

What is PostgreSQL?

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux and Windows. It includes most SQL:2008 data types. It also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces and exceptional documentation.



PostgreSQL
www.postgresql.org



PostgreSQL and the PostgreSQL logo are trademarks of The PostgreSQL Global Development Group

About PostgreSQL

An enterprise class database, PostgreSQL boasts sophisticated features such as Multi-Version Concurrency Control (MVCC), point in time recovery, tablespaces, asynchronous replication, nested transactions (savepoints), online/hot backups, a sophisticated query planner/optimizer, and write ahead logging for fault tolerance. It supports international character sets, multibyte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting. It is highly scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate. There are active PostgreSQL systems in production environments that manage in excess of 4 terabytes of data.

Fultus™



Published by Fultus Corporation
www.fultus.com



PostgreSQL
www.postgresql.org



PostgreSQL 9.0

Server Programming

Volume III



By The PostgreSQL Global Development Group

PostgreSQL 9.0

Server Programming





PostgreSQL 9.0
Official Documentation

Volume III
Server Programming



Fultus™ Books

PostgreSQL



PostgreSQL 9.0 Official Documentation

Volume III

Server Programming

ISBN-10: 1-59682-248-1

ISBN-13: 978-1-59682-248-1

Copyright © 1996-2011 The PostgreSQL Global Development Group

Cover design and book layout by Fultus Corporation



Published by Fultus Corporation

Publisher Web: *www.fultus.com*

Linbrary - Linux Library: *www.linbrary.com*

Online Bookstore: *store.fultus.com*

email: *production@fultus.com*



This material may only be distributed subject to the terms and conditions set forth in the BSD License (presently available at <http://www.postgresql.org/about/licence>).

PostgreSQL and PostgreSQL logo are trademarks or registered trademarks of The PostgreSQL Global Development Group, in the U.S. and other countries. All product names and services identified throughout this manual are trademarks or registered trademarks of their respective companies.

The author and publisher have made every effort in the preparation of this book to ensure the accuracy of the information. However, the information contained in this book is offered without warranty, either express or implied. Neither the author nor the publisher nor any dealer or distributor will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Table of Contents

List of Tables.....	9
List of Examples	10
License.....	11
Abstract	12
Part V. Server Programming.....	13
Chapter 35. Extending SQL.....	14
35.1. How Extensibility Works	14
35.2. The PostgreSQL Type System.....	15
35.2.1. Base Types	15
35.2.2. Composite Types	15
35.2.3. Domains	15
35.2.4. Pseudo-Types	15
35.2.5. Polymorphic Types.....	15
35.3. User-Defined Functions.....	16
35.4. Query Language (SQL) Functions.....	17
35.4.1. SQL Functions on Base Types	18
35.4.2. SQL Functions on Composite Types.....	19
35.4.3. SQL Functions with Parameter Names	22
35.4.4. SQL Functions with Output Parameters	23
35.4.5. SQL Functions with Variable Numbers of Arguments	24
35.4.6. SQL Functions with Default Values for Arguments.....	25
35.4.7. SQL Functions as Table Sources	26
35.4.8. SQL Functions Returning Sets	26
35.4.9. SQL Functions Returning TABLE	28
35.4.10. Polymorphic SQL Functions	28
35.5. Function Overloading.....	30
35.6. Function Volatility Categories	31
35.7. Procedural Language Functions.....	33
35.8. Internal Functions.....	33
35.9. C-Language Functions.....	34
35.9.1. Dynamic Loading	34
35.9.2. Base Types in C-Language Functions	36

35.9.3.	Version 0 Calling Conventions.....	39
35.9.4.	Version 1 Calling Conventions.....	41
35.9.5.	Writing Code.....	44
35.9.6.	Compiling and Linking Dynamically-Loaded Functions.....	45
35.9.7.	Extension Building Infrastructure.....	48
35.9.8.	Composite-Type Arguments	50
35.9.9.	Returning Rows (Composite Types).....	52
35.9.10.	Returning Sets	54
35.9.11.	Polymorphic Arguments and Return Types	59
35.9.12.	Shared Memory and LWLocks	61
35.10.	User-Defined Aggregates.....	61
35.11.	User-Defined Types	64
35.12.	User-Defined Operators	68
35.13.	Operator Optimization Information.....	69
35.13.1.	COMMUTATOR.....	69
35.13.2.	NEGATOR.....	70
35.13.3.	RESTRICT.....	71
35.13.4.	JOIN	72
35.13.5.	HASHES.....	72
35.13.6.	MERGES.....	74
35.14.	Interfacing Extensions To Indexes	74
35.14.1.	Index Methods and Operator Classes.....	75
35.14.2.	Index Method Strategies.....	75
35.14.3.	Index Method Support Routines.....	77
35.14.4.	An Example	79
35.14.5.	Operator Classes and Operator Families	81
35.14.6.	System Dependencies on Operator Classes.....	84
35.14.7.	Special Features of Operator Classes.....	85
35.15.	Using C++ for Extensibility	86
Chapter 36.	Triggers	88
36.1.	Overview of Trigger Behavior.....	88
36.2.	Visibility of Data Changes.....	90
36.3.	Writing Trigger Functions in C	91
36.4.	A Complete Trigger Example.....	94
Chapter 37.	The Rule System.....	98
37.1.	The Query Tree.....	98
37.2.	Views and the Rule System.....	101
37.2.1.	How SELECT Rules Work.....	101
37.2.2.	View Rules in Non-SELECT Statements.....	106

Table of Contents

37.2.3. The Power of Views in PostgreSQL	107
37.2.4. Updating a View	108
37.3. Rules on INSERT, UPDATE, and DELETE	108
37.3.1. How Update Rules Work	108
37.3.1.1. A First Rule Step by Step	110
37.3.2. Cooperation with Views	113
37.4. Rules and Privileges	119
37.5. Rules and Command Status	121
37.6. Rules versus Triggers	122
Chapter 38. Procedural Languages	125
38.1. Installing Procedural Languages	125
Chapter 39. PL/pgSQL - SQL Procedural Language	128
39.1. Overview	128
39.1.1. Advantages of Using PL/pgSQL	128
39.1.2. Supported Argument and Result Data Types	129
39.2. Structure of PL/pgSQL	129
39.3. Declarations	131
39.3.1. Aliases for Function Parameters	132
39.3.2. ALIAS	134
39.3.3. Copying Types	135
39.3.4. Row Types	135
39.3.5. Record Types	136
39.4. Expressions	136
39.5. Basic Statements	137
39.5.1. Assignment	137
39.5.2. Executing a Command With No Result	137
39.5.3. Executing a Query with a Single-Row Result	138
39.5.4. Executing Dynamic Commands	140
39.5.5. Obtaining the Result Status	143
39.5.6. Doing Nothing At All	144
39.6. Control Structures	145
39.6.1. Returning From a Function	145
39.6.1.1. RETURN	145
39.6.1.2. RETURN NEXT and RETURN QUERY	145
39.6.2. Conditionals	147
39.6.2.1. IF-THEN	147
39.6.2.2. IF-THEN-ELSE	147
39.6.2.3. IF-THEN-ELSIF	148
39.6.2.4. Simple CASE	149

39.6.2.5. Searched CASE.....	149
39.6.3. Simple Loops.....	150
39.6.3.1. LOOP.....	150
39.6.3.2. EXIT.....	150
39.6.3.3. CONTINUE.....	151
39.6.3.4. WHILE.....	152
39.6.3.5. FOR (integer variant).....	152
39.6.4. Looping Through Query Results.....	153
39.6.5. Trapping Errors.....	154
39.7. Cursors.....	156
39.7.1. Declaring Cursor Variables.....	156
39.7.2. Opening Cursors.....	157
39.7.2.1. OPEN FOR query.....	157
39.7.2.2. OPEN FOR EXECUTE.....	157
39.7.2.3. Opening a Bound Cursor.....	158
39.7.3. Using Cursors.....	158
39.7.3.1. FETCH.....	159
39.7.3.2. MOVE.....	159
39.7.3.3. UPDATE/DELETE WHERE CURRENT OF.....	160
39.7.3.4. CLOSE.....	160
39.7.3.5. Returning Cursors.....	160
39.7.4. Looping Through a Cursor's Result.....	162
39.8. Errors and Messages.....	162
39.9. Trigger Procedures.....	164
39.10. PL/pgSQL Under the Hood.....	170
39.10.1. Variable Substitution.....	170
39.10.2. Plan Caching.....	172
39.11. Tips for Developing in PL/pgSQL.....	174
39.11.1. Handling of Quotation Marks.....	175
39.12. Porting from Oracle PL/SQL.....	177
39.12.1. Porting Examples.....	178
39.12.2. Other Things to Watch For.....	183
39.12.2.1. Implicit Rollback after Exceptions.....	184
39.12.2.2. EXECUTE.....	184
39.12.2.3. Optimizing PL/pgSQL Functions.....	184
39.12.3. Appendix.....	184
Chapter 40. PL/Tcl - Tcl Procedural Language.....	188
40.1. Overview.....	188
40.2. PL/Tcl Functions and Arguments.....	189

Table of Contents

40.3. Data Values in PL/Tcl	190
40.4. Global Data in PL/Tcl.....	190
40.5. Database Access from PL/Tcl.....	191
40.6. Trigger Procedures in PL/Tcl.....	194
40.7. Modules and the <code>unknown</code> command.....	196
40.8. Tcl Procedure Names	196
Chapter 41. PL/Perl - Perl Procedural Language	197
41.1. PL/Perl Functions and Arguments.....	197
41.2. Data Values in PL/Perl.....	201
41.3. Built-in Functions	201
41.3.1. Database Access from PL/Perl	201
41.3.2. Utility functions in PL/Perl.....	205
41.4. Global Values in PL/Perl	206
41.5. Trusted and Untrusted PL/Perl	207
41.6. PL/Perl Triggers.....	208
41.7. PL/Perl Under the Hood.....	210
41.7.1. Configuration	210
41.7.2. Limitations and Missing Features	212
Chapter 42. PL/Python - Python Procedural Language.....	213
42.1. Python 2 vs. Python 3.....	213
42.2. PL/Python Functions.....	215
42.3. Data Values.....	216
42.3.1. Data Type Mapping.....	216
42.3.2. Null, None	217
42.3.3. Arrays, Lists.....	218
42.3.4. Composite Types	219
42.3.5. Set-Returning Functions	220
42.4. Sharing Data.....	221
42.5. Anonymous Code Blocks	222
42.6. Trigger Functions.....	222
42.7. Database Access.....	223
42.8. Utility Functions	224
42.9. Environment Variables	224
Chapter 43. Server Programming Interface.....	225
43.1. Interface Functions	225
43.2. Interface Support Functions.....	249
43.3. Memory Management.....	254
43.4. Visibility of Data Changes.....	260
43.5. Examples.....	261

List of Volumes	264
Index.....	267
Linbrary™ Advertising Club (LAC)	273
Your Advertising Here.....	281

List of Tables

Table 35-1. Equivalent C Types for Built-In SQL Types.....	38
Table 35-2. B-tree Strategies.....	76
Table 35-3. Hash Strategies.....	76
Table 35-4. GiST Two-Dimensional "R-tree" Strategies.....	77
Table 35-5. GIN Array Strategies.....	77
Table 35-6. B-tree Support Functions.....	77
Table 35-7. Hash Support Functions.....	78
Table 35-8. GiST Support Functions.....	78
Table 35-9. GIN Support Functions.....	78

List of Examples

Example 38-1. Manual Installation of PL/pgSQL.....	127
Example 39-1. Quoting values in dynamic queries	143
Example 39-2. Exceptions with UPDATE/INSERT	156
Example 39-3. A PL/pgSQL Trigger Procedure	166
Example 39-4. A PL/pgSQL Trigger Procedure For Auditing	167
Example 39-5. A PL/pgSQL Trigger Procedure For Maintaining A Summary Table.....	170
Example 39-6. Porting a Simple Function from PL/SQL to PL/pgSQL.....	178
Example 39-7. Porting a Function that Creates Another Function from PL/SQL to PL/pgSQL.....	180
Example 39-8. Porting a Procedure With String Manipulation and OUT Parameters from PL/SQL to PL/pgSQL.....	182
Example 39-9. Porting a Procedure from PL/SQL to PL/pgSQL.....	183

License

PostgreSQL is Copyright © 1996-2011 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS-IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Abstract

Welcome to the *PostgreSQL 9.0 Official Documentation*! After many years of development, PostgreSQL has become feature-complete in many areas. This release shows a targeted approach to adding features (e.g., authentication, monitoring, space reuse), and adds capabilities defined in the later SQL standards.

Part V.

Server Programming

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* (Vol.I) to gain a more formal knowledge of the SQL language, or *Part IV* (Vol.II) for information about developing applications for PostgreSQL. Those who set up and manage their own server should also read *Part III* (Vol.II).

Chapter 37.

The Rule System

This chapter discusses the rule system in PostgreSQL. Production rule systems are conceptually simple, but there are many subtle points involved in actually using them.

Some other database systems define active database rules, which are usually stored procedures and triggers. In PostgreSQL, these can be implemented using functions and triggers as well.

The rule system (more precisely speaking, the query rewrite rule system) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The theoretical foundations and the power of this rule system are also discussed in *On Rules, Procedures, Caching and Views in Database Systems*¹ and *A Unified Framework for Version Modeling Using Production Rules in a Database System* (Vol.5 "Bibliography").

37.1. The Query Tree

To understand how the rule system works it is necessary to know when it is invoked and what its input and results are.

The rule system is located between the parser and the planner. It takes the output of the parser, one query tree, and the user-defined rewrite rules, which are also query trees with some extra information, and creates zero or more query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a query tree? It is an internal representation of an SQL statement where the single parts that it is built from are stored separately. These query trees can be shown in the server log if you set the configuration parameters `debug_print_parse`, `debug_print_rewritten`, or `debug_print_plan`. The rule actions are also stored as query

¹ <http://db.cs.berkeley.edu/papers/ERL-M90-36.pdf>

trees, in the system catalog `pg_rewrite`. They are not formatted like the log output, but they contain exactly the same information.

Reading a raw query tree requires some experience. But since SQL representations of query trees are sufficient to understand the rule system, this chapter will not teach how to read them.

When reading the SQL representations of the query trees in this chapter it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

the command type

This is a simple value telling which command (`SELECT`, `INSERT`, `UPDATE` , `DELETE`) produced the query tree.

the range table

The range table is a list of relations that are used in the query. In a `SELECT` statement these are the relations given after the `FROM` key word.

Every range table entry identifies a table or view and tells by which name it is called in the other parts of the query. In the query tree, the range table entries are referenced by number rather than by name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This can happen after the range tables of rules have been merged in. The examples in this chapter will not have this situation.

the result relation

This is an index into the range table that identifies the relation where the results of the query go.

`SELECT` queries normally don't have a result relation. The special case of a `SELECT INTO` is mostly identical to a `CREATE TABLE` followed by a `INSERT . . . SELECT` and is not discussed separately here.

For `INSERT`, `UPDATE` , and `DELETE` commands, the result relation is the table (or view!) where the changes are to take effect.

the target list

The target list is a list of expressions that define the result of the query. In the case of a `SELECT`, these expressions are the ones that build the final output of the query. They correspond to the expressions between the key words `SELECT` and `FROM`. (* is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.)

`DELETE` commands don't need a target list because they don't produce any result. In fact, the planner will add a special `CTID` entry to the empty target list, but this is after the rule system and will be discussed later; for the rule system, the target list is empty. For `INSERT` commands, the target list describes the new rows that should go into the result relation. It consists of the expressions in the `VALUES` clause or the ones from the `SELECT` clause in `INSERT . . . SELECT`. The first step of the rewrite process adds target list entries for any columns that were not assigned to by the original command but have defaults. Any remaining columns (with neither a given value nor a default) will be filled in by the planner with a constant null expression.

For `UPDATE` commands, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the `SET column = expression` part of the command. The planner will handle missing columns by inserting expressions that copy the values from the old row into the new one. And it will add the special `CTID` entry just as for `DELETE`, too.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to a column of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators, etc.

the qualification

The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (`INSERT`, `UPDATE`, `DELETE`, or `SELECT`) for the final result row should be executed or not. It corresponds to the `WHERE` clause of an SQL statement.

the join tree

The query's join tree shows the structure of the `FROM` clause. For a simple query like `SELECT . . . FROM a, b, c`, the join tree is just a list of the `FROM` items, because we are allowed to join them in any order. But when `JOIN` expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the `JOIN` expressions. The restrictions associated with particular `JOIN` clauses (from `ON` or `USING` expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the top-level `WHERE` expression as a qualification attached to the top-level join-tree item, too. So really the join tree represents both the `FROM` and `WHERE` clauses of a `SELECT`.

the others

The other parts of the query tree like the `ORDER BY` clause aren't of interest here. The rule system substitutes some entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system.

37.2. Views and the Rule System

Views in PostgreSQL are implemented using the rule system. In fact, there is essentially no difference between:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the `CREATE VIEW` command does internally. This has some side effects. One of them is that the information about a view in the PostgreSQL system catalogs is exactly the same as it is for a table. So for the parser, there is absolutely no difference between a table and a view. They are the same thing: relations.

37.2.1. How `SELECT` Rules Work

Rules `ON SELECT` are applied to all queries as the last step, even if the command given is an `INSERT`, `UPDATE` or `DELETE`. And they have different semantics from rules on the other command types in that they modify the query tree in place instead of creating a new one. So `SELECT` rules are described first.

Currently, there can be only one action in an `ON SELECT` rule, and it must be an unconditional `SELECT` action that is `INSTEAD`. This restriction was required to make rules safe enough to open them for ordinary users, and it restricts `ON SELECT` rules to act like views.

The examples for this chapter are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for `INSERT`, `UPDATE`, and `DELETE` operations so that the final result will be a view that behaves like a real table with some magic functionality. This is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

For the example, we need a little `min` function that returns the lower of 2 integer values. We create that as:

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

The real tables we need in the first two rule system descriptions are these:

```
CREATE TABLE shoe_data (
    shoename    text,          -- primary key
```

```

    sh_avail    integer,      -- available number of pairs
    slcolor     text,        -- preferred shoelace color
    slminlen    real,        -- minimum shoelace length
    slmaxlen    real,        -- maximum shoelace length
    slunit      text        -- length unit
);

CREATE TABLE shoelace_data (
    sl_name     text,        -- primary key
    sl_avail    integer,    -- available number of pairs
    sl_color    text,        -- shoelace color
    sl_len      real,        -- shoelace length
    sl_unit     text        -- length unit
);

CREATE TABLE unit (
    un_name     text,        -- primary key
    un_fact     real        -- factor to transform to cm
);

```

As you can see, they represent shoe-store data.

The views are created as:

```

CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
    FROM shoe_data sh, unit un
    WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
    WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
    FROM shoe rsh, shoelace rsl
    WHERE rsl.sl_color = rsh.slcolor

```

```
AND rsl.sl_len_cm >= rsh.slminlen_cm
AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The `CREATE VIEW` command for the `shoelace` view (which is the simplest one we have) will create a relation `shoelace` and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation `shoelace` is referenced in a query's range table. The rule has no rule qualification (discussed later, with the non-`SELECT` rules, since `SELECT` rules currently cannot have them) and it is `INSTEAD`. Note that rule qualifications are not the same as query qualifications. The action of our rule has a query qualification. The action of the rule is one query tree that is a copy of the `SELECT` statement in the view creation command.



Note

The two extra range table entries for `NEW` and `OLD` that you can see in the `pg_rewrite` entry aren't of interest for `SELECT` rules.

Now we populate `unit`, `shoe_data` and `shoelace_data` and run a simple query on a view:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);
```

```
INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
```

```
INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9, 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60, 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40, 'inch');
```

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
sl1	5	black	80	cm	80
sl2	6	black	100	cm	100
sl7	7	brown	60	cm	60
sl3	0	black	35	inch	88.9
sl4	8	black	40	inch	101.6
sl8	1	brown	40	inch	101.6
sl5	4	brown	1	m	100
sl6	0	brown	0.9	m	90

(8 rows)

This is the simplest `SELECT` you can do on our views, so we take this opportunity to explain the basics of view rules. The `SELECT * FROM shoelace` was interpreted by the parser and produced the query tree:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace;
```

and this is given to the rule system. The rule system walks through the range table and checks if there are rules for any relation. When processing the range table entry for `shoelace` (the only one up to now) it finds the `_RETURN` rule with the query tree:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace old, shoelace new,
     shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;
```

To expand the view, the rewriter simply creates a subquery range-table entry containing the rule's action query tree, and substitutes this range table entry for the original one that referenced the view. The resulting rewritten query tree is almost the same as if you had typed:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM (SELECT s.sl_name,
            s.sl_avail,
            s.sl_color,
            s.sl_len,
            s.sl_unit,
            s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
      WHERE s.sl_unit = u.un_name) shoelace;
```

There is one difference however: the subquery's range table has two extra entries `shoelace old` and `shoelace new`. These entries don't participate directly in the query, since they aren't referenced by the subquery's join tree or target list. The rewriter uses them to store the access privilege check information that was originally present in the range-table entry that referenced the view. In this way, the executor will still check that the user has proper privileges to access the view, even though there's no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining range-table entries in the top query (in this example there are no more), and it will recursively check the range-table entries in the added subquery to see if any of them reference views. (But it won't expand `old` or `new` — otherwise we'd have infinite recursion!) In this example,

there are no rewrite rules for `shoelace_data` or `unit`, so rewriting is complete and the above is the final result given to the planner.

Now we want to write a query that finds out for which shoes currently in the store we have the matching shoelaces (color and length) and where the total number of exactly matching pairs is greater or equal to two.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;
```

shoename	sh_avail	sl_name	sl_avail	total_avail
sh1	2	sl1	5	2
sh3	4	sl7	7	4

(2 rows)

The output of the parser this time is the query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM shoe_ready shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

The first rule applied will be the one for the `shoe_ready` view and it results in the query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
      WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

Similarly, the rules for `shoe` and `shoelace` are substituted into the range table of the subquery, leading to a three-level final query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM (SELECT sh.shoename,
                  sh.sh_avail,
                  sl.sl_name,
                  sl.sl_avail,
                  min(sl.sl_avail, sh.sh_avail) AS total_avail
            FROM shoelace sl, shoe sh
            WHERE sl.sl_color = sh.slcolor
                  AND sl.sl_len_cm >= sh.slminlen_cm
                  AND sl.sl_len_cm <= sh.slmaxlen_cm) shoe_ready
      WHERE shoe_ready.total_avail >= 2;
```

```

        sh.slcolor,
        sh.slminlen,
        sh.slminlen * un.un_fact AS slminlen_cm,
        sh.slmaxlen,
        sh.slmaxlen * un.un_fact AS slmaxlen_cm,
        sh.slunit
    FROM shoe_data sh, unit un
    WHERE sh.slunit = un.un_name) rsh,
    (SELECT s.sl_name,
        s.sl_avail,
        s.sl_color,
        s.sl_len,
        s.sl_unit,
        s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
    WHERE s.sl_unit = u.un_name) rsl
    WHERE rsl.sl_color = rsh.slcolor
        AND rsl.sl_len_cm >= rsh.slminlen_cm
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
    WHERE shoe_ready.total_avail > 2;

```

It turns out that the planner will collapse this tree into a two-level query tree: the bottommost `SELECT` commands will be "pulled up" into the middle `SELECT` since there's no need to process them separately. But the middle `SELECT` will remain separate from the top, because it contains aggregate functions. If we pulled those up it would change the behavior of the topmost `SELECT`, which we don't want. However, collapsing the query tree is an optimization that the rewrite system doesn't have to concern itself with.

37.2.2. View Rules in Non-`SELECT` Statements

Two details of the query tree aren't touched in the description of view rules above. These are the command type and the result relation. In fact, view rules don't need this information.

There are only a few differences between a query tree for a `SELECT` and one for any other command. Obviously, they have a different command type and for a command other than a `SELECT`, the result relation points to the range-table entry where the result should go. Everything else is absolutely the same. So having two tables `t1` and `t2` with columns `a` and `b`, the query trees for the two statements:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

are nearly identical. In particular:

- The range tables contain entries for the tables `t1` and `t2`.
- The target lists contain one variable that points to column `b` of the range table entry for table `t2`.

- The qualification expressions compare the columns `a` of both range-table entries for equality.
- The join trees show a simple join between `t1` and `t2`.

The consequence is, that both query trees result in similar execution plans: They are both joins over the two tables. For the `UPDATE` the missing columns from `t1` are added to the target list by the planner and the final query tree will read as:

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

and thus the executor run over the join will produce exactly the same result set as a:

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

will do. But there is a little problem in `UPDATE`: The executor does not care what the results from the join it is doing are meant for. It just produces a result set of rows. The difference that one is a `SELECT` command and the other is an `UPDATE` is handled in the caller of the executor. The caller still knows (looking at the query tree) that this is an `UPDATE`, and it knows that this result should go into table `t1`. But which of the rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the target list in `UPDATE` (and also in `DELETE`) statements: the current tuple ID (CTID). This is a system column containing the file block number and position in the block for the row. Knowing the table, the CTID can be used to retrieve the original row of `t1` to be updated. After adding the CTID to the target list, the query actually looks like:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of PostgreSQL enters the stage. Old table rows aren't overwritten, and this is why `ROLLBACK` is fast. In an `UPDATE`, the new result row is inserted into the table (after stripping the CTID) and in the row header of the old row, which the CTID pointed to, the `ctid` and `xmax` entries are set to the current command counter and current transaction ID. Thus the old row is hidden, and after the transaction commits the vacuum cleaner can really remove it.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

37.2.3. The Power of Views in PostgreSQL

The above demonstrates how the rule system incorporates view definitions into the original query tree. In the second example, a simple `SELECT` from one view created a final query tree that is a join of 4 tables (`unit` was used twice with different names).

The benefit of implementing views with the rule system is, that the planner has all the information about which tables have to be scanned plus the relationships between these

tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single query tree. And this is still the situation when the original query is already a join over views. The planner has to decide which is the best path to execute the query, and the more information the planner has, the better this decision can be. And the rule system as implemented in PostgreSQL ensures, that this is all information available about the query up to that point.

37.2.4. Updating a View

What happens if a view is named as the target relation for an `INSERT`, `UPDATE`, or `DELETE`? After doing the substitutions described above, we will have a query tree in which the result relation points at a subquery range-table entry. This will not work, so the rewriter throws an error if it sees it has produced such a thing.

To change this, we can define rules that modify the behavior of these kinds of commands. This is the topic of the next section.

37.3. Rules on `INSERT`, `UPDATE`, and `DELETE`

Rules that are defined on `INSERT`, `UPDATE`, and `DELETE` are significantly different from the view rules described in the previous section. First, their `CREATE RULE` command allows more:

- They are allowed to have no action.
- They can have multiple actions.
- They can be `INSTEAD` or `ALSO` (the default).
- The pseudorelations `NEW` and `OLD` become useful.
- They can have rule qualifications.

Second, they don't modify the query tree in place. Instead they create zero or more new query trees and can throw away the original one.

37.3.1. How Update Rules Work

Keep the syntax:

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

in mind. In the following, *update rules* means rules that are defined on `INSERT`, `UPDATE`, or `DELETE`.

Update rules get applied by the rule system when the result relation and the command type of a query tree are equal to the object and event given in the `CREATE RULE` command. For

update rules, the rule system creates a list of query trees. Initially the query-tree list is empty. There can be zero (`NOTHING` key word), one, or multiple actions. To simplify, we will look at a rule with one action. This rule can have a qualification or not and it can be `INSTEAD` or `ALSO` (the default).

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the pseudorelations `NEW` and/or `OLD`, which basically represent the relation that was given as object (but with a special meaning).

So we have three cases that produce the following query trees for a one-action rule.

No qualification, with either `ALSO` or `INSTEAD`

the query tree from the rule action with the original query tree's qualification added

Qualification given and `ALSO`

the query tree from the rule action with the rule qualification and the original query tree's qualification added

Qualification given and `INSTEAD`

the query tree from the rule action with the rule qualification and the original query tree's qualification; and the original query tree with the negated rule qualification added

Finally, if the rule is `ALSO`, the unchanged original query tree is added to the list. Since only qualified `INSTEAD` rules already add the original query tree, we end up with either one or two output query trees for a rule with one action.

For `ON INSERT` rules, the original query (if not suppressed by `INSTEAD`) is done before any actions added by rules. This allows the actions to see the inserted row(s). But for `ON UPDATE` and `ON DELETE` rules, the original query is done after the actions added by rules. This ensures that the actions can see the to-be-updated or to-be-deleted rows; otherwise, the actions might do nothing because they find no rows matching their qualifications.

The query trees generated from rule actions are thrown into the rewrite system again, and maybe more rules get applied resulting in more or less query trees. So a rule's actions must have either a different command type or a different result relation than the rule itself is on, otherwise this recursive process will end up in an infinite loop. (Recursive expansion of a rule will be detected and reported as an error.)

The query trees found in the actions of the `pg_rewrite` system catalog are only templates. Since they can reference the range-table entries for `NEW` and `OLD`, some substitutions have to be made before they can be used. For any reference to `NEW`, the target list of the original query is searched for a corresponding entry. If found, that entry's expression replaces the

reference. Otherwise, *NEW* means the same as *OLD* (for an *UPDATE*) or is replaced by a null value (for an *INSERT*). Any reference to *OLD* is replaced by a reference to the range-table entry that is the result relation.

After the system is done applying update rules, it applies view rules to the produced query tree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

37.3.1.1. A First Rule Step by Step

Say we want to trace changes to the *sl_avail* column in the *shoelace_data* relation. So we set up a log table and a rule that conditionally writes a log entry when an *UPDATE* is performed on *shoelace_data*.

```
CREATE TABLE shoelace_log (
    sl_name    text,           -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,         -- who did it
    log_when   timestamp      -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail <> OLD.sl_avail
DO INSERT INTO shoelace_log VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    current_user,
    current_timestamp
);
```

Now someone does:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

and we look at the log table:

```
SELECT * FROM shoelace_log;
```

```
sl_name | sl_avail | log_who | log_when
-----+-----+-----+-----
sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

That's what we expected. What happened in the background is the following. The parser created the query tree:

```
UPDATE shoelace_data SET sl_avail = 6
FROM shoelace_data shoelace_data
WHERE shoelace_data.sl_name = 'sl7';
```

There is a rule *log_shoelace* that is ON *UPDATE* with the rule qualification expression:

```
NEW.sl_avail <> OLD.sl_avail
```

and the action:

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old;
```

(This looks a little strange since you cannot normally write `INSERT ... VALUES ... FROM`. The `FROM` clause here is just to indicate that there are range-table entries in the query tree for `new` and `old`. These are needed so that they can be referenced by variables in the `INSERT` command's query tree.)

The rule is a qualified `ALSO` rule, so the rule system has to return two query trees: the modified rule action and the original query tree. In step 1, the range table of the original query is incorporated into the rule's action query tree. This results in:

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data;
```

In step 2, the rule qualification is added to it, so the result set is restricted to rows where `sl_avail` changes:

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail;
```

(This looks even stranger, since `INSERT ... VALUES` doesn't have a `WHERE` clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for `INSERT ... SELECT`.)

In step 3, the original query tree's qualification is added, restricting the result set further to only the rows that would have been touched by the original query:

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

Step 4 replaces references to `NEW` by the target list entries from the original query tree or by the matching variable references from the result relation:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
```

```

        current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';

```

Step 5 changes OLD references into result relation references:

```

INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

```

That's it. Since the rule is ALSO, we also output the original query tree. In short, the output from the rule system is a list of two query trees that correspond to these statements:

```

INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

```

```

UPDATE shoelace_data SET sl_avail = 6
WHERE sl_name = 'sl7';

```

These are executed in this order, and that is exactly what the rule was meant to do.

The substitutions and the added qualifications ensure that, if the original query would be, say:

```

UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 'sl7';

```

no log entry would get written. In that case, the original query tree does not contain a target list entry for sl_avail, so NEW.sl_avail will get replaced by shoelace_data.sl_avail. Thus, the extra command generated by the rule is:

```

INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, shoelace_data.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data
WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

```

and that qualification will never be true.

It will also work if the original query modifies multiple rows. So if someone issued the command:

```

UPDATE shoelace_data SET sl_avail = 0
WHERE sl_color = 'black';

```

four rows in fact get updated (s11, s12, s13, and s14). But s13 already has sl_avail = 0. In this case, the original query trees qualification is different and that results in the extra query tree:

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
FROM shoelace_data
WHERE 0 <> shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';
```

being generated by the rule. This query tree will surely insert three new log entries. And that's absolutely correct.

Here we can see why it is important that the original query tree is executed last. If the UPDATE had been executed first, all the rows would have already been set to zero, so the logging INSERT would not find any row where 0 <> shoelace_data.sl_avail.

37.3.2. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to run INSERT , UPDATE, or DELETE on them is to let those query trees get thrown away. So we could create the rules:

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
DO INSTEAD NOTHING;
```

If someone now tries to do any of these operations on the view relation shoe, the rule system will apply these rules. Since the rules have no actions and are INSTEAD, the resulting list of query trees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

A more sophisticated way to use the rule system is to create rules that rewrite the query tree into one that does the right operation on the real tables. To do that on the shoelace view, we create the following rules:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
);
```

```

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
DO INSTEAD
UPDATE shoelace_data
  SET sl_name = NEW.sl_name,
      sl_avail = NEW.sl_avail,
      sl_color = NEW.sl_color,
      sl_len = NEW.sl_len,
      sl_unit = NEW.sl_unit
WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
DO INSTEAD
DELETE FROM shoelace_data
WHERE sl_name = OLD.sl_name;

```

If you want to support RETURNING queries on the view, you need to make the rules include RETURNING clauses that compute the view rows. This is usually pretty trivial for views on a single table, but it's a bit tedious for join views such as shoelace. An example for the insert case is:

```

CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
  NEW.sl_name,
  NEW.sl_avail,
  NEW.sl_color,
  NEW.sl_len,
  NEW.sl_unit
)
RETURNING
  shoelace_data.*,
  (SELECT shoelace_data.sl_len * u.un_fact
   FROM unit u WHERE shoelace_data.sl_unit = u.un_name);

```

Note that this one rule supports both INSERT and INSERT RETURNING queries on the view - the RETURNING clause is simply ignored for INSERT.

Now assume that once in a while, a pack of shoelaces arrives at the shop and a big parts list along with it. But you don't want to manually update the shoelace view every time. Instead we setup two little tables: one where you can insert the items from the part list, and one with a special trick. The creation commands for these are:

```

CREATE TABLE shoelace_arrive (
  arr_name  text,
  arr_quant integer
);

CREATE TABLE shoelace_ok (
  ok_name   text,
  ok_quant  integer
);

```

```
CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
DO INSTEAD
UPDATE shoelace
SET sl_avail = sl_avail + NEW.ok_quant
WHERE sl_name = NEW.ok_name;
```

Now you can fill the table `shoelace_arrive` with the data from the parts list:

```
SELECT * FROM shoelace_arrive;
```

arr_name	arr_quant
s13	10
s16	20
s18	20

(3 rows)

Take a quick look at the current data:

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s13	0	black	35	inch	88.9
s14	8	black	40	inch	101.6
s18	1	brown	40	inch	101.6
s15	4	brown	1	m	100
s16	0	brown	0.9	m	90

(8 rows)

Now move the arrived shoelaces in:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

and check the results:

```
SELECT * FROM shoelace ORDER BY sl_name;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s14	8	black	40	inch	101.6
s13	10	black	35	inch	88.9
s18	21	brown	40	inch	101.6
s15	4	brown	1	m	100
s16	20	brown	0.9	m	90

(8 rows)

```
SELECT * FROM shoelace_log;
```

sl_name	sl_avail	log_who	log_when
sl7	6	Al	Tue Oct 20 19:14:45 1998 MET DST
sl3	10	Al	Tue Oct 20 19:25:16 1998 MET DST
sl6	20	Al	Tue Oct 20 19:25:16 1998 MET DST
sl8	21	Al	Tue Oct 20 19:25:16 1998 MET DST

(4 rows)

It's a long way from the one `INSERT ... SELECT` to these results. And the description of the query-tree transformation will be the last in this chapter. First, there is the parser's output:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Now the first rule `shoelace_ok_ins` is applied and turns this into:

```
UPDATE shoelace
SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace
WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

and throws away the original `INSERT` on `shoelace_ok`. This rewritten query is passed to the rule system again, and the second applied rule `shoelace_upd` produces:

```
UPDATE shoelace_data
SET sl_name = shoelace.sl_name,
sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
sl_color = shoelace.sl_color,
sl_len = shoelace.sl_len,
sl_unit = shoelace.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data
WHERE shoelace.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = shoelace.sl_name;
```

Again it's an `INSTEAD` rule and the previous query tree is trashed. Note that this query still uses the view `shoelace`. But the rule system isn't finished with this step, so it continues and applies the `_RETURN` rule on it, and we get:

```
UPDATE shoelace_data
SET sl_name = s.sl_name,
sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
sl_color = s.sl_color,
sl_len = s.sl_len,
sl_unit = s.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data,
```

```

    shoelace old, shoelace new,
    shoelace_data s, unit u
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name;

```

Finally, the rule `log_shoelace` gets applied, producing the extra query tree:

```

INSERT INTO shoelace_log
SELECT s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    current_user,
    current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
    shoelace_ok old, shoelace_ok new,
    shoelace shoelace, shoelace old,
    shoelace new, shoelace_data shoelace_data,
    shoelace old, shoelace new,
    shoelace_data s, unit u,
    shoelace_data old, shoelace_data new
    shoelace_log shoelace_log
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name
    AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;

```

After that the rule system runs out of rules and returns the generated query trees.

So we end up with two final query trees that are equivalent to the SQL statements:

```

INSERT INTO shoelace_log
SELECT s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    current_user,
    current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
    shoelace_data s
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name
    AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
    SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive,
    shoelace_data shoelace_data,
    shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
    AND shoelace_data.sl_name = s.sl_name;

```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries, it turns out that the `shoelace_data` relation appears twice in the range table where it could definitely be

reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the `INSERT` will be

```
Nested Loop
-> Merge Join
    -> Seq Scan
        -> Sort
            -> Seq Scan on s
    -> Seq Scan
        -> Sort
            -> Seq Scan on shoelace_arrive
-> Seq Scan on shoelace_data
```

while omitting the extra range table entry would result in a

```
Merge Join
-> Seq Scan
    -> Sort
        -> Seq Scan on s
-> Seq Scan
    -> Sort
        -> Seq Scan on shoelace_arrive
```

which produces exactly the same entries in the log table. Thus, the rule system caused one extra scan on the table `shoelace_data` that is absolutely not necessary. And the same redundant scan is done once more in the `UPDATE`. But it was a really hard job to make that all possible at all.

Now we make a final demonstration of the PostgreSQL rule system and its power. Say you add some shoelaces with extraordinary colors to your database:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

We would like to make a view to check which `shoelace` entries do not fit any shoe in color. The view for this is:

```
CREATE VIEW shoelace_mismatch AS
  SELECT * FROM shoelace WHERE NOT EXISTS
    (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Its output is:

```
SELECT * FROM shoelace_mismatch;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
sl9	0	pink	35	inch	88.9
sl10	1000	magenta	40	inch	101.6

Now we want to set it up so that mismatching shoelaces that are not in stock are deleted from the database. To make it a little harder for PostgreSQL, we don't delete it directly. Instead we create one more view:

```
CREATE VIEW shoelace_can_delete AS
  SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
  (SELECT * FROM shoelace_can_delete
   WHERE sl_name = shoelace.sl_name);
```

Voila:

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s14	8	black	40	inch	101.6
s13	10	black	35	inch	88.9
s18	21	brown	40	inch	101.6
s110	1000	magenta	40	inch	101.6
s15	4	brown	1	m	100
s16	20	brown	0.9	m	90

(9 rows)

A `DELETE` on a view, with a subquery qualification that in total uses 4 nesting/joined views, where one of them itself has a subquery qualification containing a view and where calculated view columns are used, gets rewritten into one single query tree that deletes the requested data from a real table.

There are probably only a few situations out in the real world where such a construct is necessary. But it makes you feel comfortable that it works.

37.4. Rules and Privileges

Due to rewriting of queries by the PostgreSQL rule system, other tables/views than those used in the original query get accessed. When update rules are used, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The PostgreSQL rule system changes the behavior of the default access control system. Relations that are used due to rules get checked against the privileges of the rule owner, not the user invoking the rule. This means that a user only needs the required privileges for the tables/views that he names explicitly in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
```

```
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the `phone_data` table. But because of the `GRANT`, the secretary can run a `SELECT` on the `phone_number` view. The rule system will rewrite the `SELECT` from `phone_number` into a `SELECT` from `phone_data`. Since the user is the owner of `phone_number` and therefore the owner of the rule, the read access to `phone_data` is now checked against his privileges and the query is permitted. The check for accessing `phone_number` is also performed, but this is done against the invoking user, so nobody but the user and the secretary can use it.

The privileges are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to the public. Then, anyone can see the `phone_number` data through the secretary's view. What the secretary cannot do is to create a view that directly accesses `phone_data`. (Actually he can, but it will not work since every access will be denied during the permission checks.) And as soon as the user will notice, that the secretary opened his `phone_number` view, he can revoke his access. Immediately, any access to the secretary's view would fail.

One might think that this rule-by-rule checking is a security hole, but in fact it isn't. But if it did not work this way, the secretary could set up a table with the same columns as `phone_number` and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A `GRANT` command means, "I trust you". If someone you trust does the thing above, it's time to think it over and then use `REVOKE`.

Note that while views can be used to hide the contents of certain columns using the technique shown above, they cannot be used to reliably conceal the data in unseen rows. For example, the following view is insecure:

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

This view might seem secure, since the rule system will rewrite any `SELECT` from `phone_number` into a `SELECT` from `phone_data` and add the qualification that only entries where `phone` does not begin with 412 are wanted. But if the user can create his or her own functions, it is not difficult to convince the planner to execute the user-defined function prior to the `NOT LIKE` expression.

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END
$$ LANGUAGE plpgsql COST 0.000000000000000000000001;
SELECT * FROM phone_number WHERE tricky(person, phone);
```

Every person and phone number in the `phone_data` table will be printed as a `NOTICE`, because the planner will choose to execute the inexpensive `tricky` function before the more expensive `NOT LIKE`. Even if the user is prevented from defining new functions, built-in functions can be used in similar attacks. (For example, casting functions include their inputs in the error messages they produce.)

Similar considerations apply to update rules. In the examples of the previous section, the owner of the tables in the example database could grant the privileges `SELECT`, `INSERT`, `UPDATE`, and `DELETE` on the `shoelace` view to someone else, but only `SELECT` on `shoelace_log`. The rule action to write log entries will still be executed successfully, and that other user could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones. In this case, there is no possibility of subverting the rules by convincing the planner to alter the order of operations, because the only rule which references `shoelace_log` is an unqualified `INSERT`. This might not be true in more complex scenarios.

37.5. Rules and Command Status

The PostgreSQL server returns a command status string, such as `INSERT 149592 1`, for each command it receives. This is simple enough when there are no rules involved, but what happens when the query is rewritten by rules?

Rules affect the command status as follows:

- If there is no unconditional `INSTEAD` rule for the query, then the originally given query will be executed, and its command status will be returned as usual. (But note that if there were any conditional `INSTEAD` rules, the negation of their qualifications will have been added to the original query. This might reduce the number of rows it processes, and if so the reported status will be affected.)
- If there is any unconditional `INSTEAD` rule for the query, then the original query will not be executed at all. In this case, the server will return the command status for the last query that was inserted by an `INSTEAD` rule (conditional or unconditional) and is of the same command type (`INSERT`, `UPDATE`, or `DELETE`) as the original query. If no query meeting those requirements is added by any rule, then the returned command status shows the original query type and zeroes for the row-count and OID fields.

(This system was established in PostgreSQL 7.3. In versions before that, the command status might show different results when rules exist.)

The programmer can ensure that any desired `INSTEAD` rule is the one that sets the command status in the second case, by giving it the alphabetically last rule name among the active rules, so that it gets applied last.

37.6. Rules versus Triggers

Many things that can be done using triggers can also be implemented using the PostgreSQL rule system. One of the things that cannot be implemented by rules are some kinds of constraints, especially foreign keys. It is possible to place a qualified rule that rewrites a command to `NOTHING` if the value of a column does not appear in another table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger.

On the other hand, a trigger cannot be created on views because there is no real data in a view relation; however `INSERT`, `UPDATE`, and `DELETE` rules can be created on views.

For the things that can be implemented by both, which is best depends on the usage of the database. A trigger is fired for any affected row once. A rule manipulates the query or generates an additional query. So if many rows are affected in one statement, a rule issuing one extra command is likely to be faster than a trigger that is called for every single row and must execute its operations many times. However, the trigger approach is conceptually far simpler than the rule approach, and is easier for novices to get right.

Here we show an example of how the choice of rules versus triggers plays out in one situation. There are two tables:

```
CREATE TABLE computer (  
    hostname      text,      -- indexed  
    manufacturer  text      -- indexed  
);
```

```
CREATE TABLE software (  
    software      text,      -- indexed  
    hostname      text      -- indexed  
);
```

Both tables have many thousands of rows and the indexes on `hostname` are unique. The rule or trigger should implement a constraint that deletes rows from `software` that reference a deleted computer. The trigger would use this command:

```
DELETE FROM software WHERE hostname = $1;
```

Since the trigger is called for each individual row deleted from `computer`, it can prepare and save the plan for this command and pass the `hostname` value in the parameter. The rule would be written as:

```
CREATE RULE computer_del AS ON DELETE TO computer  
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a:

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table `computer` is scanned by index (fast), and the command issued by the trigger would also use an index scan (also fast). The extra command from the rule would be:

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                        AND software.hostname = computer.hostname;
```

Since there are appropriate indexes setup, the planner will create a plan of

```
Nestloop
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation.

With the next delete we want to get rid of all the 2000 computers where the `hostname` starts with `old`. There are two possible commands to do that. One is:

```
DELETE FROM computer WHERE hostname >= 'old'
                        AND hostname < 'ole'
```

The command added by the rule will be:

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                        AND software.hostname = computer.hostname;
```

with the plan

```
Hash Join
-> Seq Scan on software
-> Hash
-> Index Scan using comp_hostidx on computer
```

The other possible command is:

```
DELETE FROM computer WHERE hostname ~ '^old';
```

which results in the following executing plan for the command added by the rule:

```
Nestloop
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

This shows, that the planner does not realize that the qualification for `hostname` in `computer` could also be used for an index scan on `software` when there are multiple qualification expressions combined with `AND`, which is what it does in the regular-expression version of the command. The trigger will get invoked once for each of the 2000 old computers that have to be deleted, and that will result in one index scan over `computer` and 2000 index scans over `software`. The rule implementation will do it with two commands that use indexes. And it depends on the overall size of the table `software` whether the rule will still be faster in the sequential scan situation. 2000 command executions from the trigger over the SPI manager take some time, even if all the index blocks will soon be in the cache.

The last command we look at is:

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

Again this could result in many rows to be deleted from `computer`. So the trigger will again run many commands through the executor. The command generated by the rule will be:

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                        AND software.hostname = computer.hostname;
```

The plan for that command will again be the nested loop over two index scans, only using a different index on `computer`:

```
Nestloop
-> Index Scan using comp_manufidx on computer
-> Index Scan using soft_hostidx on software
```

In any of these cases, the extra commands from the rule system will be more or less independent from the number of affected rows in a command.

The summary is, rules will only be significantly slower than triggers if their actions result in large and badly qualified joins, a situation where the planner fails.

List of Volumes

Volume I.

Preface

- What is PostgreSQL?
- A Brief History of PostgreSQL
- Conventions
- Further Information
- Bug Reporting Guidelines

Part I. Tutorial

1. Getting Started
2. The SQL Language
3. Advanced Features

Part II. The SQL Language

4. SQL Syntax
5. Data Definition
6. Data Manipulation
7. Queries
8. Data Types
9. Functions and Operators
10. Type Conversion
11. Indexes
12. Full Text Search
13. Concurrency Control
14. Performance Tips

Volume II.

Part III. Server Administration

15. Installation from Source Code
16. Installation from Source Code on Windows
17. Server Setup and Operation

List of Volumes

18. Server Configuration
19. Client Authentication
20. Database Roles and Privileges
21. Managing Databases
22. Localization
23. Routine Database Maintenance Tasks
24. Backup and Restore
25. High Availability, Load Balancing, and Replication
26. Recovery Configuration
27. Monitoring Database Activity
28. Monitoring Disk Usage
29. Reliability and the Write-Ahead Log
30. Regression Tests

Part IV. Client Interfaces

31. libpq - C Library
32. Large Objects
33. ECPG - Embedded SQL in C
34. The Information Schema

Volume III.

Part V. Server Programming

35. Extending SQL
36. Triggers
37. The Rule System
38. Procedural Languages
39. PL/pgSQL - SQL Procedural Language
40. PL/Tcl - Tcl Procedural Language
41. PL/Perl - Perl Procedural Language
42. PL/Python - Python Procedural Language
43. Server Programming Interface

Volume IV.

Part VI. Reference

- I. SQL Commands
- II. PostgreSQL Client Applications
- III. PostgreSQL Server Applications

Volume V.

Part VII. Internals

- 44. Overview of PostgreSQL Internals
- 45. System Catalogs
- 46. Frontend/Backend Protocol
- 47. PostgreSQL Coding Conventions
- 48. Native Language Support
- 49. Writing A Procedural Language Handler
- 50. Genetic Query Optimizer
- 51. Index Access Method Interface Definition
- 52. GiST Indexes
- 53. GIN Indexes
- 54. Database Physical Storage
- 55. BKI Backend Interface
- 56. How the Planner Uses Statistics

Part VIII. Appendixes

- A. PostgreSQL Error Codes
- B. Date/Time Support
- C. SQL Key Words
- D. SQL Conformance
- E. Release Notes
- F. Additional Supplied Modules
- G. External Projects
- H. The CVS Repository
- I. Documentation
- J. Acronyms

Bibliography

Index

Index

<p>\$libdir.....34</p> <p>_PG_fini.....34</p> <p>_PG_init.....34</p> <p>aggregate function</p> <p> user-defined.....61</p> <p>any.....15</p> <p>anyarray15</p> <p>anyelement.....15</p> <p>anyenum.....15</p> <p>anynonarray.....15</p> <p>ARRAY</p> <p> of user-defined type.....64</p> <p>base type.....15</p> <p>BSD/OS</p> <p> shared library.....45</p> <p>C++.....87</p> <p>composite type15, 219</p> <p>computed field19</p> <p>CONTINUE</p> <p> in PL/pgSQL151</p> <p>cstring15</p> <p>ctid.....106</p> <p>cursor</p> <p> in PL/pgSQL156, 157, 158, 160</p> <p>data type.....129</p> <p> base.....15</p> <p> category128, 188</p> <p> composite15</p> <p> internal organization36</p> <p> user-defined.....64</p>	<p>decode_bytea</p> <p> in PL/Perl205</p> <p>Digital UNIX</p> <p> See Tru64 UNIX.....45</p> <p>dynamic loading34</p> <p>dynamic_library_path.....34</p> <p>elog</p> <p> in PL/Perl205</p> <p> in PL/Python.....224</p> <p> in PL/Tcl191</p> <p>encode_array_constructor</p> <p> in PL/Perl205</p> <p>encode_array_literal</p> <p> in PL/Perl205</p> <p>encode_bytea</p> <p> in PL/Perl205</p> <p>environment variable224</p> <p>exceptions</p> <p> in PL/PgSQL154</p> <p>EXIT</p> <p> in PL/pgSQL150</p> <p>extending SQL.....14</p> <p>field</p> <p> computed19</p> <p>FreeBSD</p> <p> shared library45</p> <p>function</p> <p> default values for arguments25</p> <p> internal.....33</p> <p> named parameter22</p>
---	--

output parameter	23	MacOS X	
polymorphic	15	shared library	45
RETURNS TABLE	28	magic block.....	34
type resolution		MANPATH.....	224
in an invocation.....	28, 33, 184, 201, 215, 220, 225, 249	memory context	
user-defined.....	16	in SPI.....	254
in C.....	34	NetBSD	
in SQL.....	17	shared library	45
variadic.....	24	null value	
with SETOF.....	26	in PL/Perl	197
global data		PL/Python.....	217
in PL/Python.....	221	opaque.....	15
in PL/Tcl.....	190	OpenBSD	
history		shared library	45
of PostgreSQL.....	265	operator	
HP-UX		user-defined.....	68
shared library	45	operator class.....	75, 81
IMMUTABLE	31	operator family.....	81
index		Oracle	
for user-defined data type	75	porting	
input function.....	64	from PL/SQL to PL/pgSQL.....	177
of a data type.....	64	ordering operator.....	85
instr	178	output function	64
internal	15	of a data type.....	64
IRIX		overloading	
shared library	45	functions.....	30
language_handler	15	operators	68
library finalization function	34	malloc	44
library initialization function	34	PATH.....	224
Linux		perl	197
shared library	45	pfree.....	44
looks_like_number		pg_config	
in PL/Perl	205	with user-defined C functions	44
loop		PGAPPNAME.....	224
in PL/pgSQL.....	150	PGCLIENTENCODING	224
		PGCONNECT_TIMEOUT.....	224
		PGDATABASE.....	224

Index

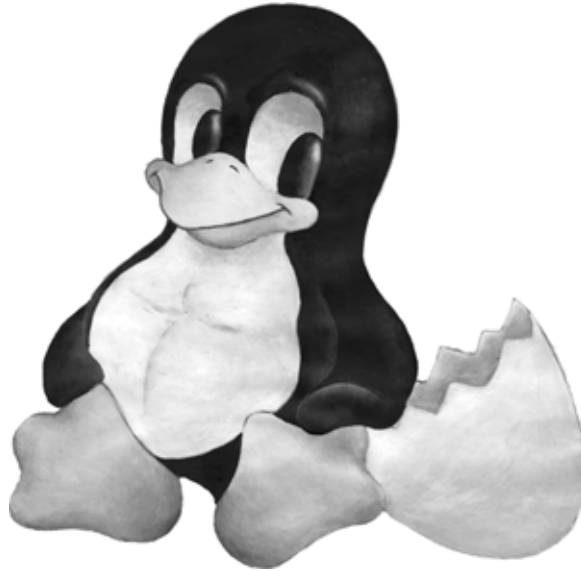
PGDATESTYLE.....	224	plperl.on_plperlu_init configuration parameter	210
PGGEQO	224	plperl.use_strict configuration parameter	210
PGSSLIB	224	plpgsql.variable_conflict configuration parameter	170
PGHOST.....	224	polymorphic function.....	15
PGHOSTADDR.....	224	polymorphic type.....	15
PGKRBSRVNAME.....	224	preparing a query in PL/pgSQL	172
PGLOCALEDIR	224	in PL/Python.....	223
PGOPTIONS.....	224	in PL/Tcl	191
PGPASSFILE.....	224	privilege with rules	119
PGPASSWORD	224	with views.....	119
PGPORT	224	procedural language.....	125
PGREALM	224	externally maintained.....	125
PGREQUIRESSL.....	224	Python.....	213
PGSERVICE	224	query tree	98
PGSERVICEFILE.....	224	quote_ident in PL/Perl	205
PGSSLCERT	224	use in PL/PgSQL	140
PGSSLCRL	224	quote_literal in PL/Perl	205
PGSSLKEY	224	use in PL/PgSQL	140
PGSSLMODE.....	224	quote_nullable in PL/Perl	205
PGSSLROOTCERT.....	224	use in PL/PgSQL	140
PGSYSCONFDIR	224	RAISE.....	162
PGTZ.....	224	range table.....	98
PGUSER.....	224	record.....	15
pgxs.....	48	reporting errors in PL/PgSQL	162
PIC.....	45	RETURN NEXT in PL/PgSQL	145
PL/Perl.....	197	RETURN QUERY in PL/PgSQL	145
PL/PerlU.....	207		
PL/pgSQL.....	128		
PL/Python	213		
PL/SQL (Oracle) porting to PL/pgSQL	177		
PL/Tcl.....	188		
plperl.on_init configuration parameter	210		
plperl.on_plperl_init configuration parameter	210		



RETURNING INTO	
in PL/pgSQL	138
row type	15, 219
rule	98
and views	101
compared with triggers.....	122
for DELETE.....	108
for INSERT.....	108
for SELECT	101
for UPDATE	108
schema.....	128, 188
SELECT INTO	
in PL/pgSQL	138
shared library	45
shared_preload_libraries	61
Solaris	
shared library	45
SPI	225
SPI_copytuple.....	256
SPI_cursor_close	247
in PL/Perl	201
SPI_cursor_fetch	244
SPI_cursor_find.....	244
SPI_cursor_move	245
SPI_cursor_open	240
SPI_cursor_open_with_args.....	241
SPI_cursor_open_with_paramlist	243
SPI_exec.....	231
spi_exec_prepared	
in PL/Perl	201
spi_exec_query	
in PL/Perl	201
SPI_execp	239
SPI_execute	228
SPI_execute_plan	237
SPI_execute_plan_with_paramlist	239
SPI_execute_with_args	231
spi_fetchrow	
in PL/Perl	201
SPI_fname	249
SPI_fnumber	250
SPI_freeplan	
in PL/Perl	201
SPI_freetuple	259
SPI_freetuptable.....	259
SPI_getargcount	236
SPI_getargtypeid.....	236
SPI_getbinval.....	251
SPI_getnspname.....	253
SPI_getrelname	253
SPI_gettype	252
SPI_gettypeid	252
SPI_getvalue	250
SPI_is_cursor_plan	237
spi_lastoid.....	191
SPI_modifytuple	258
SPI_palloc.....	255
SPI_pfree	256
SPI_pop	227
SPI_prepare	233
in PL/Perl	201
SPI_prepare_cursor	234
SPI_prepare_params	235
SPI_push	227
spi_query	
in PL/Perl	201
spi_query_prepared	
in PL/Perl	201
SPI_realloc	255
SPI_returntuple.....	257
SPI_saveplan	248
SPI_scroll_cursor_fetch.....	246
SPI_scroll_cursor_move.....	247
STABLE.....	31

Index

suppress_redundant_updates_trigger..	222	type	
target list.....	98	polymorphic	15
Tcl.....	188	See data type.....	129
TOAST		unaccent.....	28, 33, 184, 201, 215, 220, 225, 249
and user-defined types.....	64	UnixWare	
trigger	15, 88, 208	shared library	45
arguments for trigger functions.....	88	variadic function	24
compared with rules.....	122	view	
in C.....	91	implementation through rules	101
in PL/pgSQL	164	updating.....	113
in PL/Python.....	222	void	15
in PL/Tcl	194	VOLATILE	31
Tru64 UNIX		volatility	
shared library.....	45	functions.....	31
trusted		WHILE	
PL/Perl.....	207	in PL/pgSQL	152

Linbrary™ Advertising Club (LAC)



Linbrary™  Official Docs as a Real Books <http://www.linbrary.com>  **Linux Library**



Linbrary Advertising Club

Advertising



Linux Documentation Project - Machtelt Garrels

<http://www.tldp.org/>

Version	Title	Edition	ISBN- 10	ISBN- 13
TLDP	Introduction to Linux (Third Edition)	paperback	1-59682-199-X	978-1-59682-199-6
		eBook (pdf)	1-59682-200-7	978-1-59682-200-9
	Bash Guide for Beginners (Second Edition)	paperback	1-59682-201-5	978-1-59682-201-6
		eBook (pdf)	1-59682-202-3	978-1-59682-202-3
<i>http://www.linbrary.com/linux-tldp/</i>				



Linbrary Advertising Club



Fedora Project Official Documentation

<http://docs.fedoraproject.org>

Version	Title	Edition	ISBN- 10	ISBN- 13
Fedora 14	Fedora 14 Installation Guide	paperback	1-59682-228-7	978-1-59682-228-3
		eBook (pdf)	1-59682-233-3	978-1-59682-233-7
	Fedora 14 User Guide	paperback	1-59682-229-5	978-1-59682-229-0
		eBook (pdf)	1-59682-234-1	978-1-59682-234-4
	Fedora 14 Security Guide	paperback	1-59682-230-9	978-1-59682-230-6
		eBook (pdf)	1-59682-235-X	978-1-59682-235-1
	Fedora 14 Storage Administration Guide	paperback	1-59682-231-7	978-1-59682-231-3
		eBook (pdf)	1-59682-236-8	978-1-59682-236-8
Fedora 14 Musicians Guide	paperback	1-59682-232-5	978-1-59682-232-0	
	eBook (pdf)	1-59682-237-6	978-1-59682-237-5	
Fedora 13	Fedora 13 Installation Guide	paperback	1-59682-212-0	978-1-59682-212-2
		eBook (pdf)	1-59682-217-1	978-1-59682-217-7
	Fedora 13 User Guide	paperback	1-59682-213-9	978-1-59682-213-9
		eBook (pdf)	1-59682-218-X	978-1-59682-218-4
	Fedora 13 Security Guide	paperback	1-59682-214-7	978-1-59682-214-6
		eBook (pdf)	1-59682-219-8	978-1-59682-219-1
	Fedora 13 SE Linux User Guide	paperback	1-59682-215-5	978-1-59682-215-3
		eBook (pdf)	1-59682-220-1	978-1-59682-220-7
	Fedora 13 Virtualization Guide	paperback	1-59682-216-3	978-1-59682-216-0
		eBook (pdf)	1-59682-221-X	978-1-59682-221-4



Linbrary Advertising Club



Fedora Project Official Documentation

<http://docs.fedoraproject.org>

Version	Title	Edition	ISBN- 10	ISBN- 13
Fedora 12	Fedora 12 Installation Guide	paperback	1-59682-179-5	978-1-59682-179-8
		eBook (pdf)	1-59682-184-1	978-1-59682-184-2
	Fedora 12 User Guide	paperback	1-59682-180-9	978-1-59682-180-4
		eBook (pdf)	1-59682-185-X	978-1-59682-185-9
	Fedora 12 Security Guide	paperback	1-59682-181-7	978-1-59682-181-1
		eBook (pdf)	1-59682-186-8	978-1-59682-186-6
	Fedora 12 SE Linux User Guide	paperback	1-59682-182-5	978-1-59682-182-8
		eBook (pdf)	1-59682-187-6	978-1-59682-187-3
Fedora 12 Virtualization Guide	paperback	1-59682-183-3	978-1-59682-183-5	
	eBook (pdf)	1-59682-188-4	978-1-59682-188-0	
Fedora 11	Fedora 11 Installation Guide	paperback	1-59682-142-6	978-1-59682-142-2
		eBook (pdf)	1-59682-146-9	978-1-59682-146-0
	Fedora 11 User Guide	paperback	1-59682-143-4	978-1-59682-143-9
		eBook (pdf)	1-59682-147-7	978-1-59682-147-7
	Fedora 11 Security Guide	paperback	1-59682-144-2	978-1-59682-144-6
		eBook (pdf)	1-59682-148-5	978-1-59682-148-4
	Fedora 11 SE Linux User Guide	paperback	1-59682-145-0	978-1-59682-145-3
		eBook (pdf)	1-59682-149-3	978-1-59682-149-1
http://www.linlibrary.com/fedora/				



Linlibrary Advertising Club



Ubuntu Official Documentation

<http://www.ubuntu.com/>

Version	Title	Edition	ISBN- 10	ISBN- 13
Ubuntu 10.10	Ubuntu 10.10 Installation Guide	paperback	1-59682-238-4	978-1-59682-238-2
		eBook (pdf)	1-59682-242-2	978-1-59682-242-9
	Ubuntu 10.10 Desktop Guide	paperback	1-59682-239-2	978-1-59682-239-9
		eBook (pdf)	1-59682-243-0	978-1-59682-243-6
	Ubuntu 10.10 Server Guide	paperback	1-59682-240-6	978-1-59682-240-5
		eBook (pdf)	1-59682-244-9	978-1-59682-244-3
Ubuntu 10.10 Packaging Guide	paperback	1-59682-241-4	978-1-59682-241-2	
	eBook (pdf)	1-59682-245-7	978-1-59682-245-0	
Ubuntu 10.04 LTS	Ubuntu 10.04 LTS Installation Guide	paperback	1-59682-203-1	978-1-59682-203-0
		eBook (pdf)	1-59682-207-4	978-1-59682-207-8
	Ubuntu 10.04 LTS Desktop Guide	paperback	1-59682-204-X	978-1-59682-204-7
		eBook (pdf)	1-59682-208-2	978-1-59682-208-5
	Ubuntu 10.04 LTS Server Guide	paperback	1-59682-205-8	978-1-59682-205-4
		eBook (pdf)	1-59682-209-0	978-1-59682-209-2
	Ubuntu 10.04 LTS Packaging Guide	paperback	1-59682-206-6	978-1-59682-206-1
		eBook (pdf)	1-59682-210-4	978-1-59682-210-8



Linbrary Advertising Club



Ubuntu Official Documentation

<http://www.ubuntu.com/>

Version	Title	Edition	ISBN- 10	ISBN- 13
Ubuntu 9.10	Ubuntu 9.10 Installation Guide	paperback	1-59682-171-X	978-1-59682-171-2
		eBook (pdf)	1-59682-175-2	978-1-59682-175-0
	Ubuntu 9.10 Desktop Guide	paperback	1-59682-172-8	978-1-59682-172-9
		eBook (pdf)	1-59682-176-0	978-1-59682-176-7
	Ubuntu 9.10 Server Guide	paperback	1-59682-173-6	978-1-59682-173-6
		eBook (pdf)	1-59682-177-9	978-1-59682-177-4
Ubuntu 9.10 Packaging Guide	paperback	1-59682-174-4	978-1-59682-174-3	
	eBook (pdf)	1-59682-178-7	978-1-59682-178-1	
Ubuntu 9.04	Ubuntu 9.04 Installation Guide	paperback	1-59682-150-7	978-1-59682-150-7
		eBook (pdf)	1-59682-154-X	978-1-59682-154-5
	Ubuntu 9.04 Desktop Guide	paperback	1-59682-151-5	978-1-59682-151-4
		eBook (pdf)	1-59682-155-8	978-1-59682-155-2
	Ubuntu 9.04 Server Guide	paperback	1-59682-152-3	978-1-59682-152-1
		eBook (pdf)	1-59682-156-6	978-1-59682-156-9
	Ubuntu 9.04 Packaging Guide	paperback	1-59682-153-1	978-1-59682-153-8
		eBook (pdf)	1-59682-157-4	978-1-59682-157-6
<i>http://www.linbrary.com/ubuntu/</i>				



Linbrary Advertising Club



PostgreSQL Official Documentation

<http://www.postgresql.org/>

Version	Title	Edition	ISBN- 10	ISBN- 13
PostgreSQL 9.0	PostgreSQL 9.0 Volume I. The SQL Language	paperback	1-59682-246-5	978-1-59682-246-7
		eBook (pdf)	1-59682-251-1	978-1-59682-251-1
	PostgreSQL 9.0 Volume II. Server Administration	paperback	1-59682-247-3	978-1-59682-247-4
		eBook (pdf)	1-59682-252-X	978-1-59682-252-8
	PostgreSQL 9.0 Volume III. Server Programming	paperback	1-59682-248-1	978-1-59682-248-1
		eBook (pdf)	1-59682-253-8	978-1-59682-253-5
	PostgreSQL 9.0 Volume IV. Reference	paperback	1-59682-249-X	978-1-59682-249-8
		eBook (pdf)	1-59682-254-6	978-1-59682-254-2
	PostgreSQL 9.0 Volume V. Internals & Appendixes	paperback	1-59682-250-3	978-1-59682-250-4
		eBook (pdf)	1-59682-255-4	978-1-59682-255-9
PostgreSQL 8.04	PostgreSQL 8.04 Volume I. The SQL Language	paperback	1-59682-158-2	978-1-59682-158-3
		eBook (pdf)	1-59682-163-9	978-1-59682-163-7
	PostgreSQL 8.04 Volume II. Server Administration	paperback	1-59682-159-0	978-1-59682-159-0
		eBook (pdf)	1-59682-164-7	978-1-59682-164-4
	PostgreSQL 8.04 Volume III. Server Programming	paperback	1-59682-160-4	978-1-59682-160-6
		eBook (pdf)	1-59682-165-5	978-1-59682-165-1
	PostgreSQL 8.04 Volume IV. Reference	paperback	1-59682-161-2	978-1-59682-161-3
		eBook (pdf)	1-59682-166-3	978-1-59682-166-8
	PostgreSQL 8.04 Volume V. Internals & Appendixes	paperback	1-59682-162-0	978-1-59682-162-0
		eBook (pdf)	1-59682-167-1	978-1-59682-167-5
http://www.linlibrary.com/postgresql/				




Linlibrary Advertising Club



The Apache Software Foundation Official Documentation

<http://www.apache.org/>

Version	Title	Edition	ISBN- 10	ISBN- 13
Apache Web Server 2.2	Apache HTTP Server 2.2 Vol.I. Server Administration	paperback	1-59682-191-4	978-1-59682-191-0
		eBook (pdf)	1-59682-195-7	978-1-59682-195-8
	Apache HTTP Server 2.2 Vol.II. Security & Server Programs	paperback	1-59682-192-2	978-1-59682-192-7
		eBook (pdf)	1-59682-196-5	978-1-59682-196-5
	Apache HTTP Server 2.2 Vol.III. Modules (A-H)	paperback	1-59682-193-0	978-1-59682-193-4
		eBook (pdf)	1-59682-197-3	978-1-59682-197-2
	Apache HTTP Server 2.2 Vol.IV. Modules (I-V)	paperback	1-59682-194-9	978-1-59682-194-1
		eBook (pdf)	1-59682-198-1	978-1-59682-198-9
<i>http://www.linbrary.com/apache-http/</i>				

Version	Title	Edition	ISBN- 10	ISBN- 13
Subversion 1.6	Subversion 1.6 Version Control with Subversion	paperback	1-59682-169-8	978-1-59682-169-9
		eBook (pdf)	1-59682-170-1	978-1-59682-170-5
				
<i>http://www.linbrary.com/subversion/</i>				

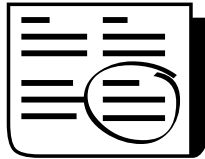


Linbrary Advertising Club

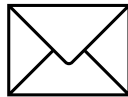


Linbrary Advertising Club

Your Advertising Here



More Books Coming Soon!!!



Please Feel Free to Contact Us at

production@fultus.com