sequence

The syntax for creating a sequence is as follows:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

The `TEMPORARY` and `TEMP` keywords indicate that the sequence should be created only for the existing session and then dropped on session exit. By default, a sequence increments one at a time, but you can change this by using the optional `INCREMENT BY` keywords. The `MINVALUE` and `MAXVALUE` keywords work as expected, supplying a minimum and maximum value for the sequence to generate. The default values are `1` and `263` (roughly 9 million trillion) – 1. The `START WITH` keywords allow you to specify an initial number for the sequence to begin with other than `1`. The `CACHE` option allows you to specify a number of sequence values to be pre-allocated and stored in memory for faster access. Finally, the `CYCLE` and `NO CYCLE` options control whether the sequence should wrap around to the starting value once `MAXVALUE` has been reached, or should throw an error, which is the default behavior.

Modifying Sequences

You can modify the majority of values of a sequence by using the `ALTER SEQUENCE` command. The syntax is as follows:

```
ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ RESTART [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

As you can see, the `ALTER SEQUENCE` command follows the same structure as the `CREATE SEQUENCE` command, and its keywords match those of the former command as well. Additionally, starting in PostgreSQL 8.1, you can issue the following command to change which schema a sequence is located in:

```
ALTER SEQUENCE name SET SCHEMA new_schema
```

Sequence Functions

The primary interaction with sequences is handled through several sequence-manipulation functions. The functions allow you to retrieve and manipulate the values within a sequence.

nextval

The `nextval` function is used to generate the next value of a sequence. We'll discuss functions more in Chapter 33, but for now the syntax should be straightforward enough:

```
SELECT nextval('sequence_name');
```

currval

The `currval` function is used to determine the most recently obtained value of a sequence on the given connection. This assumes you have called `nextval` at least once during the session; otherwise, `currval` will fail. The syntax follows that of `nextval`:

```
SELECT currval ('sequence_name');
```

lastval

The `lastval` function, new in PostgreSQL 8.1, operates similarly to `currval`, except that instead of explicitly stating the sequence to be called against, `lastval` automatically returns the value of the last sequence `nextval` was called against:

```
SELECT lastval();
```

This makes it a little easier to manipulate tables, because you can insert into a table and retrieve the generated serial key value without having to know the name of the sequence. Like `currval`, calling `lastval` in a session where `nextval` has not been called will generate an error.

setval

The last of the sequence-manipulation functions, `setval` is used to set a sequence's value to a specified number. The `setval` function actually offers two different syntaxes, the first of which follows:

```
SELECT setval('sequence_name',value);
```

This version of `setval` is fairly straightforward, setting the named sequence's value to `value`. Once `setval` has been executed in this way, subsequent `nextval` calls will begin returning the next value based on the sequence definition. For example, if you call `setval` on a sequence and give it a value of `2112`, calling `nextval` on the sequence will return `2113`, and then increase from there. Optionally, you can pass in a third value to `setval` to control this behavior, using the following syntax:

```
SELECT setval('sequence_name', value, is_called);
```

In this form, the `value` determines if the sequence will treat the number passed in as having been called before. By setting `is_called` as `TRUE`, you achieve the same behavior as the two-parameter form of `setval`; however, by setting `is_called` as `FALSE`, the sequence will start with the number passed into `setval` rather than the next value in the sequence. For example, if passed in with a value of `2112` and `is_called` set to `FALSE`, calling `nextval` will first return `2112` and then increase from there.

Deleting a Sequence

To delete a sequence, simply use the `DROP SEQUENCE` command:

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

The `DROP SEQUENCE` command allows you to enter one or more sequence names to be dropped in a given command. The `CASCADE` and `RESTRICT` keywords function just like with other objects; if `CASCADE` is specified, any dependent objects will be dropped automatically; if `RESTRICT` is specified, PostgreSQL will refuse to drop the sequence.

Datatypes and Attributes

It makes sense that you would want to wield some level of control over the data placed into each column of a PostgreSQL table. For example, you might want to make sure that the value doesn't surpass a maximum limit, fall out of the bounds of a specific format, or even constrain the allowable values to a predefined set. To help in this task, PostgreSQL offers an array of datatypes that can be assigned to each column in a table. Each datatype forces the data to conform to a predetermined set of rules inherent to that datatype, such as size, type (string, integer, or decimal, for instance), and format (ensuring that it conforms to a valid date or time representation, for example).

The behavior of these datatypes can be further tuned through the inclusion of attributes. This section introduces PostgreSQL's supported datatypes, as well as many of the commonly used attributes. Because many datatypes support the same attributes, the definitions are grouped under the heading "Datatype Attributes" rather than presented for each datatype. Any special behavior will be noted as necessary, however.

PostgreSQL also offers the ability to create composite types and domains. A *composite type* is, in simple terms, a list of base types with associated field names. *Domains* are also derived from other types, but are based on a particular base type. However, they usually have some type of constraint that limits their values to a subset of what the underlying base type would allow. We will cover both of these features in this section as well.

Datatypes

Because PostgreSQL enables users to create their own custom types, any discussion of PostgreSQL's datatypes is bound to be incomplete. For purposes of the discussion here, we will cover the most common datatypes, offering information about the name, purpose, format, and range of each. If you would like more information on other datatypes offered by PostgreSQL, such as the `inet` type used for holding IP information, or the `bytea` type used for holding binary data, be sure to reference Chapter 8, "Data Types," of the PostgreSQL online manual. To facilitate later reference of the material here, this section breaks down the datatypes into four categories: date and time, numeric, string, and Boolean.

Date and Time Datatypes

Numerous types are available for representing time- and date-based data. The `TIME`, `TIMESTAMP`, and `INTERVAL` datatypes can be declared with a precision value using the optional `(p)` argument. This argument specifies the number of fractional digits retained in the seconds field.

DATE

The `DATE` datatype is responsible for storing date information. By default, PostgreSQL displays `DATE` values in a standard `YYYY-MM-DD` format, although the values can be inserted using strings in a variety of different formats. For example, both `'20040810'` and `'2004-08-10'` would be accepted as valid input.

The range for the `DATE` datatype is `4713 BC` to `32767 AD`, and the storage requirement is 4 bytes.

■ **Note** For all date and time datatypes, PostgreSQL accepts any type of nonalphanumeric delimiter to separate the various date and time values. For example, `'20040810'`, `'2004*08*10'`, `'2004, 08, 10'`, and `'2004!08!10'` are all the same as far as PostgreSQL is concerned.

TIME [(p)] [without time zone]

The `TIME` datatype is responsible for storing time information. The `TIME` datatype can take input in a number of string formats. The formats `'04:05:06.789'`, `'04:05 PM'`, and `'040506'` are all examples of valid time input. The range for the `TIME` datatype is from `00:00:00.00` to `23:59:59.99`, and the storage requirement is 8 bytes.

The following is an example of using the `(p)` argument in `psql`:

```
company=# SELECT '12:34:56.543'::time(2);
    time
-------------
 12:34:56.54
```

As you can see from the example, we cast the value to a `time(2)`, meaning the time value will be stored only to the last two digits of precision. Normally, you do not have to worry about precision, because by default there is no explicit bound.

TIME [(p)] WITH TIME ZONE

The `TIME` datatype is responsible for storing time information along with time zone information. The `TIME` datatype can take input in a number of string formats. The formats `'04:05:06.789 PST'`, `'04:05 PM'`, and `'040506-08'` are all examples of valid time input. The range for the `TIME` datatype is from `00:00:00.00` to `23:59:59.99`, and the storage requirement is 8 bytes.

■ **Tip** For datatypes `WITH TIME ZONE`, if a time zone is not specified, the default system time zone is used. You can view the system time zone with the `SHOW TIMEZONE` command.

TIMESTAMP [(p)] [without time zone]

The `TIMESTAMP` datatype is responsible for storing a combination of date and time information. Like `DATE`, `TIMESTAMP` values are stored in a standard format, `YYYY-MM-DD HH:MM:SS`; the values can be inserted in a variety of string formats. For example, both `'20040810 153510'` and `'2004-08-10 15:35:10'` would be accepted as valid input. The range for the `TIMESTAMP` datatype is `4713 BC` to `5874897 AD`. The storage requirement is 8 bytes

TIMESTAMP [(p)] WITH TIME ZONE

The `TIMESTAMP WITH TIME ZONE` datatype, often referred to as just `TIMESTAMPTZ`, is responsible for storing a combination of date and time information along with time zone information. Like `DATE`, `TIMSTAMPTZ` values are stored in a standard format, `YYYY-MM-DD HH:MM:SS+TZ`; the values can be inserted in a variety of string formats. For example, both `'20040810 153510'` and `'2004-08-10 15:35:10+02'` would be accepted as valid input. The range for the `TIMESTAMP WITH TIME ZONE` datatype is `4713 BC` to `5874897 AD`. The storage requirement is 8 bytes

INTERVAL [(p)]

The `INTERVAL` datatype is responsible for holding time intervals. The format for `INTERVAL` data can take the form of either explicitly declared intervals or implied intervals. For example, `'4 05:01:02'` and `'4 days 5 hours 1 min 2 sec'` are equivalent, valid input formats. Valid units for the `INTERVAL` type include `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, and `millennium` (and their plurals). The range for the `INTERVAL` type is `-178000000` years to `178000000` years and the storage requirement is 12 bytes.

Here's the generic syntax of `INTERVAL`:

```
quantity unit [quantity unit...]
```

Numeric Datatypes

Numeric datatypes consist of 2-, 4-, and 8-byte integers, 4- and 8-byte floating-point numbers, and selectable-precision decimals.

SMALLINT

The `SMALLINT` datatype offers PostgreSQL's smallest integer range, supporting a range of `-32,768` to `32,767`. It is also referred to as `INT2`. The storage requirement is 2 bytes.

INTEGER

The `INTEGER` datatype is the usual choice for integer type, supporting a range of `-2,147,483,648` to `2,147,483,647`. It is also referred to as `INT` or `INT4`. The storage requirement is 4 bytes.

BIGINT

The `BIGINT` datatype offers PostgreSQL's largest integer range, supporting a range of `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`. It is also referred to as `INT8`. The storage requirement is 8 bytes.

NUMERIC (P,(S))

The NUMERIC datatype can store numbers with up to 1,000 digits of precision, and will perform calculations exactly. It is normally recommended that you store monetary values in this datatype, though you should be aware that arithmetic on NUMERIC values may be slow relative to the other numeric types. The NUMERIC type can be declared with an optional precision and scale. The precision is the total number of digits on both sides of the decimal, whereas the scale is the total number of digits to the right of the decimal. For example, the value 123.45 would be represented as numeric(5,2). If you attempt to insert a number that is too large, PostgreSQL rounds the number to a legal value for INSERT; for example, inserting 123.456 into the above representation would be stored as 123.46. You should also be aware that you can declare NUMERIC without a scale, which implies 0 for the scale, or without either a scale or precision, which implies no limits, although this latter practice is not recommended, for performance reasons.

The storage requirement for NUMERIC varies depending on the number being stored. The basic formula is 4 bytes for a variable-length header, 4 bytes for a numeric header, and 2 bytes for every 4 decimal digits, but even 0 requires 8 bytes.

■ **Note** The DECIMAL(P,(S)) type is equivalent to the NUMERIC type.

REAL

The REAL datatype is an inexact, variable-precision, floating-point number. On most platforms it has a range of at least 1E-37 to 1E+37 and supports a precision of at least six decimal places. Using the NUMERIC datatype rather than REAL is usually recommended, because REAL stores numbers in an inexact, though standards-compliant, way. The storage requirement is 4 bytes.

DOUBLE PRECISION

The DOUBLE PRECISION datatype is a variable-precision, floating-point number, supporting a range of around 11E-307 to 1E+308 and a precision of at least 15 digits. The storage requirement is 8 bytes.

FLOAT [(p)]

The FLOAT datatype is a SQL-standard notation for specifying inexact numeric types. FLOAT takes an optional argument, (p), which signifies the minimum acceptable precision in binary digits. PostgreSQL interprets FLOAT(1) to FLOAT(24) as the REAL datatype, and FLOAT(25) to FLOAT(53) as the DOUBLE PRECISION datatype.

■ **Note** REAL, DOUBLE PRECISION, and FLOAT also accept three special values in addition to ordinary numeric values. Those values represent the IEEE 754 special values of 'Infinity', '-Infinity' (negative infinity), and 'NaN' (not a number). On input, these strings are recognized in a case-insensitive manner.

SERIAL

The SERIAL type is not a true type, but is actually a notational convenience for setting up auto-incrementing identifier columns. In PostgreSQL 8.0, specifying

```
CREATE TABLE tblname (
    colname SERIAL
);
```

is the equivalent of:

```
CREATE TABLE tblname (
    colname INTEGER DEFAULT nextval('tblname_colname_seq') NOT NULL
);
```

Both cases create an INTEGER column and arrange for its value to be assigned from a sequence generator, with the difference being that the SERIAL syntax also attempts to create the sequence automatically upon table creation, rather than having to create the sequence manually. Conversely, sequences created via the SERIAL syntax will also be dropped automatically if the column or table is dropped.

Normally, when creating a SERIAL column, you also want to specify a UNIQUE or PRIMARY KEY constraint to prevent duplicate values from being inserted by accident, but this is not automatic. Primary keys are explained a bit later, in the "PRIMARY KEY" section.

Since the SERIAL type is implemented using the INTEGER type, it can only hold up to 2,147,483,647 values. While this is usually enough for most applications, if you expect that you will need more identifiers, you can use the type BIGSERIAL. BIGSERIAL behaves in all the same respects as SERIAL, except that it uses the BIGINT type for its underpinnings, and thus can support a range up to 9,223,372,036,854,775,807 values.

String Datatypes

PostgreSQL's string types are greatly simplified compared to many other database systems, but they are still the basis for storing string data. This section introduces the string types.

CHAR[(length)]

The CHAR datatype offers PostgreSQL's fixed-length string representation. If a string longer than length is inserted, it will produce an error, unless the characters are all spaces, in which case the string will be truncated to length. (This exception, while odd to some, is required by the SQL standard.) If an inserted string does not occupy length spaces, the remaining space will be padded by blank spaces. A CHAR declaration without a length is equivalent to CHAR(1). CHAR is equivalent to the SQL standard CHARACTER(n), and both names can be used interchangeably.

■ **Note** There is also a datatype "char" (note the quotes and lowercase) that is different from CHAR or CHAR(1) in that it uses only 1 byte for storage. It is used internally within the system tables and is not intended for general use. It is mentioned here because some applications and developers may accidentally quote the CHAR attribute, which can lead to unexpected behavior.

VARCHAR[(length)]

The VARCHAR datatype offers PostgreSQL variable-length string representation. If a string longer than length is inserted, it will produce an error, unless the characters are all spaces, in which case the string will be truncated to length. (Again, this exception is required by the SQL standard.) If an inserted string does not occupy length spaces, the remaining space will not be padded; the shorter string will simply be stored as is. VARCHAR without length will accept a string of any size (this is a PostgreSQL extension). VARCHAR is equivalent to the SQL standard CHARACTER VARYING(n), and both names can be used interchangeably.

TEXT

The TEXT datatype also offers a variable-length string representation. The TEXT type accepts strings of any length. Although type TEXT is not in the SQL standard, it is common among database management systems, and is the recommended datatype for strings that do not require an absolute length requirement.

■ **Tip** Unlike most database systems, PostgreSQL does not have performance differences between CHAR(n), VARCHAR(n), and TEXT. Likewise, the storage requirements between these types are the same (save for the space needed for padding for CHAR types). For this reason, in most cases it is simpler to use TEXT than to make up an arbitrary limit for one of the other types.

Boolean Datatype

The BOOLEAN datatype is PostgreSQL's logical Boolean representation, and it complies with the SQL standard notion of Boolean. PostgreSQL's Boolean can be one of three states: TRUE, FALSE, or NULL (where NULL implies the notion of being unknown). BOOLEAN accepts a number of different representations for TRUE and FALSE including TRUE, 't', 'true', 'y', 'yes', '1', and FALSE, 'f', 'false', 'n', 'no', '0', respectively. The keywords TRUE and FALSE are SQL-compliant, and thus are generally preferred. While PostgreSQL's C library, and applications that build on top of that library, tends to display Booleans as 't' and 'f', they are not equivalent to string data and are not stored as such; BOOLEAN datatypes require only 1 byte of storage.

Datatype Attributes

Although this list is not exhaustive, this section covers those attributes you'll most commonly use, as well as those that will be used throughout the remainder of this book.

CHECK

The CHECK attribute provides a means for restricting the values in a column and, as such, is commonly referred to as a check constraint. The constraint must equate to a Boolean expression, and the value in question must resolve the expression to either TRUE or NULL. A common example of a check constraint is creating a column that should not accept negative values. You could define such a column as follows:

```
vacation_days_earned INTEGER CHECK (vacation_days > 0)
```

You can also reference other columns in a check constraint:

```
vacation_days_taken INTEGER CHECK (vacation_days_taken < vacation_days_earned)
```

PostgreSQL also allows you to define constraints at the table level. One advantage of using a table constraint is that you can give each constraint a unique name, as follows, which vastly simplifies deducing error messages that are given when a constraint is violated:

```
CREATE TABLE employee (
    employee_id SERIAL UNIQUE NOT NULL,
    vacation_days_earned INTEGER CHECK (vacation_days_earned  > 0),
    vacation_days_taken INTEGER CHECK (vacation_days_taken > 0),
    CONSTRAINT no_vacation_days_left CHECK
        (vacation_days_taken <=     vacation_days_earned)
);
```

The advantage of this becomes clear with an example error message; here we will try to insert an employee who has earned 5 days of vacation but has taken 10 days:

```
company=# INSERT INTO employee VALUES (DEFAULT,5,10);
ERROR:  new row for relation "employee" violates check constraint
"no_vacation_days_left"
```

DEFAULT

The DEFAULT attribute ensures that some designated value will be assigned when no other value is available. The value can be a literal value or a simple expression, but cannot include a subquery or reference other columns within its definition, and the value must result in a valid type for the given column. One common example of a DEFAULT is the value now() for TIMESTAMP columns:

```
initial_registration TIMESTAMPTZ DEFAULT now()
```

Using now() causes the system to insert the current system time into the field upon insert. If DEFAULT is not specified on a column, a default value of NULL will be inserted into the column.

In the following example, we create a small test table to hold just an example identifier and a column to hold our timestamp value. We then insert three different entries; the first passes in the DEFAULT keyword, the second specifies a specific time, and the third leaves out the TIMESTAMP column altogether.

```
rob=# CREATE TABLE default_now_example (
rob-#    attempt text,
rob-#    insert_time timestamptz DEFAULT now()
rob -# );
CREATE TABLE
rob=# INSERT INTO default_now_example VALUES ('a', DEFAULT);
INSERT 0 1
rob=# INSERT INTO default_now_example (attempt, insert_time)
rob-# VALUES ('b','1492-01-13 21:12');
INSERT 0 1
rob=# INSERT INTO default_now_example (attempt) VALUES ('c');
INSERT 0 1
```

You can view the results of these entries easily enough:

```
rob=# SELECT * FROM default_now_example;
 attempt |          insert_time
---------+-------------------------------
 a       | 2005-10-16 15:41:39.382608-05
 b       | 1492-01-13 21:12:00-05
 c       | 2005-10-16 15:42:17.860467-05
rows)
```

As you can see, in our first INSERT statement, the default time was entered because we passed in the DEFAULT keyword. In the second, the time we specified was entered. In the third, an autogenerated time was inserted because we did not specify a value; this is the same behavior as using the DEFAULT keyword.

NOT NULL

Defining a column as NOT NULL disallows any attempt to insert a NULL value into the column. Using the NOT NULL attribute, where relevant, is always suggested, because it results in at least baseline verification that all necessary values have been passed to the query. An example of a NOT NULL column assignment follows:

```
zipcode VARCHAR(10) NOT NULL
```

NULL

Simply stated, the NULL attribute means that NULL values are acceptable for the given field. This is also the default value for the field if no data is given and there is no DEFAULT attribute specified. This is the default characteristic for columns in PostgreSQL, so you will not often see it stated explicitly.

PRIMARY KEY

The PRIMARY KEY attribute is used to guarantee uniqueness for a given row. No values residing in a column designated as PRIMARY KEY are repeatable or nullable within that column. It's quite common to see SERIAL columns designated as a primary key, because this column doesn't necessarily have to bear any relation to the row data, other than acting as its unique identifier. However, there are two other ways for ensuring a record's uniqueness:

- **Single-field primary keys**: Typically used when a pre-existing, nonmodifiable unique identifier exists for each row entered into the database, such as a part number or social security number. Note that this key should never change once it is set.
- **Multiple-field primary keys**: Can be useful when it is not possible to guarantee uniqueness from any single field within a record. Thus, multiple fields are conjoined to ensure uniqueness. If the number of columns required to ensure uniqueness grows cumbersome, it is common practice to simply designate a SERIAL integer as the primary key, to alleviate the need to somehow generate unique identifiers with every insertion.

The following three examples demonstrate creation of the auto-increment, single-field, and multiple-field primary key fields, respectively.

Creating an automatically incrementing primary key:

```
CREATE TABLE staff (
   staffid SERIAL NOT NULL PRIMARY KEY,
   fname TEXT NOT NULL,
   lname TEXT NOT NULL,
   email TEXT NOT NULL
);
```

Creating a single-field primary key:

```
CREATE TABLE citizen (
   ssid VARCHAR(9) NOT NULL PRIMARY KEY,
   fname TEXT NOT NULL,
   lname TEXT NOT NULL,
   zipcode VARCHAR(10) NOT NULL
);
```

Creating a multiple-field primary key:

```
CREATE TABLE friend (
   fname TEXT NOT NULL,
   lname TEXT NOT NULL,
   nickname TEXT NOT NULL,
   PRIMARY KEY(lname, nickname)
);
```

REFERENCES

The `REFERENCES` attribute specifies that the values in a column (or group of columns) must match the values appearing in some row of another table. This is done to ensure referential integrity between the two tables. As an example, we could rewrite the `staff` table in our previous example to the following:

```
CREATE TABLE staff (
    staffid SERIAL NOT NULL PRIMARY KEY,
    ssid VARCHAR(9) REFERENCES citizen (ssid),
    email TEXT NOT NULL
);
```

Created this way, it is now impossible to add an entry to the `staff` table that does not have a corresponding entry in the `citizen` table. While some would say this approach to staffing might be short-sighted in today's global economy, opponents of illegal immigration would surely applaud this design.

This relationship between the two tables is often referred to as a *foreign key* (no pun intended), and it provides other benefits as well. You'll notice that we eliminated the `fname` and `lname` columns from our table; we did this because we can now infer this information from the relationship between the two tables. This also means that, should someone's name change (for example, when someone gets married), we do not have to write extra application code to propagate the changes throughout our database: the change can be made in one place and all related tables can be left alone. We can also create foreign keys between tables based on a group of columns between the two tables. We will re-create our `staff` table again to show the syntax:

```
CREATE TABLE staff (
    staffid SERIAL NOT NULL PRIMARY KEY,
    email TEXT NOT NULL,
    lname TEXT,
    nickname TEXT,
    FOREIGN KEY (lname,nickname) REFERENCES friends(lname,nickname)
);
```

This syntax sets up the relationship just like our previous example; any entry in `staff` must now have a corresponding entry, based on both the `lname` and `fname` columns, in the `friends` table.

UNIQUE

A column assigned the `UNIQUE` attribute ensures that all values possess distinct values, except that `NULL` values are repeatable. You typically designate a column as `UNIQUE` to ensure that all fields within that column are distinct—for example, to prevent the same e-mail address from being inserted into a newsletter subscriber table multiple times, while at the same time acknowledging that the field could potentially be empty (`NULL`). An example of a column designated as `UNIQUE` follows:

```
email TEXT UNIQUE
```

Composite Datatypes

A composite datatype defines the structure of a row or record. In simple terms, it is a list of field names and their datatypes. Once a composite type is created, it can be used much like any other datatype, such as when defining a column in a table or declaring a return type for a function. This can prove very useful when you want to tightly couple related information together into a single logical piece.

Creating Composite Types

You can use the `CREATE TYPE` command to create composite types. As shown next, the syntax is similar to that of the `CREATE TABLE` command, though only field names and datatypes can be specified. No constraints or default values can be included.

```
CREATE TYPE im_accounts AS (
jabber    text,
aim    text,
irc    text
);
```

Let's run through a quick example so that you can see exactly how this works. First, we create a table in which to use our new composite type:

```
company=# CREATE TABLE contacts (employee_id integer, im im_accounts);
CREATE TABLE
```

Next, we insert some data into our table. Note that the syntax for inserting into a composite type simply involves encapsulating the various pieces of information that make up the field within parentheses:

```
company=# INSERT INTO contacts (employee_id, im)
company-# VALUES (1,('bigceo@jabber.org','thebigceo','bigceo76'));
INSERT 0 1
```

And finally, for good measure, let's take a look at our data:

```
company=# SELECT * FROM contacts;
 employee_id |                   im
-------------+--------------------------------------
           1 | (bigceo@jabber.org,thebigceo,bigceo76)
(1 rows)
```

Altering Composite Types

The `ALTER TYPE` command can be used to change the definition of an existing composite type. In versions prior to PostgreSQL 8.1, this is limited to changing the owner of the type:

```
ALTER TYPE im_accounts OWNER TO amber;
```

Starting in 8.1, PostgreSQL also gives you the ability to alter the schema of a given type:

```
ALTER TYPE im_accounts SET SCHEMA mynewschema;
```

Dropping Composite Types

Dropping a composite type is done through the `DROP TYPE` command. By default, you cannot drop a composite type that is referenced by any other objects. This can be controlled by using the `CASCADE` or `RESTRICT` keywords, and can be schema-qualified if needed:

```
DROP TYPE mynewschema.im_accounts CASCADE;
```

■ **Note** The `DROP CASCADE` command may have different effects depending on the dependent object. For example, if a table references the composite type, only the column in question will be dropped. However, if a view references the composite type, the entire view will be dropped.

Working with Domains

Domains can be considered a cross between a datatype and a constraint. Creating a domain generally requires two pieces of information: the underlying base type that the domain will use, and the constraint limiting the acceptable values for the domain. While you might think this sounds complicated, it isn't especially, and domains can be quite useful when applied properly. One good example is handling phone numbers. Many databases have a phone number column in several of their tables, which then requires each table to set up its own constraints to handle the data. Rather than go through that hassle, you could instead create a domain to handle phone numbers and then use that in all of your tables.

Creating Domains

Domains are created by using the `CREATE DOMAIN` command. Domains generally comprise a set of attributes, `CHECK`, `DEFAULT`, `NOT NULL`, or `NULL`, that behave like other datatype attributes within PostgreSQL. In this example, we set up a domain to match a valid U.S. phone number, which we define as starting with 1, followed by a dash, three numbers, another dash, three more numbers, a third dash, and then four numbers:

```
CREATE DOMAIN us_phone_number AS TEXT CONSTRAINT "valid_phone_number"
CHECK (VALUE ~ '^1-\\d{3}-\\d{3}-\\d{4}$');

CREATE TABLE us_contact_info (
fullname    TEXT NOT NULL,
email    TEXT NOT NULL,
phone us_phone_number NOT NULL
);
```

As you can see, writing the regular expression once in the domain is much simpler than writing this expression several times in multiple tables. This also gives us one place to change should we need to modify our phone number definition.

Altering Domains

You can use the `ALTER DOMAIN` command to modify any aspect of a domain's definition. Each form of the `ALTER DOMAIN` command takes the form of `ALTER DOMAIN` *`domain_name`* followed by one of the following subforms:

- `{ SET DEFAULT` *`expression`* `| DROP DEFAULT }`: Sets *`expression`* as the default value or drops the existing default value.
- `{ SET | DROP } NOT NULL`: Controls whether or not the domain allows `NULL` values.
- `ADD` *`domain_constraint`*: Adds a new constraint to the domain using the same syntax as the `CREATE DOMAIN` command. It will succeed only if all values in an existing column satisfy the new constraint.
- `DROP CONSTRAINT` *`constraint_name`* `[ RESTRICT | CASCADE ]`: Drops constraints on a domain.
- `OWNER TO` *`new_owner`*: Changes the ownership of the domain.

Using these commands should be fairly straightforward, but just to make sure, let's walk through a few examples. This command would forbid someone from entering `NULL` values into our `DOMAIN`:

```
ALTER DOMAIN us_phone_number SET NOT NULL;
```

This combination would change the owner of the domain to a database user named `amber`:

```
ALTER DOMAIN us_phone_number OWNER TO amber;
```

Dropping Domains

You can remove a domain from the database by using the `DROP DOMAIN` command. By default, you cannot drop a domain that is referenced inside another database object. However, you can control this behavior by using the `CASCADE` or `RESTRICT` keyword along with the `DROP` command:

```
DROP DOMAIN us_phone_number CASCADE;
```

■**Note** The `DROP CASCADE` command may have different effects depending on the dependent object. For example, if a table references the domain, only the column in question will be dropped. However, if a view references the domain, the entire view will be dropped.

Summary

In this chapter, you learned about the many ingredients that go into designing a PostgreSQL database. The chapter began with an overview of some helpful commands for dealing with databases, schemas, and tables. This discussion was followed by an introduction to PostgreSQL's supported datatypes, offering information about the name, purpose, and range of each. The chapter then examined many of the most commonly used attributes, which serve to further tweak column behavior. The chapter concluded with a discussion of how to make use of more advanced datatype objects, including composite datatypes and domains, to help simplify datatype management.

In the next chapter, we'll dive into another key PostgreSQL feature: security. You'll learn all about PostgreSQL's powerful authentication system, as well as learn more about how to secure the PostgreSQL server and create secure PostgreSQL connections using SSL.