CHAPTER 32

Views and Rules

In Chapter 28 we looked at the basic objects within PostgreSQL that you can use to help design your project's data model. While schemas, tables, and other items such as domains are very helpful, they by no means comprise a complete list of the tools at your disposal. In this chapter, we look at PostgreSQL's support of a more formal object in relational theory, the view, and also introduce you to PostgreSQL's powerful rule system. By the end of the chapter, we will have covered the following:

- How to create and manipulate views within PostgreSQL
- PostgreSQL's rule system, including what types of commands can be used from within the rule system
- Updateable views and how you can use PostgreSQL's rule system to implement powerful versions of this classic relational concept

Working with Views

When working on a large data model, you frequently have to use complex queries to retrieve information from several joined tables, often with a long list of `WHERE` conditionals. Duplicating these complex queries in different parts of your application code often can be troublesome, especially if your database has multiple interfaces to it. One minor variation between these interfaces can lead to trouble when the end results don't match up.

What would be handy here is a way to name the complex query so that it could be stored in the database, and accessed in a uniform manner by outside applications. This is where views come in. A *view* is defined in PostgreSQL as a stored representation of a given query. Once defined, in many respects a view can be thought of as a virtual table. While a view holds no data itself, it can be queried just like any other table, and you can even create views based on other views.

Creating a View

You create a view by using the `CREATE VIEW` statement. When using the `CREATE VIEW` statement, you specify both a name for the view and an SQL query that defines the structure of the view:

```
CREATE VIEW database_books AS SELECT * FROM books WHERE subject = 'PostgreSQL';
```

This would create a view called database_books that would have an equivalent structure to the books table, as far as column names and their types are concerned. In older versions of PostgreSQL (7.2 and older), if you needed to change the definition of a view, you had to first drop the view and then re-create it. However, in current versions, we can take advantage of the CREATE OR REPLACE VIEW command:

```
CREATE OR REPLACE VIEW database_books AS SELECT * FROM books
WHERE subject = 'PostgreSQL' OR subject = 'Sqlite';
```

■ **Note** The CREATE OR REPLACE VIEW command will work only if you do not change the layout of the column names or their types. If your query produces a different column layout, you will need to drop and then re-create the view.

You can create views by using very complex SQL statements, and you do not need to limit the view to columns from existing tables; derived values and constants are also acceptable, as shown in this example:

```
CREATE VIEW featured_technical_books AS SELECT 'Best Sellers',title,
(copies_sold/months_in_print) AS average_sales FROM books WHERE
current_stock >= 1000 AND genre = 'technical';
```

Dropping a View

Dropping a view is accomplished by using the DROP VIEW command. This command takes an optional keyword of RESTRICT or CASCADE, to determine what behavior to use when dealing with dependent objects, such as different views or functions that query on the view being dropped. The RESTRICT keyword prevents the view from being dropped if it has dependent objects. The CASCADE keyword, used in the following example, drops the dependent objects:

```
DROP VIEW database_books CASCADE;
```

The PostgreSQL Rule System

Many databases use active rules within the database, based on some combination of functions and triggers, that they use to enforce things like data constraints and foreign keys. PostgreSQL also offers those features, but it also offers an alternative system for rewriting queries on the fly. Using the rule system, you can do such things as enforce data integrity, create read-only tables, and make views interactive. These "query-rewrite" rules can be broken down into one of four types; SELECT, INSERT, UPDATE, and DELETE. In this section, we look at the syntax for creating rules and introduce the four different types of rules.

Working with Rules

This section presents the basic syntax used to create and drop rules.

Creating a Rule

You can create a rule by using the CREATE RULE command, the complete syntax for which follows:

```
CREATE [ OR REPLACE ] RULE rule_name AS ON event_type
TO object_name [ WHERE conditional ] DO [ ALSO | INSTEAD ] COMMAND
```

The following list describes the various parts of this syntax:

- CREATE [OR REPLACE] RULE rule_name: Specifies the name of the rule, and takes an optional OR REPLACE clause, which tells PostgreSQL to replace an existing rule with the same name on the same table or view the rule is being created on.
- AS ON event_type: Determines what type of event the rule will be carried out on. The event_type can be one of SELECT, INSERT, UPDATE, or DELETE, each of which is described in more detail in the upcoming "Rule Types" section.
- TO object_name [WHERE conditional]: Specifies the name of the table or view that the rule applies to. An optional conditional can be specified if you want the rule to be carried out only in certain cases. Any SQL conditional expression can be used provided that it returns boolean, does not contain aggregate functions, and references no other tables or views except, optionally, the NEW and OLD pseudo-relations, if appropriate.
- DO [ALSO | INSTEAD]: Describes whether the rule should be applied in addition to the action in the original SQL statement against the table, or if the rule should be applied in place of the original SQL statement. If neither ALSO nor INSTEAD is specified, ALSO is used as the default.
- COMMAND: Specifies the desired query to be run for the rule. Commands take the form of SELECT, INSERT, UPDATE, DELETE, or NOTIFY statements. SQL queries used in the COMMAND section of a rule can access the NEW and OLD pseudo-relations, as appropriate. The COMMAND section can also use the NOTHING keyword, if you do not want any action to be executed for the rule.

■ **Tip** The action specified in the COMMAND section does not have to match that specified in the event_type. For example, you can create a rule that will insert into another table every time someone attempts to update a view.

Removing a Rule

To remove a rule, you use the DROP RULE command. Compared to the CREATE RULE command, the syntax is much simpler:

```
DROP RULE rule_name ON object_name [ CASCADE | RESTRICT]
```

The key elements to the DROP RULE command are the name of the rule, the name of the view or table that the rule applies to, and, optionally, either the CASCADE or RESTRICT keyword. If CASCADE is chosen, then all objects dependent on the rule will be dropped; if RESTRICT is specified, then PostgreSQL will refuse to drop the rule if it has any dependent objects; the default is RESTRICT.

Rule Types

As previously indicated, PostgreSQL has four basic rule types: select, insert, update, and delete. Each of these rules has some unique characteristics, though they all follow the same basic patterns.

Select Rules

The most basic type of rule is the select rule. A select rule is defined as `ON SELECT`, and its action must be an unconditional `SELECT` action that is marked to run instead of the original query. In this way, rules on tables mimic the functionality of views, so much so that the following two examples are functionally equivalent:

```
CREATE VIEW ourbooks AS SELECT * FROM books;
```

and

```
CREATE TABLE ourbooks AS SELECT * FROM books;
CREATE RULE "_RETURN" AS ON SELECT TO ourbooks DO INSTEAD SELECT * FROM books;
```

The use of the name `"_RETURN"` is required by PostgreSQL for `ON SELECT` rules to help signal to the internal query rewriter that the relation being queried is a view. Views within PostgreSQL use select rules automatically to handle select calls and retrieve data from their base tables, but in most cases, you will not need to work with select rules directly.

Insert Rules

The next type of rule used within PostgreSQL is the insert rule. Insert rules can have an action that is either `ALSO` or `INSTEAD`, and can have multiple actions or no action, as desired. The actions defined in an insert rule can contain conditionals and can also make use of the `NEW` pseudo-relation. An insert rule's syntax looks like this:

```
CREATE RULE database_book_insert AS ON INSERT TO database_books DO INSTEAD INSERT
INTO books (title, copies_on_hand, genre) VALUES (NEW.title, 1, 'technical');
```

With this rule in place, any insert directed at the `database_books` view would instead insert the specified values into the original `books` table.

Update Rules

The next type of rule is the update rule. Like insert rules, update rules can have an action that is either `ALSO` or `INSTEAD`, and can have multiple actions or no actions, as desired. The actions defined in an update rule can contain conditionals and can make use of both the `NEW` and `OLD` pseudo-relations. An example update rule might look like this:

```
CREATE RULE database_books_update AS ON UPDATE TO database_books WHERE
NEW.title <> OLD.title DO INSTEAD UPDATE books SET title = NEW.title;
```

With this rule, updates directed at our `database_books` view, where the title had been changed, would instead update the original `books` table with the new title.

Delete Rules

The last type of rule is the delete rule. It can have an action of either ALSO or INSTEAD, and can have multiple actions or no actions, as desired. The actions defined in a delete rule can contain conditionals and can make use of the OLD pseudo-relation. An example delete rule might look like this:

```
CREATE RULE database_books_delete AS ON DELETE TO database_books
DO INSTEAD NOTHING;
```

Here, we use the NOTHING keyword to prevent any deletions from taking place if someone attempts to delete from the database_books view. This has two effects: We prevent those who have access on the database_books view from deleting from our main books table, and we allow DELETE statements against the database_books table to be executed without error. Do not dismiss this second effect too quickly. In most normal cases, deleting from a view (and inserting and updating as well) would raise an error, but with the use of rules, we can handle these errors if our application calls for it.

Making Views Interactive

Many databases these days offer some form of updateable views, allowing you to run UPDATE statements against a view and have them update the underlying table. However, you'll often find that there are heavy restrictions placed on the allowed definitions of this type of view, such as not allowing joined queries or not allowing special formatting of entries in the view definition. PostgreSQL, by way of its rule system, allows you to bypass these restrictions, giving you much more flexibility in the types of updatable views you can create. The next section walks you through several examples of putting PostgreSQL's rule system to use.

Updatable, Insertable, Deletable Views

The first part of our example creates a simple set of tables that we will be working with:

```
CREATE TABLE employee (
  employee_id INTEGER PRIMARY KEY,
  fname TEXT NOT NULL,
  lname TEXT NOT NULL
);
CREATE TABLE phone (
  employee_id INTEGER REFERENCES employee (employee_id) ON DELETE CASCADE,
  npa INTEGER NOT NULL,
  nxx INTEGER NOT NULL,
  xxxx INTEGER NOT NULL);
```

This sets up two tables, one for holding our employee names and one for holding their phone information. Of course, while using column names like npa, nxx, and xxxx may follow the official format of the North American Numbering Plan, they are a little unwieldy to work with, so let's go ahead and make our view:

```
CREATE VIEW directory AS (SELECT employee.employee_id,
fname || ' ' || lname AS name,
npa || '-' || nxx || '-' || xxxx AS number
FROM employee JOIN phone USING (employee_id));
```

This creates a three-column view: one for the employee ID, one for the employee's full name, and one for the employee's phone number, formatted a little bit nicer. Now that we have the structure, let's go ahead and add some data:

```
INSERT INTO employee(employee_id,fname,lname) VALUES
(1,'Amber','Lee');
INSERT INTO phone(employee_id, npa, nxx, xxxx) VALUES
(1,607,555,5210);
```

And take a look at it through our view:

```
rob=# SELECT * FROM directory;
employee_id |   name       |    number
------------+-----------+--------------
                1 | Amber Lee | 607-555-5210
(1 row)
```

This looks nice enough, but suppose we want to add someone new into our directory. To do this, we have to be aware of the underlying tables and insert into both tables:

```
INSERT INTO employee(employee_id, fname, lname)
  VALUES (2,'Dylan','Jairus');
INSERT INTO phone(employee_id, npa, nxx, xxxx)
  VALUES (2,813,555,5040);
```

As you can see, to add information into the system, we have to add data into two tables. It would be nice if we had a way to enter data into the directory directly, in the format that we are comfortable with. This is where our first rule comes into place. We will create a rule that handles direct inserts on the `directory` view, splitting the data into the proper tables and converting it into the proper types needed by the base tables:

```
CREATE RULE directory_addition AS ON INSERT TO directory DO INSTEAD
(
       INSERT INTO employee VALUES
                (NEW.employee_id,
                 split_part(NEW.name,' ', 1),
                 split_part(NEW.name,' ', 2) );
        INSERT INTO phone VALUES
                (NEW.employee_id,
                 split_part(NEW.number,'-', 1)::INTEGER,
                 split_part(NEW.number,'-', 2)::INTEGER,
                 split_part(NEW.number,'-', 3)::INTEGER );
) ;
```

Since the directory combines the data from the base tables' columns into single columns, we must split up this data to insert it back into the underlying tables; in this case, we use the built-in database function `split_part` (added in PostgreSQL 7.3), which splits text strings based on a given delimiter. That is an important thing to be aware of; as long as you can derive a way to deconstruct the formula used in a view, you can push data back into the base table with the rule system. With this rule in place, we can now insert directly into our view:

```
rob=# INSERT INTO directory VALUES (3,'Emma Jane','352-555-6120');
INSERT 107999 1
```

The `INSERT` indicates that our `INSERT` statement succeeded, but we know that the data did not go into our `directory` view. Let's take a look at our base tables:

```
rob=# SELECT * FROM employee;
```

```
employee_id | fname | lname
------------+-------+--------
                1 | Amber | Lee
                2 | Dylan  | Jairus
                3 | Emma | Jane
(3 rows)
```

The `employee` table has our new employee, which is good, but what about their contact information?

```
rob=# SELECT * FROM phone;
```

```
employee_id | npa | nxx  | xxxx
------------+-----+-----+------
                1 | 607 | 555 | 5210
                2 | 813 | 555 | 5040
                3 | 352 | 555 | 6120
(3 rows)
```

It worked! We have inserted data into our view, and the rule split up the data and inserted it into the proper tables as needed. Of course, once you can insert into a view, you surely want to delete from it, and we can use a rule to make this work as well:

```
CREATE RULE youre_fired AS ON DELETE TO directory DO INSTEAD
DELETE FROM employee WHERE fname=split_part(OLD.name,' ', 1) AND
  lname=split_part(OLD.name,' ', 2);
```

Rules on joined tables do not have to reference all the tables in the view if you don't want them to. Notice that in this rule, we only reference the `employee` table, which works fine for our example because the entries in the `phone` table will be removed due to the cascading reference that we defined in our original table. Let's fire an employee:

```
rob=# DELETE FROM directory WHERE name = 'Amber Lee';
DELETE 1
```

Again, the database indicates that our delete was successful, so let's take a look at the base tables to verify our data:

```
rob=# SELECT * FROM employee;
```

```
employee_id | fname | lname
------------+-------+--------
                2 | Dylan  | Jairus
                3 | Emma | Jane
(2 rows)
```

The employee we wanted to delete is no longer in the `employee` table, so let's check on their contact information:

```
rob=# SELECT * FROM phone;
```

```
employee_id | npa | nxx  | xxxx
------------+-----+-----+------
                2 | 813 | 555 | 5040
                3 | 352 | 555 | 6120
(2 rows)
```

Again, we see that the rule system was able to properly pass our request on to the appropriate tables, and the entries for the employee and their phone information have been removed. Since we can now insert and delete from our view, it only makes sense that we would want to be able to update the data that we have as well. For this example, we will create a rule that allows someone to modify the phone number information, but not the name information:

```
CREATE RULE modify_employee AS
        ON UPDATE TO directory DO INSTEAD
        UPDATE phone SET
                        npa=split_part(NEW.number,'-', 1)::INTEGER,
                        nxx=split_part(NEW.number,'-', 2)::INTEGER,
                        xxxx=split_part(NEW.number,'-', 3)::INTEGER
                                  WHERE employee_id = NEW.employee_id;
```

This rule combines a number of items that we have talked about already. It interacts with only one of the base tables in a join, it reverses a complex formula used in the view definition, and it converts the data type on the fly to properly match what was defined in our base table. Now we'll update the directory:

```
rob=# UPDATE directory SET number='352-555-7120'
WHERE employee_id = 3;
UPDATE 1
```

As always, we receive a successful return code from PostgreSQL regarding our statement, but let's double-check the base tables to see what happened exactly:

```
rob=# SELECT * FROM phone;
```

```
employee_id | npa | nxx  | xxxx
------------+-----+-----+------
                2 | 813 | 555 | 5040
                3 | 352 | 555 | 7120
(2 rows)
```

As you can see, the new number is now stored in the phone table. Since we did not need to update the employee table, let's take a look at our view again to see if the changes are reflected:

```
rob=# SELECT * FROM directory;
```

```
employee_id |     name       |    number
------------+--------------+--------------
                2 | Dylan Jairus | 813-555-5040
                3 | Emma Jane  | 352-555-7120
(2 rows)
```

Of course, the rule system can be used for more than just making interactive views: You can also use it on tables, and you can do more than just directly reference tables. For the next example, we create a new table called salary and insert some information into the table:

```
CREATE TABLE salary (employee_id INTEGER REFERENCES
  employee(employee_id) ON DELETE CASCADE, salary INTEGER);
INSERT INTO salary VALUES (2,400000);
INSERT INTO salary VALUES (3,200000);
```

The first thing we decide is that we want to prevent anyone from deleting an employee's salary. Normally this would be accomplished by using the REVOKE DELETE command, but REVOKE DELETE will not prevent superusers from deleting data accidentally, so we want to go the extra step:

```
CREATE RULE always_pay AS ON DELETE TO salary DO INSTEAD NOTHING;
```

This is the INSTEAD form of a rule, and it causes the query to be rewritten so as not to be executed at all. Now, even if a superuser tries to delete from the table, deletes will be prevented:

```
rob=# DELETE FROM salary WHERE employee_id = 2;
DELETE 0
```

Another thing we might need is a log of any changes to an employee's salary that might take place. We accomplish this through the combination of a logging table and an update rule:

```
CREATE TABLE salary_log (employee_id INTEGER REFERENCES
  employee(employee_id) ON DELETE CASCADE, salary_change INTEGER,
  changed_by TEXT, log_time TIMESTAMP DEFAULT now());
CREATE RULE log_salary_changes AS ON UPDATE TO salary DO ALSO INSERT
  INTO salary_log VALUES (NEW.employee_id, NEW.salary - OLD.salary,
  CURRENT_USER);
```

Notice that this rule is of the DO ALSO variety, meaning that the original query will be executed along with the actions specified in the rule. It uses data from the original query, a mathematical operation, and the internal CURRENT_USER function, which produces the current user logged into the database:

```
rob=# UPDATE salary SET salary = 250000 WHERE employee_id = 3;
UPDATE 1
rob=# SELECT * FROM salary_log;
```

```
employee_id | salary_change | changed_by |         log_time
------------+---------------+------------+--------------------------
                3 |         50000    | rob               |2005-07-11
15:19:33.703885
(1 row)
```

One thing we haven't touched upon is that rules fire for all the rows touched by the initial query, not just one row. This can sometimes catch people off guard, but it can also be very helpful. For example, if we have to give everyone in the company a 15 percent pay cut, we can track those changes with no extra effort:

```
rob=# UPDATE salary SET salary = (salary - salary*.15) ;
UPDATE 2
rob=# select * from salary_log;
```

```
employee_id | salary_change | changed_by |         log_time
------------+---------------+------------+--------------------------
                3 |         50000    | rob               | 2005-07-11
15:19:33.703885
                2 |        -60000    | rob               | 2005-07-11
15:30:41.008909
                3 |        -37500    | rob               | 2005-07-11
15:30:41.008909
(3 rows)
```

Since we updated two rows in the table, we get two additional entries inserted in our log table, which is what we want.

Working with Views from Within PHP

Although views have some different characteristics from tables at the database level, within PHP, querying from a view is no different than querying from a table. Listing 32-1 shows a simple PHP page that queries from our directory view and displays the results onscreen. You'll notice that it is very similar to the examples used in Chapter 31 when we were querying against tables.

Listing 32-1. *Querying a View with PHP*

```
<?php
   include "pgsql.class.php";

   // Create new pgsql object
   $pgsqldb = new pgsql("localhost","rob","rob","secret");

   // Connect to the database server and select a database
   $pgsqldb->connect();

   // Query the database
   $pgsqldb->query("SELECT name, number FROM directory
                    ORDER BY name");

   // Output the data
   while ($row = $pgsqldb->fetchObject())
      echo "$row->name, $row->number<br />";
?>
```

When executed in a browser, this code will return the following results:

```
Dylan Jairus, 813-555-5040
Emma Jane, 352-555-7120
```

You can see that querying against a view is no different from querying against a table to either the PHP application developer or the end user. For this reason, as a rule of thumb, you should treat views and tables as if there is no difference when building your applications. This is especially true if you use PostgreSQL rules to make your views fully interactive.

Summary

In this chapter we took a good look at how a few of PostgreSQL's advanced features can be used to make powerful, interactive relationships within the database. We first looked at views, including how they work within the database and the commands to add and delete them. We next explained the PostgreSQL rule system, by discussing the different types of rules and giving a basic overview of how those rules can be defined. We concluded with some examples of how you can combine views, rules, and tables within PostgreSQL to make highly interactive database schemas. The examples were kept simple, but should give you some good ideas on how you could take advantage of these powerful features.