

CHAPTER 32



PostgreSQL

Ioannis Alagiannis and Renata Borovica-Gajic

PostgreSQL is an open-source object-relational database management system. It is a descendant of one of the earliest such systems, the POSTGRES system developed under Professor Michael Stonebraker at the University of California, Berkeley. The name “postgres” is derived from the name of a pioneering relational database system, Ingres, also developed under Stonebraker at Berkeley. Currently, PostgreSQL supports many aspects of SQL:2003 and offers features such as complex queries, foreign keys, triggers, views, transactional integrity, full-text searching, and limited data replication. In addition, users can extend PostgreSQL with new data types, functions, operators, or index methods. PostgreSQL supports a variety of programming languages (including C, C++, Java, Perl, Tcl, and Python) as well as the database interfaces JDBC and ODBC. Another notable point of PostgreSQL is that it, along with MySQL, is one of the two most widely used open-source relational database systems. PostgreSQL is released under the BSD license, which grants permission to anyone for the use, modification, and distribution of the PostgreSQL code and documentation for any purpose without fee.

32.1 Introduction

In the course of two decades, PostgreSQL has undergone several major releases. The first prototype system, under the name POSTGRES, was demonstrated at the 1988 ACM SIGMOD conference. The first version, distributed to users in 1989, provided features such as extensible data types, a preliminary rule system, and a query language named POSTQUEL. After the subsequent versions added a new rule system, support for multiple storage managers, and an improved query executor, the system developers focused on portability and performance until 1994, when an SQL language interpreter was added. Under a new name, Postgres95, the system was released to the Web and later commercialized by Illustra Information Technologies (later merged into Informix, which is now owned by IBM). By 1996, the name Postgres95 was replaced by PostgreSQL, to

reflect the relationship between the original POSTGRES and the more recent versions with SQL capability.

PostgreSQL runs under virtually all Unix-like operating systems, including Linux and Apple Macintosh OS X. Early versions of the PostgreSQL server can be run under Microsoft Windows in the Cygwin environment, which provides Linux emulation under Windows. Version 8.0, released in January 2005, introduced native support for Microsoft Windows.

Today, PostgreSQL is used to implement several different research and production applications (such as the PostGIS system for geographic information) and an educational tool at several universities. The system continues to evolve through the contributions of a community of about 1000 developers. In this chapter, we explain how PostgreSQL works, starting from user interfaces and languages and continuing into the heart of the system (the data structures and the concurrency-control mechanism).

32.2 User Interfaces

The standard distribution of PostgreSQL comes with command-line tools for administering the database. However, there is a wide range of commercial and open-source graphical administration and design tools that support PostgreSQL. Software developers may also access PostgreSQL through a comprehensive set of programming interfaces.

32.2.1 Interactive Terminal Interfaces

Like most database systems, PostgreSQL offers command-line tools for database administration. The main interactive terminal client is `psql`, which is modeled after the Unix shell and allows execution of SQL commands on the server, as well as several other operations (such as client-side copying). Some of its features are:

- **Variables.** `psql` provides variable substitution features, similar to common Unix command shells.
- **SQL interpolation.** The user can substitute (“interpolate”) `psql` variables into regular SQL statements by placing a colon in front of the variable name.
- **Command-line editing.** `psql` uses the GNU readline library for convenient line editing, with tab-completion support.

PostgreSQL may also be accessed from a Tcl/Tk shell, which provides a flexible scripting language commonly used for rapid prototyping. This functionality is enabled in Tcl/Tk by loading the `pgtcl` library, which is distributed as an optional extension to PostgreSQL.

32.2.2 Graphical Interfaces

The standard distribution of PostgreSQL does not contain any graphical tools. However, several graphical user interface tools exist, and users can choose among commer-

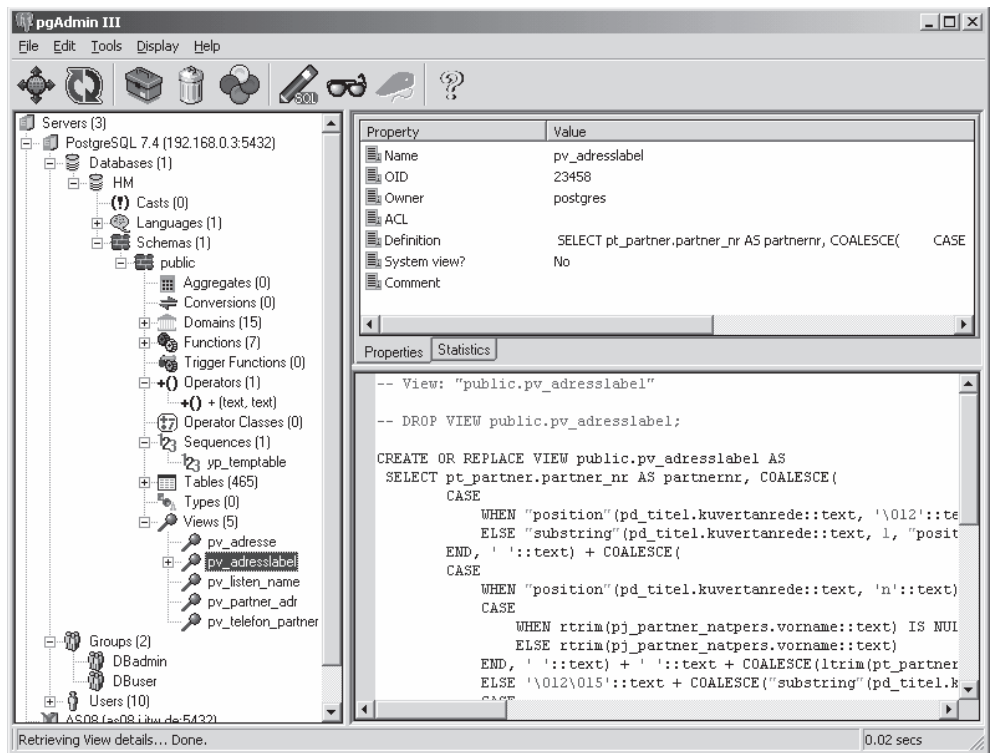


Figure 32.1 pgAdmin III: An open-source database administration GUI.

cial and open-source alternatives. Many of these go through rapid release cycles; the following list reflects the state of affairs at the time of this writing.

There are graphical tools for administration, including pgAccess and pgAdmin, the latter of which is shown in Figure 32.1. Tools for database design include TORA and Data Architect, the latter of which is shown in Figure 32.2. PostgreSQL works with several commercial forms-design and report-generation tools. Open-source alternatives include Rekall (shown in Figure 32.3 and Figure 32.4), GNU Report Generator, and a more comprehensive tool suite, GNU Enterprise.

32.2.3 Programming Language Interfaces

PostgreSQL provides native interfaces for ODBC and JDBC, as well as bindings for most programming languages, including C, C++, PHP, Perl, Tcl/Tk, ECPG, Python, and Ruby.

The libpq library provides the C API for PostgreSQL; libpq is also the underlying engine for most programming-language bindings. The libpq library supports both synchronous and asynchronous execution of SQL commands and prepared statements, through a reentrant and thread-safe interface. The connection parameters of libpq

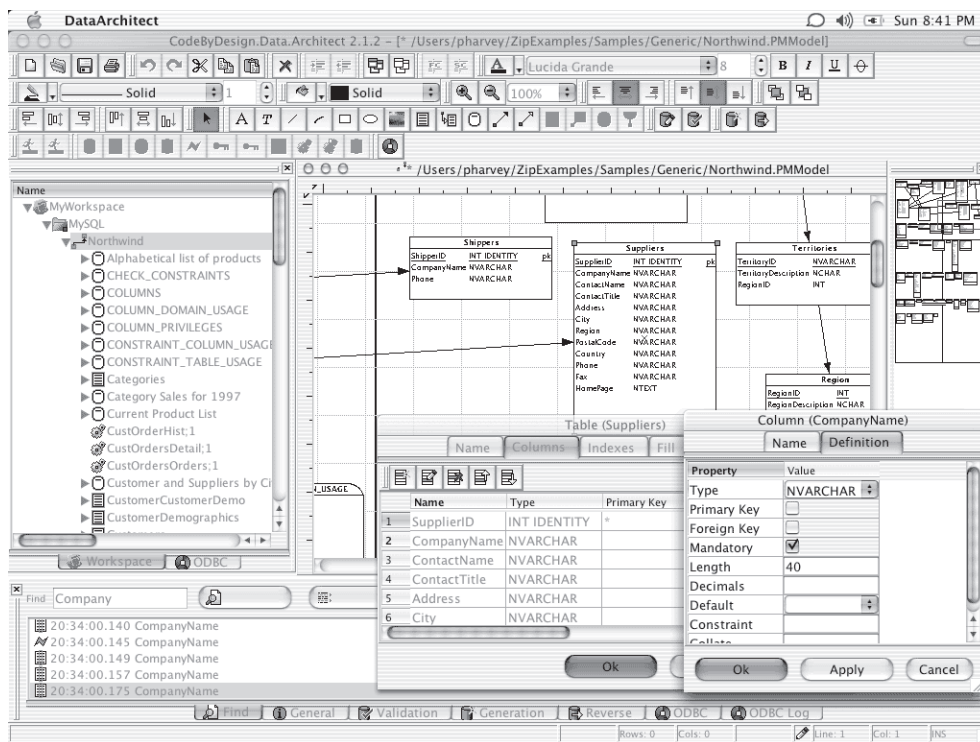


Figure 32.2 Data Architect: A multiplatform database design GUI.

may be configured in several flexible ways, such as setting environment variables, placing settings in a local file, or creating entries on an LDAP server.

32.3 SQL Variations and Extensions

The current version of PostgreSQL supports almost all entry-level SQL-92 features, as well as many of the intermediate- and full-level features. It also supports many SQL:1999 and SQL:2003 features, including most object-relational features described in Chapter 29 and the SQL/XML features for parsed XML data described in Chapter 30. In fact, some features of the current SQL standard (such as arrays, functions, and inheritance) were pioneered by PostgreSQL or its ancestors. It lacks OLAP features (most notably, **cube** and **rollup**), but data from PostgreSQL can be easily loaded into open-source external OLAP servers (such as Mondrian) as well as commercial products.

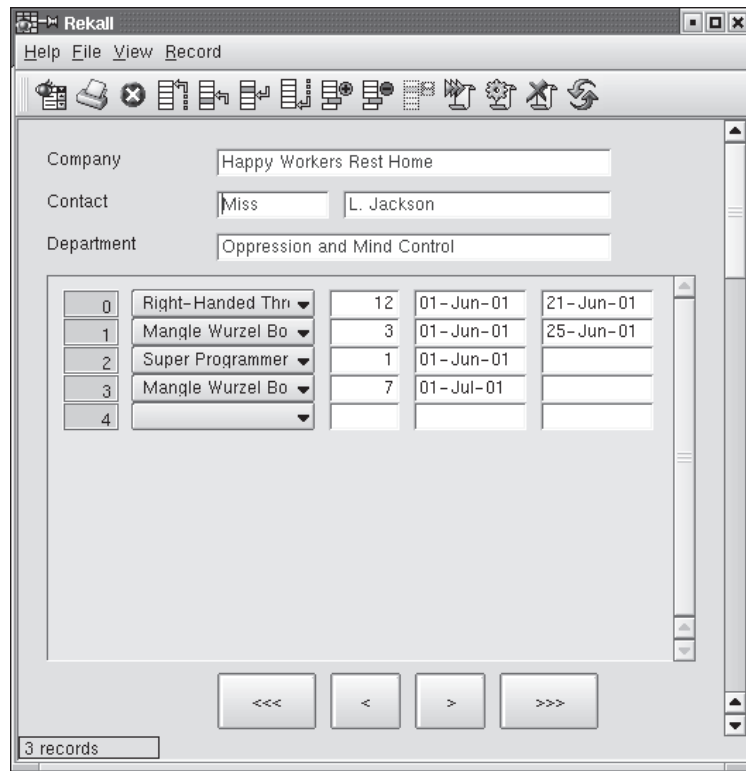


Figure 32.3 Rekall: Form-design GUI.

32.3.1 PostgreSQL Types

PostgreSQL has support for several nonstandard types, useful for specific application domains. Furthermore, users can define new types with the **create type** command. This includes new low-level base types, typically written in C (see Section 32.3.3.1).

32.3.1.1 The PostgreSQL Type System

PostgreSQL types fall into the following categories:

- **Base types.** Base types are also known as **abstract data types**; that is, modules that encapsulate both state and a set of operations. These are implemented below the SQL level, typically in a language such as C (see Section 32.3.3.1). Examples are **int4** (already included in PostgreSQL) or **complex** (included as an optional extension type). A base type may represent either an individual scalar value or a variable-length array of values. For each scalar type that exists in a database, PostgreSQL automatically creates an array type that holds values of the same scalar type.

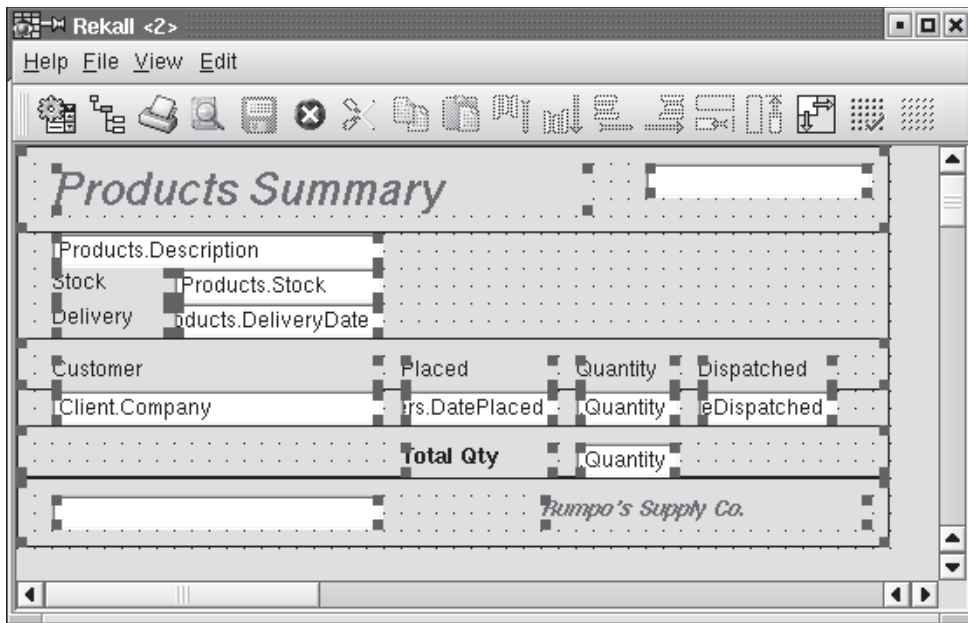


Figure 32.4 ReCALL: Report-design GUI.

- **Composite types.** These correspond to table rows; that is, they are a list of field names and their respective types. A composite type is created implicitly whenever a table is created, but users may also construct them explicitly.
- **Domains.** A domain type is defined by coupling a base type with a constraint that values of the type must satisfy. Values of the domain type and the associated base type may be used interchangeably, provided that the constraint is satisfied. A domain may also have an optional default value, whose meaning is similar to the default value of a table column.
- **Enumerated types.** These are similar to enum types used in programming languages such as C and Java. An enumerated type is essentially a fixed list of named values. In PostgreSQL, enumerated types may be converted to the textual representation of their name, but this conversion must be specified explicitly in some cases to ensure type safety. For instance, values of different enumerated types may not be compared without explicit conversion to compatible types.
- **Pseudotypes.** Currently, PostgreSQL supports the following pseudotypes: *any*, *anyarray*, *anyelement*, *anyenum*, *anynonarray*, *cstring*, *internal*, *opaque*, *language_handler*, *record*, *trigger*, and *void*. These cannot be used in composite types (and thus cannot be used for table columns), but can be used as argument and return types of user-defined functions.

- **Polymorphic types.** Four of the pseudotypes *anyelement*, *anyarray*, *anynonarray*, and *anyenum* are collectively known as **polymorphic**. Functions with arguments of these types (correspondingly called **polymorphic functions**) may operate on any actual type. PostgreSQL has a simple type-resolution scheme that requires that: (1) in any particular invocation of a polymorphic function, all occurrences of a polymorphic type must be bound to the same actual type (that is, a function defined as $f(\textit{anyelement}, \textit{anyelement})$ may operate only on pairs of the same actual type), and (2) if the return type is polymorphic, then at least one of the arguments must be of the same polymorphic type.

32.3.1.2 Nonstandard Types

The types described in this section are included in the standard distribution. Furthermore, thanks to the open nature of PostgreSQL, there are several contributed extension types, such as complex numbers, and ISBN/ISSNs (see Section 32.3.3).

Geometric data types (*point*, *line*, *lseg*, *box*, *polygon*, *path*, *circle*) are used in geographic information systems to represent two-dimensional spatial objects such as points, line segments, polygons, paths, and circles. Numerous functions and operators are available in PostgreSQL to perform various geometric operations such as scaling, translation, rotation, and determining intersections. Furthermore, PostgreSQL supports indexing of these types using R-trees (Section 24.4.2 and Section 32.5.2.1).

Full-text searching is performed in PostgreSQL using the *tsvector* type that represents a document and the *tsquery* type that represents a full-text query. A *tsvector* stores the distinct words in a document, after converting variants of each word to a common normal form (for example, removing word stems). PostgreSQL provides functions to convert raw text to a *tsvector* and concatenate documents. A *tsquery* specifies words to search for in candidate documents, with multiple words connected by Boolean operators. For example, the query ‘index & !(tree | hash)’ finds documents that contain “index” without using the words “tree” or “hash.” PostgreSQL natively supports operations on full-text types, including language features and indexed search.

PostgreSQL offers data types to store network addresses. These data types allow network-management applications to use a PostgreSQL database as their data store. For those familiar with computer networking, we provide a brief summary of this feature here. Separate types exist for IPv4, IPv6, and Media Access Control (MAC) addresses (*cidr*, *inet* and *macaddr*, respectively). Both *inet* and *cidr* types can store IPv4 and IPv6 addresses, with optional subnet masks. Their main difference is in input/output formatting, as well as the restriction that classless Internet domain routing (CIDR) addresses do not accept values with nonzero bits to the right of the netmask. The *macaddr* type is used to store MAC addresses (typically, Ethernet card hardware addresses). PostgreSQL supports indexing and sorting on these types, as well as a set of operations (including subnet testing, and mapping MAC addresses to hardware manufacturer names). Furthermore, these types offer input-error checking. Thus, they are preferable over plain text fields.

The PostgreSQL *bit* type can store both fixed- and variable-length strings of 1s and 0s. PostgreSQL supports bit-logical operators and string-manipulation functions for these values.

32.3.2 Rules and Other Active-Database Features

PostgreSQL supports SQL constraints and triggers (and stored procedures; see Section 32.3.3). Furthermore, it features query-rewriting rules that can be declared on the server.

PostgreSQL allows **check** constraints, **not null** constraints, and primary-key and foreign-key constraints (with restricting and cascading deletes).

Like many other relational database systems, PostgreSQL supports triggers, which are useful for nontrivial constraints and consistency checking or enforcement. Trigger functions can be written in a procedural language such as PL/pgSQL (see Section 32.3.3.4) or in C, but not in plain SQL. Triggers can execute before or after **insert**, **update**, or **delete** operations and either once per modified row, or once per SQL statement.

The PostgreSQL rules system allows users to define query-rewrite rules on the database server. Unlike stored procedures and triggers, the rule system intervenes between the query parser and the planner and modifies queries on the basis of the set of rules. After the original query tree has been transformed into one or more trees, they are passed to the query planner. Thus, the planner has all the necessary information (tables to be scanned, relationships between them, qualifications, join information, and so forth) and can come up with an efficient execution plan, even when complex rules are involved.

The general syntax for declaring rules is:

```
create rule rule_name as
  on { select | insert | update | delete }
to table [ where rule_qualification ]
do [ instead ] { nothing | command | ( command ; command ... ) }
```

The rest of this section provides examples that illustrate the rule system's capabilities. More details on how rules are matched to query trees and how the latter are subsequently transformed can be found in the PostgreSQL documentation (see the bibliographical notes). The rule system is implemented in the rewrite phase of query processing and explained in Section 32.6.1.

First, PostgreSQL uses rules to implement views. A view definition such as:

```
create view myview as select * from mytab;
```

is converted into the following rule definition:

```
create table myview (same column list as mytab);  
create rule _return as on select to myview do instead  
  select * from mytab;
```

Queries on *myview* are transformed before execution to queries on the underlying table *mytab*. The **create view** syntax is considered better programming form in this case, since it is more concise and it also prevents creation of views that reference each other (which is possible if rules are carelessly declared, resulting in potentially confusing runtime errors). However, rules can be used to define update actions on views explicitly (**create view** statements do not allow this).

As another example, consider the case where the user wants to log all increases of instructor salaries. This could be achieved by a rule such as:

```
create rule salary_audit as on update to instructor  
  where new.salary <> old.salary  
  do insert into salary_audit  
  values (current_timestamp, current_user,  
         new.name, old.salary, new.salary);
```

Finally, we give a slightly more complicated insert/update rule. Assume that pending salary increases are stored in a table *salary_increases(name, increase)*. We can declare a “dummy” table *approved_increases* with the same fields and then define the following rule:

```
create rule approved_increases_insert  
  as on insert to approved_increases  
  do instead  
    update instructor  
      set salary = salary + new.increase  
      where name = new.name;
```

Then the following query:

```
insert into approved_increases select * from salary_increases;
```

will update all salaries in the *instructor* table at once. Since the **instead** keyword was specified in the rule, the *approved_increases* table is unchanged.

There is some overlap between the functionality provided by rules and per-row triggers. The PostgreSQL rule system can be used to implement most triggers, but some kinds of constraints (in particular, foreign keys) cannot be implemented by rules. Also, triggers have the added ability to generate error messages to signal constraint violations, whereas a rule may only enforce data integrity by silently suppressing invalid values. On the other hand, triggers cannot be used for the **update** or **delete** actions that rules

enable on views. Since there is no real data in a view relation, the trigger would never be called.

An important difference between triggers and views is that a trigger is executed iteratively for every affected row. A rule, on the other hand, manipulates the query tree before query planning. So if a statement affects many rows, a rule is far more efficient than a trigger.

The implementation of triggers and constraints in PostgreSQL is outlined briefly in Section 32.6.4.

32.3.3 Extensibility

Like most relational database systems, PostgreSQL stores information about databases, tables, columns, and so forth, in what are commonly known as **system catalogs**, which appear to the user as normal tables. Other relational database systems are typically extended by changing hard-coded procedures in the source code or by loading special extension modules written by the vendor.

Unlike most relational database systems, PostgreSQL goes one step further and stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. Therefore, PostgreSQL is easy for users to extend and facilitates rapid prototyping of new applications and storage structures. PostgreSQL can also incorporate user-written code into the server, through dynamic loading of shared objects. This provides an alternative approach to writing extensions that can be used when catalog-based extensions are not sufficient.

Furthermore, the `contrib` module of the PostgreSQL distribution includes numerous user functions (for example, array iterators, fuzzy string matching, cryptographic functions), base types (for example, encrypted passwords, ISBN/ISSNs, n -dimensional cubes) and index extensions (for example, RD-trees, indexing for hierarchical labels). Thanks to the open nature of PostgreSQL, there is a large community of PostgreSQL professionals and enthusiasts who also actively extend PostgreSQL on an almost daily basis. Extension types are identical in functionality to the built-in types (see also Section 32.3.1.2); the latter are simply already linked into the server and preregistered in the system catalog. Similarly, this is the only difference between built-in and extension functions.

32.3.3.1 Types

PostgreSQL allows users to define composite types, enumeration types, and even new base types.

A composite-type definition is similar to a table definition (in fact, the latter implicitly does the former). Stand-alone composite types are typically useful for function arguments. For example, the definition:

```
create type city_t as (name varchar(80), state char(2));
```

allows functions to accept and return *city_t* tuples, even if there is no table that explicitly contains rows of this type.

Enumeration types are easy to define, by simply listing the names of the values. The following example creates an enumerated type to represent the status of a software product.

```
create type status_t as enum ('alpha', 'beta', 'release');
```

The order of listed names is significant in comparing values of an enumerated type. This can be useful for a statement such as:

```
select name from products
where status > 'alpha';
```

which retrieves the names of products that have moved past the alpha stage.

Adding base types to PostgreSQL is straightforward; an example can be found in *complex.sql* and *complex.c* in the tutorials of the PostgreSQL distribution. The base type can be declared in C, for example:

```
typedef struct Complex {
    double x;
    double y;
} Complex;
```

The next step is to define functions to read and write values of the new type in text format (see Section 32.3.3.2). Subsequently, the new type can be registered using the statement:

```
create type complex {
    internallength = 16,
    input = complex_in,
    output = complex_out,
    alignment = double
};
```

assuming the text I/O functions have been registered as *complex_in* and *complex_out*.

The user has the option of defining binary I/O functions as well (for more efficient data dumping). Extension types can be used like the existing base types of PostgreSQL. In fact, their only difference is that the extension types are dynamically loaded and linked into the server. Furthermore, indices may be extended easily to handle new base types; see Section 32.3.3.3.

32.3.3.2 Functions

PostgreSQL allows users to define functions that are stored and executed on the server. PostgreSQL also supports function overloading (that is, functions may be declared by using the same name but with arguments of different types). Functions can be written as plain SQL statements, or in several procedural languages (covered in Section 32.3.3.4). Finally, PostgreSQL has an application programmer interface for adding functions written in C (explained in this section).

User-defined functions can be written in C (or a language with compatible calling conventions, such as C++). The actual coding conventions are essentially the same for dynamically loaded, user-defined functions, as well as for internal functions (which are statically linked into the server). Hence, the standard internal function library is a rich source of coding examples for user-defined C functions. Once the shared library containing the function has been created, a declaration such as the following registers it on the server:

```
create function complex_out(complex)
returns cstring
as 'shared_object_filename'
language C immutable strict;
```

The entry point to the shared object file is assumed to be the same as the SQL function name (here, *complex_out*), unless otherwise specified.

The example here continues the one from Section 32.3.3.1. The application program interface hides most of PostgreSQL's internal details. Hence, the actual C code for the above text output function of *complex* values is quite simple:

```
PG_FUNCTION_INFO_V1(complex_out);
Datum complex_out(pg_function_args) {
    Complex *complex = (Complex *) pg_getarg_pointer(0);
    char *result;
    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    pg_return_cstring(result);
}
```

The first line declares the function *complex_out*, and the following lines implement the output function. The code uses several PostgreSQL-specific constructs, such as the *palloc* function, which dynamically allocates memory controlled by PostgreSQL's memory manager. More details may be found in the PostgreSQL documentation (see the bibliographical notes).

Aggregate functions in PostgreSQL operate by updating a **state value** via a **state transition** function that is called for each tuple value in the aggregation group. For example, the state for the **avg** operator consists of the running sum and the count of values. As each tuple arrives, the transition function should simply add its value to the running sum and increment the count by one. Optionally, a *final* function may be called to compute the return value based on the state information. For example, the final function for **avg** would simply divide the running sum by the count and return it.

Thus, defining a new aggregate is as simple as defining these two functions. For the *complex* type example, if *complex_add* is a user-defined function that takes two complex arguments and returns their sum, then the **sum** aggregate operator can be extended to complex numbers using the simple declaration:

```
create aggregate sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = '(0,0)'
);
```

Note the use of function overloading: PostgreSQL will call the appropriate *sum* aggregate function, on the basis of the actual type of its argument during invocation. The *basetype* is the argument type and *stype* is the state value type. In this case, a final function is unnecessary, since the return value is the state value itself (that is, the running sum in both cases).

User-defined functions can also be invoked by using operator syntax. Beyond simple “syntactic sugar” for function invocation, operator declarations can also provide hints to the query optimizer in order to improve performance. These hints may include information about commutativity, restriction and join selectivity estimation, and various other properties related to join algorithms.

32.3.3.3 Index Extensions

PostgreSQL currently supports the usual B-tree and hash indices, as well as two index methods that are unique to PostgreSQL: the Generalized Search Tree (GiST) and the Generalized Inverted Index (GIN), which is useful for full-text indexing (these index structures are explained in Section 32.5.2.1). Finally, PostgreSQL provides indexing of two-dimensional spatial objects with an R-tree index, which is implemented using a GiST index behind the scenes. All of these can be easily extended to accommodate new base types.

Adding index extensions for a type requires definition of an **operator class**, which encapsulates the following:

- **Index-method strategies.** These are a set of operators that can be used as qualifiers in **where** clauses. The particular set depends on the index type. For example, B-tree indices can retrieve ranges of objects, so the set consists of five operators ($<$, $<=$, $=$, $>=$, and $>$), all of which can appear in a **where** clause involving a B-tree index. A hash index allows only equality testing and an R-tree index allows a number of spatial relationships (for example contained, to-the-left, and so forth).
- **Index-method support routines.** The above set of operators is typically not sufficient for the operation of the index. For example, a hash index requires a function to compute the hash value for each object. An R-tree index needs to be able to compute intersections and unions and to estimate the size of indexed objects.

For example, if the following functions and operators are defined to compare the magnitude of *complex* numbers (see Section 32.3.3.1), then we can make such objects indexable by the following declaration:

```
create operator class complex_abs_ops
default for type complex using btree as
    operator 1 < (complex, complex),
    operator 2 <= (complex, complex),
    operator 3 = (complex, complex),
    operator 4 >= (complex, complex),
    operator 5 > (complex, complex),
    function 1 complex_abs_cmp(complex, complex);
```

The **operator** statements define the strategy methods and the **function** statements define the support methods.

32.3.3.4 Procedural Languages

Stored functions and procedures can be written in a number of procedural languages. Furthermore, PostgreSQL defines an application programmer interface for hooking up any programming language for this purpose. Programming languages can be registered on demand and are either **trusted** or **untrusted**. The latter allow unlimited access to the DBMS and the file system, and writing stored functions in them requires superuser privileges.

- **PL/pgSQL.** This is a trusted language that adds procedural programming capabilities (for example, variables and control flow) to SQL. It is very similar to Oracle's PL/SQL. Although code cannot be transferred verbatim from one to the other, porting is usually simple.
- **PL/Tcl, PL/Perl, and PL/Python.** These leverage the power of Tcl, Perl, and Python to write stored functions and procedures on the server. The first two come in both

trusted and untrusted versions (PL/Tcl, PL/Perl and PL/TclU, PL/PerlU, respectively), while PL/Python is untrusted at the time of this writing. Each of these has bindings that allow access to the database system via a language-specific interface.

32.3.3.5 Server Programming Interface

The server programming interface (SPI) is an application programmer interface that allows user-defined C functions (see Section 32.3.3.2) to run arbitrary SQL commands inside their functions. This gives writers of user-defined functions the ability to implement only essential parts in C and easily leverage the full power of the relational database system engine to do most of the work.

32.4 Transaction Management in PostgreSQL

Transaction management in PostgreSQL uses both both snapshot isolation and two-phase locking. Which one of the two protocols is used depends on the type of statement being executed. For DML statements¹ the snapshot isolation technique presented in Section 18.8 is used; the snapshot isolation scheme is referred to as the multiversion concurrency control (MVCC) scheme in PostgreSQL. Concurrency control for DDL statements, on the other hand, is based on standard two-phase locking.

32.4.1 PostgreSQL Concurrency Control

Since the concurrency control protocol used by PostgreSQL depends on the *isolation level* requested by the application, we begin with an overview of the isolation levels offered by PostgreSQL. We then describe the key ideas behind the MVCC scheme, followed by a discussion of their implementation in PostgreSQL and some of the implications of MVCC. We conclude this section with an overview of locking for DDL statements and a discussion of concurrency control for indices.

32.4.1.1 PostgreSQL Isolation Levels

The SQL standard defines three weak levels of consistency, in addition to the serializable level of consistency, on which most of the discussion in this book is based. The purpose of providing the weak consistency levels is to allow a higher degree of concurrency for applications that don't require the strong guarantees that serializability provides. Examples of such applications include long-running transactions that collect statistics over the database and whose results do not need to be precise.

¹A DML statement is any statement that updates or reads data within a table, that is, **select**, **insert**, **update**, **fetch**, and **copy**. DDL statements affect entire tables; they can remove a table or change the schema of a table, for example. DDL statements and some other PostgreSQL-specific statements will be discussed later in this section.

<i>Isolated level</i>	<i>Dirty Read</i>	<i>Non repeatable Read</i>	<i>Phantom Read</i>
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeated Read	No	No	Maybe
Serializable	No	No	No

Figure 32.5 Definition of the four standard SQL isolation levels.

The SQL standard defines the different isolation levels in terms of three phenomena that violate serializability. The three phenomena are called *dirty read*, *nonrepeatable read*, and *phantom read*, and are defined as follows:

- **Dirty read.** The transaction reads values written by another transaction that hasn't committed yet.
- **Nonrepeatable read.** A transaction reads the same object twice during execution and finds a different value the second time, although the transaction has not changed the value in the meantime.
- **Phantom read.** A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed as a result of another recently committed transaction. (A more detailed explanation of the phantom phenomenon, including the concept of a phantom conflict, can be found in Section 18.4.3; eliminating phantom reads does not eliminate all phantom conflicts.)

It should be obvious that each of the above phenomena violates transaction isolation, and hence violates serializability. Figure 32.5 shows the definition of the four SQL isolation levels specified in the SQL standard—read uncommitted, read committed, repeatable read, and serializable—in terms of these phenomena. PostgreSQL supports two of the four different isolation levels, read committed (which is the default isolation level in PostgreSQL) and serializable. However, the PostgreSQL implementation of the serializable isolation level uses snapshot isolation, which does not truly ensure serializability as we have seen earlier in Section 18.8.

32.4.1.2 Concurrency Control for DML Commands

The MVCC scheme used in PostgreSQL is an implementation of the snapshot isolation protocol which we saw in Section 18.8. The key idea behind MVCC is to maintain different versions of each row that correspond to instances of the row at different points in time. This allows a transaction to see a consistent **snapshot** of the data, by selecting the most recent version of each row that was committed before taking the snapshot. The MVCC protocol uses snapshots to ensure that every transaction sees a consistent view of the database: before executing a command, the transaction chooses a snapshot

of the data and processes the row versions that are either in the snapshot or created by earlier commands of the same transaction. This view of the data is “consistent” since it only takes full transactions into account, but the snapshot is not necessarily equal to the current state of the data.

The motivation for using MVCC is that readers never block writers and vice versa. Readers access the most recent version of a row that is part of the transaction’s snapshot. Writers create their own separate copy of the row to be updated. Section 32.4.1.3 shows that the only conflict that causes a transaction to be blocked arises if two writers try to update the same row. In contrast, under the standard two-phase locking approach, both readers and writers might be blocked, since there is only one version of each data object and both read and write operations are required to obtain a lock before accessing any data.

The MVCC scheme in PostgreSQL implements the first-updater-wins version of the snapshot isolation protocol, by acquiring exclusive locks on rows that are written, but using a snapshot (without any locking) when reading rows; additional validation is done when exclusive locks are obtained, as outlined earlier in Section 18.8.

32.4.1.3 PostgreSQL Implementation of MVCC

At the core of PostgreSQL MVCC is the notion of *tuple visibility*. A PostgreSQL tuple refers to a version of a row. Tuple visibility defines which of the potentially many versions of a row in a table is valid within the context of a given statement or transaction. A transaction determines tuple visibility based on a database snapshot that is chosen before executing a command.

A tuple is visible for a transaction T if the following two conditions hold:

1. The tuple was created by a transaction that committed before transaction T took its snapshot.
2. Updates to the tuple (if any) were executed by a transaction that is either
 - aborted, *or*
 - started running after T took its snapshot, *or*
 - was active when T took its snapshot.

To be precise, a tuple is also visible to T if it was created by T and not subsequently updated by T . We omit the details of this special case for simplicity.

The goal of the above conditions is to ensure that each transaction sees a consistent view of the data. PostgreSQL maintains the following state information to check these conditions efficiently:

- A *transaction ID*, which at the same time serves as a timestamp, is assigned to every transaction at transaction start time. PostgreSQL uses a logical counter (as described in Section 18.5.1) for assigning transaction IDs.

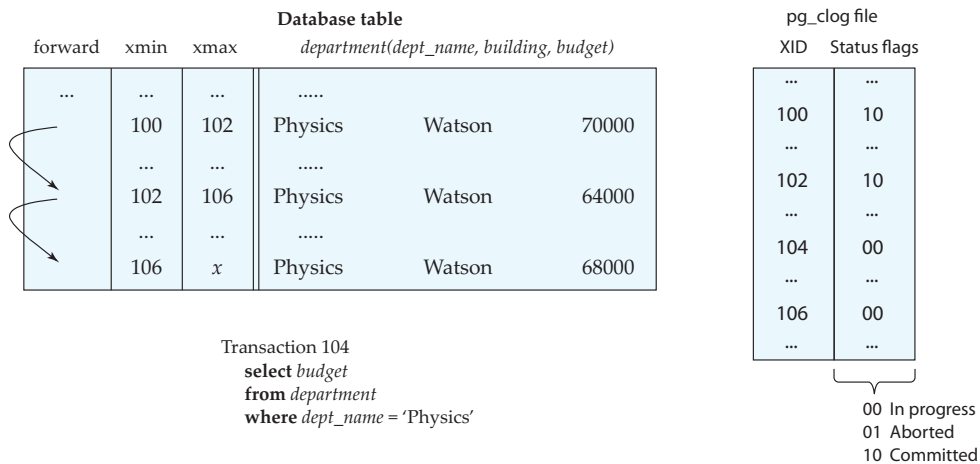


Figure 32.6 The PostgreSQL data structures used for MVCC.

- A log file called *pg_clog* contains the current status of each transaction. The status can be either in progress, committed, or aborted.
- Each tuple in a table has a header with three fields: *xmin*, which contains the transaction ID of the transaction that created the tuple and which is therefore also called the *creation-transaction ID*; *xmax*, which contains the transaction ID of the replacing/deleting transaction (or *null* if not deleted/replaced) and which is also referred to as the *expire-transaction ID*; and a forward link to new versions of the same logical row, if there are any.
- A *SnapshotData* data structure is created either at transaction start time or at query start time, depending on the isolation level (described in more detail below). Its main purpose is to decide whether a tuple is visible to the current command. The *SnapshotData* stores information about the state of transactions at the time it is created, which includes a list of active transactions and *xmax*, a value equal to 1 + the highest ID of any transaction that has started so far. The value *xmax* serves as a “cutoff” for transactions that may be considered visible.

Figure 32.6 illustrates some of this state information through a simple example involving a database with only one table, the *department* table from Figure 32.7. The *department* table has three columns, the name of the department, the building where the department is located, and the budget of the department. Figure 32.6 shows a fragment of the *department* table containing only the (versions of) the row corresponding to the Physics department. The tuple headers indicate that the row was originally created by transaction 100, and later updated by transaction 102 and transaction 106. Figure 32.6 also shows a fragment of the corresponding *pg_clog* file. On the basis of the *pg_clog*

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 32.7 The *department* relation.

file, transactions 100 and 102 are committed, while transactions 104 and 106 are in progress.

Given the above state information, the two conditions that need to be satisfied for a tuple to be visible can be rewritten as follows:

1. The creation-transaction ID in the tuple header
 - a. is a committed transaction according to the *pg_clog* file, and
 - b. is less than the cutoff transaction ID *xmax* recorded by *SnapshotData*, and
 - c. is not one of the active transactions stored in *SnapshotData*.
2. The expire-transaction ID, if it exists,
 - a. is an aborted transaction according to the *pg_clog* file, or
 - b. is greater than or equal to the cutoff transaction ID *xmax* recorded by *SnapshotData*, or
 - c. is one of the active transactions stored in *SnapshotData*.

Consider the example database in Figure 32.6 and assume that the *SnapshotData* used by transaction 104 simply uses 103 as the cutoff transaction ID *xmax* and does not show any earlier transactions to be active. In this case, the only version of the row corresponding to the Physics department that is visible to transaction 104 is the second version in the table, created by transaction 102. The first version, created by transaction 100, is not visible, since it violates condition 2: The expire-transaction ID of this tuple is 102, which corresponds to a transaction that is not aborted and that has a transaction ID less than or equal to 103. The third version of the Physics tuple is not visible, since it was created by transaction 106, which has a transaction ID larger than transaction 103, implying that this version had not been committed at the time *SnapshotData* was created. Moreover, transaction 106 is still in progress, which violates another one of the conditions. The second version of the row meets all the conditions for tuple visibility.

The details of how PostgreSQL MVCC interacts with the execution of SQL statements depends on whether the statement is an **insert**, **select**, **update**, or **delete** statement. The simplest case is an **insert** statement, which may simply create a new tuple based on the data in the statement, initialize the tuple header (the creation ID), and insert the new tuple into the table. Unlike two-phase locking, this does not require any interaction with the concurrency-control protocol unless the insertion needs to be checked for integrity conditions, such as uniqueness or foreign key constraints.

When the system executes a **select**, **update**, or **delete** statement the interaction with the MVCC protocol depends on the isolation level specified by the application. If the isolation level is read committed, the processing of a new statement begins with creating a new *SnapshotData* data structure (independent of whether the statement starts a new transaction or is part of an existing transaction). Next, the system identifies *target tuples*, that is, the tuples that are visible with respect to the *SnapshotData* and that match the search criteria of the statement. In the case of a **select** statement, the set of target tuples make up the result of the query.

In the case of an **update** or **delete** statement in read committed mode, an extra step is necessary after identifying the target tuples, before the actual update or delete operation can take place. The reason is that visibility of a tuple ensures only that the tuple has been created by a transaction that committed before the **update/delete** statement in question started. However, it is possible that, since query start, this tuple has been updated or deleted by another concurrently executing transaction. This can be detected by looking at the expire-transaction ID of the tuple. If the expire-transaction ID corresponds to a transaction that is still in progress, it is necessary to wait for the completion of this transaction first. If the transaction aborts, the **update** or **delete** statement can proceed and perform the actual modification. If the transaction commits, the search criteria of the statement need to be evaluated again, and only if the tuple still meets these criteria can the row be modified. If the row is to be deleted, the main step is to update the expire-transaction ID of the old tuple. A row update also performs this step, and additionally creates a new version of the row, sets its creation-transaction ID, and sets the forward link of the old tuple to reference the new tuple.

Going back to the example from Figure 32.6, transaction 104, which consists of a **select** statement only, identifies the second version of the Physics row as a target tuple and returns it immediately. If transaction 104 were an update statement instead, for example, trying to increment the budget of the Physics department by some amount, it would have to wait for transaction 106 to complete. It would then re-evaluate the search condition and, only if it is still met, proceed with its update.

Using the protocol described above for **update** and **delete** statements provides only the read-committed isolation level. Serializability can be violated in several ways. First, nonrepeatable reads are possible. Since each query within a transaction may see a different snapshot of the database, a query in a transaction might see the effects of an **update** command completed in the meantime that weren't visible to earlier queries within the same transaction. Following the same line of thought, phantom reads are possible when a relation is modified between queries.

In order to provide the PostgreSQL serializable isolation level, PostgreSQL MVCC eliminates violations of serializability in two ways: First, when it is determining tuple visibility, all queries within a transaction use a snapshot as of the start of the transaction, rather than the start of the individual query. This way successive queries within a transaction always see the same data.

Second, the way updates and deletes are processed is different in serializable mode compared to read-committed mode. As in read-committed mode, transactions wait after identifying a visible target row that meets the search condition and is currently updated or deleted by another concurrent transaction. If the concurrent transaction that executes the update or delete aborts, the waiting transaction can proceed with its own update. However, if the concurrent transaction commits, there is no way for PostgreSQL to ensure serializability for the waiting transaction. Therefore, the waiting transaction is rolled back and returns with the error message “could not serialize access due to concurrent update.”

It is up to the application to handle an error message like the above appropriately, by aborting the current transaction and restarting the entire transaction from the beginning. Observe that rollbacks due to serializability issues are possible only for **update** and **delete** statements. It is still the case that **select** statements never conflict with any other transactions.

32.4.1.4 Implications of Using MVCC

Using the PostgreSQL MVCC scheme has implications in three different areas: (1) extra burden is placed on the storage system, since it needs to maintain different versions of tuples; (2) developing concurrent applications takes some extra care, since PostgreSQL MVCC can lead to subtle, but important, differences in how concurrent transactions behave, compared to systems where standard two-phase locking is used; (3) PostgreSQL performance depends on the characteristics of the workload running on it. The implications of PostgreSQL MVCC are described in more detail below.

Creating and storing multiple versions of every row can lead to excessive storage consumption. To alleviate this problem, PostgreSQL frees up space when possible by identifying and deleting versions of tuples that cannot be visible to any active or future transactions, and are therefore no longer needed. The task of freeing space is nontrivial, because indices may refer to the location of an unneeded tuple, so these references need to be deleted before reusing the space. To lessen this issue, PostgreSQL avoids indexing multiple versions of a tuple that have identical index attributes. This allows the space taken by nonindexed tuples to be freed efficiently by any transaction that finds such a tuple.

For more aggressive space reuse, PostgreSQL provides the **vacuum** command, which correctly updates indices for each freed tuple. PostgreSQL employs a background process to **vacuum** tables automatically, but the command can also be executed by the user directly. The **vacuum** command offers two modes of operation: Plain **vacuum** simply identifies tuples that are not needed, and makes their space available for reuse. This

form of the command can operate in parallel with normal reading and writing of the table. **Vacuum full** does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

Because of the use of multiversion concurrency control in PostgreSQL, porting applications from other environments to PostgreSQL might require some extra care to ensure data consistency. As an example, consider a transaction T_A executing a **select** statement. Since readers in PostgreSQL don't lock data, data read and selected by T_A can be overwritten by another concurrent transaction T_B , while T_A is still running. As a result some of the data that T_A returns might not be current anymore at the time of completion of T_A . T_A might return rows that in the meantime have been changed or deleted by other transactions. To ensure the current validity of a row and protect it against concurrent updates, an application must either use **select for share** or explicitly acquire a lock with the appropriate **lock table** command.

PostgreSQL's approach to concurrency control performs best for workloads containing many more reads than updates, since in this case there is a very low chance that two updates will conflict and force a transaction to roll back. Two-phase locking may be more efficient for some update-intensive workloads, but this depends on many factors, such as the length of transactions and the frequency of deadlocks.

32.4.1.5 DDL Concurrency Control

The MVCC mechanisms described in the previous section do not protect transactions against operations that affect entire tables, for example, transactions that drop a table or change the schema of a table. Toward this end, PostgreSQL provides explicit locks that DDL commands are forced to acquire before starting their execution. These locks are always table based (rather than row based) and are acquired and released in accordance with the strict two-phase locking protocol.

Figure 32.8 lists all types of locks offered by PostgreSQL, which locks they conflict with, and some commands that use them (the **create index concurrently** command is covered in Section 32.5.2.3). The names of the lock types are often historical and don't necessarily reflect the use of the lock. For example, all the locks are table-level locks, although some contain the word "row" in the name. DML commands acquire only locks of the first three types. These three lock types are compatible with each other, since MVCC takes care of protecting these operations against each other. DML commands acquire these locks only for protection against DDL commands.

While their main purpose is providing PostgreSQL internal concurrency control for DDL commands, all locks in Figure 32.8 can also be acquired explicitly by PostgreSQL applications through the **lock table** command.

Locks are recorded in a lock table that is implemented as a shared-memory hash table keyed by a signature that identifies the object being locked. If a transaction wants to acquire a lock on an object that is held by another transaction in a conflicting mode, it needs to wait until the lock is released. Lock waits are implemented through

<i>Lock name</i>	<i>Conflicts with</i>	<i>Acquired by</i>
ACCESS SHARE	ACCESS EXCLUSIVE	select query
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	select for update query select for share query
ROW EXCLUSIVE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	update delete insert queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	vacuum analyze create index concurrently
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	create index
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	---
EXCLUSIVE	All except ACCESS SHARE	---
ACCESS EXCLUSIVE	All modes	drop table alter table vaccum full

Figure 32.8 Table-level lock modes.

semaphores, each of which is associated with a unique transaction. When waiting for a lock, a transaction actually waits on the semaphore associated with the transaction holding the lock. Once the lock holder releases the lock, it will signal the waiting transaction(s) through the semaphore. By implementing lock waits on a per-lock-holder basis, rather than on a per-object basis, PostgreSQL requires at most one semaphore per concurrent transaction, rather than one semaphore per lockable object.

Deadlock detection in PostgreSQL is based on time-outs. By default, deadlock detection is triggered if a transaction has been waiting for a lock for more than 1 second. The deadlock-detection algorithm constructs a wait-for graph based on the information in the lock table and searches this graph for circular dependencies. If it finds any, meaning a deadlock was detected, the transaction that triggered the deadlock detection aborts and returns an error to the user. If no cycle is detected, the transaction continues waiting on the lock. Unlike some commercial systems, PostgreSQL does not tune the

lock time-out parameter dynamically, but it allows the administrator to tune it manually. Ideally, this parameter should be chosen on the order of a transaction lifetime, in order to optimize the trade-off between the time it takes to detect a deadlock and the work wasted for running the deadlock detection algorithm when there is no deadlock.

32.4.1.6 Locking and Indices

All current types of indices in PostgreSQL allow for concurrent access by multiple transactions. This is typically enabled by page-level locks, so that different transactions may access the index in parallel if they do not request conflicting locks on a page. These locks are usually held for a short time to avoid deadlock, with the exception of hash indices, which lock pages for longer periods and may participate in deadlock.

32.4.2 Recovery

Historically, PostgreSQL did not use write-ahead logging (WAL) for recovery, and therefore was not able to guarantee consistency in the case of crash. A crash could potentially result in inconsistent index structures or, worse, totally corrupted table contents, because of partially written data pages. As a result, starting with version 7.1, PostgreSQL employs WAL-based recovery. The approach is similar to standard recovery techniques such as ARIES (Section 19.9), but recovery in PostgreSQL is simplified in some ways because of the MVCC protocol.

First, under PostgreSQL, recovery doesn't have to undo the effects of aborted transactions: an aborting transaction makes an entry in the *pg_clog* file, recording the fact that it is aborting. Consequently, all versions of rows it leaves behind will never be visible to any other transactions. The only case where this approach could potentially lead to problems is when a transaction aborts because of a crash of the corresponding PostgreSQL process and the PostgreSQL process doesn't have a chance to create the *pg_clog* entry before the crash. PostgreSQL handles this as follows: Before checking the status of another transaction in the *pg_clog* file, it checks whether the transaction is running on any of the PostgreSQL processes. If no PostgreSQL process is currently running the transaction and the *pg_clog* file shows the transaction as still running, it is safe to assume that the transaction crashed and the transaction's *pg_clog* entry is updated to "aborted".

Second, recovery is simplified by the fact that PostgreSQL MVCC already keeps track of some of the information required by WAL logging. More precisely, there is no need for logging the start, commit, and abort of transactions, since MVCC logs the status of every transaction in the *pg_clog*.

32.5 Storage and Indexing

PostgreSQL's approach to data layout and storage is aimed at the goals of (1) a simple and clean implementation and (2) ease of administration. As a step toward these goals,

PostgreSQL relies on “cooked” file systems, instead of handling the physical layout of data on raw disk partitions by itself. PostgreSQL maintains a list of directories in the file hierarchy to use for storage, which are conventionally referred to as **tablespaces**. Each PostgreSQL installation is initialized with a default tablespace, and additional tablespaces may be added at any time. When creating a table, index, or entire database, the user may specify any existing tablespace in which to store the related files. It is particularly useful to create multiple tablespaces if they reside on different physical devices, so that the faster devices may be dedicated to data that are in higher demand. Moreover, data that are stored on separate disks may be accessed in parallel more efficiently.

The design of the PostgreSQL storage system potentially leads to some performance limitations, due to clashes between PostgreSQL and the file system. The use of cooked file systems results in double buffering, where blocks are first fetched from disk into the file system’s cache (in kernel space) before being copied to PostgreSQL’s buffer pool. Performance can also be limited by the fact that PostgreSQL stores data in 8-KB blocks, which may not match the block size used by the kernel. It is possible to change the PostgreSQL block size when the server is installed, but this may have undesired consequences: small blocks limit the ability of PostgreSQL to store large tuples efficiently, while large blocks are wasteful when a small region of a file is accessed.

On the other hand, modern enterprises increasingly use external storage systems, such as network-attached storage and storage-area networks, instead of disks attached to servers. The philosophy here is that storage is a service that is easily administered and tuned for performance separately. One approach used by these systems is RAID, which offers both parallelism and redundant storage as explained in Section 12.5. PostgreSQL may directly leverage these technologies because of its reliance on cooked file systems. Thus, the feeling of many PostgreSQL developers is that, for a vast majority of applications, and indeed PostgreSQL’s audience, the performance limitations are minimal and justified by the ease of administration and management, as well as simplicity of implementation.

32.5.1 Tables

The primary unit of storage in PostgreSQL is a table. In PostgreSQL, tables are stored in *heap files*. These files use a form of the standard *slotted-page* format described in Section 13.2. The PostgreSQL format is shown in Figure 32.9. In each page, a header is followed by an array of “line pointers.” A line pointer holds the offset (relative to the start of the page) and length of a specific tuple in the page. The actual tuples are stored in reverse order of line pointers from the end of the page.

A record in a heap file is identified by its **tuple ID (TID)**. The TID consists of a 4-byte block ID that specifies the page in the file containing the tuple and a 2-byte slot ID. The slot ID is an index into the line pointer array that in turn is used to access the tuple.

Although this infrastructure permits tuples in a page to be deleted or updated, under PostgreSQL’s MVCC approach, neither operation actually deletes or replaces old

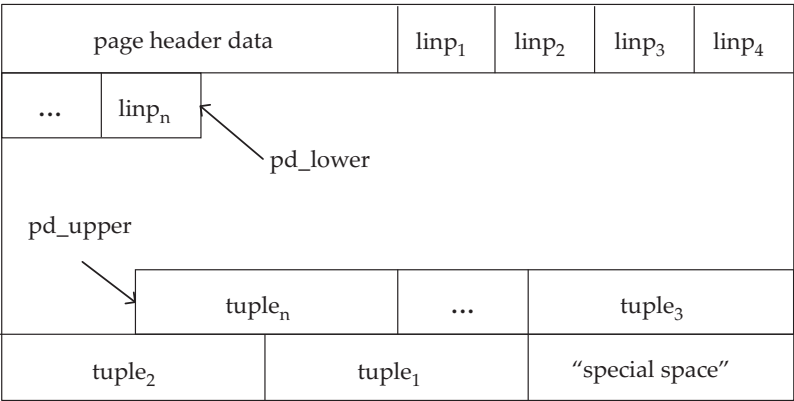


Figure 32.9 Slotted-page format for PostgreSQL tables.

versions of rows immediately. As explained in Section 32.4.1.4, expired tuples may be physically deleted by later commands, causing holes to be formed in a page. The indirection of accessing tuples through the line pointer array permits the reuse of such holes.

The length of a tuple is normally limited by the length of a data page. This makes it difficult to store very long tuples. When PostgreSQL encounters such a large tuple, it tries to “toast” individual large attributes. In some cases, toasting an attribute may be accomplished by compressing the value. If this does not shrink the tuple enough to fit in the page (often the case), the data in the toasted attribute is replaced with a reference to a copy that is stored outside the page.

32.5.2 Indices

A PostgreSQL index is a data structure that provides a dynamic mapping from search predicates to sequences of tuple IDs from a particular table. The returned tuples are intended to match the search predicate, although in some cases the predicate must be rechecked in the heap file. PostgreSQL supports several different index types, including those that are based on user-extensible access methods. Although an access method may use a different page format, all the indices available in PostgreSQL use the same slotted-page format described above in Section 32.5.1.

32.5.2.1 Index Types

PostgreSQL supports the following types of indices:

- **B-tree.** The default index type is a B⁺-tree index based on Lehman and Yao’s B-link trees (B-link trees, described in Section 18.10.2, support high concurrency of operations). These indices are useful for equality and range queries on sortable data and also for certain pattern-matching operations such as the **like** expression.

- **Hash.** PostgreSQL's hash indices are an implementation of linear hashing (for more information on hash indices, see Section 14.5). Such indices are useful only for simple equality operations. The hash indices used by PostgreSQL have been shown to have lookup performance no better than that of B-trees, but have considerably larger size and maintenance costs. Moreover, hash indices are the only indices in PostgreSQL that do not support crash recovery. Thus it is almost always preferable to use B-tree indices instead of hash indices.
- **GiST.** PostgreSQL supports a highly extensible index called GiST, or Generalized Search Tree. GiST is a balanced, tree-structured access method that makes it easy for a domain expert who is well versed in a particular data type (such as image data) to develop performance-enhancing indices without having to deal with the internal details of the database system. Examples of some indices built using GiST include B-trees and R-trees, as well as less conventional indices for multidimensional cubes and full-text search. New GiST access methods can be implemented by creating an operator class as explained in Section 32.3.3.3. Operator classes for GiST are different from B-trees, as each GiST operator class may have a different set of strategies that indicate the search predicates implemented by the index. GiST also relies on seven support functions for operations such as testing set membership and splitting sets of entries for page overflows.

It is interesting to note that the original PostgreSQL implementation of R-trees (Section 24.4.2) was replaced by GiST operator classes in version 8.2. This allowed R-trees to take advantage of the WAL logging and concurrency capabilities that were added to GiST in version 8.1. Since the original R-tree implementation did not have these features, this change illustrates the benefits of an extensible indexing method. See the bibliographical notes for references to more information on the GiST index.

- **GIN.** The newest type of index in PostgreSQL is the Generalized Inverted Index (GIN). A GIN index interprets both index keys and search keys as sets, making the index type appropriate for set-oriented operators. One of the intended uses of GIN is to index documents for full-text search, which is implemented by reducing documents and queries to sets of search terms. Like GiST, a GIN index may be extended to handle any comparison operator by creating an operator class with appropriate support functions.

To evaluate a search, GIN efficiently identifies index keys that overlap the search key, and computes a bitmap indicating which searched-for elements are members of the index key. This is accomplished using support functions that extract members from a set and compare individual members. Another support function decides if the search predicate is satisfied based on the bitmap and the original predicate. If the search predicate cannot be resolved without the full indexed attribute, the decision function must report a match and recheck the predicate in the heap file.

32.5.2.2 Other Index Variations

For some of the index types described above, PostgreSQL supports more complex variations such as:

- **Multicolumn indices.** These are useful for conjuncts of predicates over multiple columns of a table. Multicolumn indices are only supported for B-tree and GiST indices.
- **Unique indices.** Unique and primary-key constraints can be enforced by using unique indices in PostgreSQL. Only B-tree indices may be defined as being unique.
- **Indices on expressions.** In PostgreSQL, it is possible to create indices on arbitrary scalar expressions of columns, and not just specific columns, of a table. This is especially useful when the expressions in question are “expensive”—say, involving complicated user-defined computation. An example is to support case-insensitive comparisons by defining an index on the expression *lower(column)* and using the predicate *lower(column) = 'value'* in queries. One disadvantage is that the maintenance costs of indices on expressions is high.
- **Operator classes.** The specific comparison functions used to build, maintain, and use an index on a column are tied to the data type of that column. Each data type has associated with it a default “operator class” (described in Section 32.3.3.3) that identifies the actual operators that would normally be used for it. While this default operator class is normally sufficient for most uses, some data types might possess multiple “meaningful” classes. For instance, in dealing with complex numbers, it might be desirable to index either the real or imaginary component. PostgreSQL provides some built-in operator classes for pattern-matching operations (such as **like**) on text data that do not use the standard locale-specific collation rules (in other words, language specific sort orders).
- **Partial indices.** These are indices built over a subset of a table defined by a predicate. The index contains only entries for tuples that satisfy the predicate. Partial indices are suited for cases where a column might contain a large number of occurrences of a very small number of values. In such cases, the common values are not worth indexing, since index scans are not beneficial for queries that require a large subset of the base table. A partial index that excludes the common values is small and incurs less I/O. The partial indices are less expensive to maintain, as a large fraction of inserts do not participate in the index.

32.5.2.3 Index Construction

An index may be added to the database using the **create index** command. For example, the following DDL statement creates a B-tree index on instructor salaries.

```
create index inst_sal_idx on instructor (salary);
```

This statement is executed by scanning the *instructor* relation to find row versions that might be visible to a future transaction, then sorting their index attributes and building the index structure. During this process, the building transaction holds a lock on the *instructor* relation that prevents concurrent **insert**, **delete**, and **update** statements. Once the process is finished, the index is ready to use and the table lock is released.

The lock acquired by the **create index** command may present a major inconvenience for some applications where it is difficult to suspend updates while the index is built. For these cases, PostgreSQL provides the **create index concurrently** variant, which allows concurrent updates during index construction. This is achieved by a more complex construction algorithm that scans the base table twice. The first table scan builds an initial version of the index, in a way similar to normal index construction described above. This index may be missing tuples if the table was concurrently updated; however, the index is well formed, so it is flagged as being ready for insertions. Finally, the algorithm scans the table a second time and inserts all tuples it finds that still need to be indexed. This scan may also miss concurrently updated tuples, but the algorithm synchronizes with other transactions to guarantee that tuples that are updated during the second scan will be added to the index by the updating transaction. Hence, the index is ready to use after the second table scan. Since this two-pass approach can be expensive, the plain **create index** command is preferred if it is easy to suspend table updates temporarily.

32.6 Query Processing and Optimization

When PostgreSQL receives a query, it is first parsed into an internal representation, which goes through a series of transformations, resulting in a query plan that is used by the **executor** to process the query.

32.6.1 Query Rewrite

The first stage of a query's transformation is **rewrite** and it is this stage that is responsible for the PostgreSQL **rules** system. As explained in Section 32.3, in PostgreSQL, users can create **rules** that are fired on different events such as **update**, **delete**, **insert**, and **select** statements. A view is implemented by the system by converting a view definition into a **select** rule. When a query involving a **select** statement on the view is received, the **select** rule for the view is fired, and the query is rewritten using the definition of the view.

A rule is registered in the system using the **create rule** command, at which point information on the rule is stored in the catalog. This catalog is then used during query rewrite to uncover all candidate rules for a given query.

The rewrite phase first deals with all **update**, **delete**, and **insert** statements by firing all appropriate rules. Such statements might be complicated and contain **select** clauses. Subsequently, all the remaining rules involving only **select** statements are fired. Since the firing of a rule may cause the query to be rewritten to a form that may require

another rule to be fired, the rules are repeatedly checked on each form of the rewritten query until a fixed point is reached and no more rules need to be fired.

There exist no default rules in PostgreSQL—only those defined explicitly by users and implicitly by the definition of views.

32.6.2 Query Planning and Optimization

Once the query has been rewritten, it is subject to the planning and optimization phase. Here, each query block is treated in isolation and a plan is generated for it. This planning begins bottom-up from the rewritten query's innermost subquery, proceeding to its outermost query block.

The optimizer in PostgreSQL is, for the most part, cost based. The idea is to generate an access plan whose estimated cost is minimal. The cost model includes as parameters the I/O cost of sequential and random page fetches, as well as the CPU costs of processing heap tuples, index tuples, and simple predicates.

The actual process of optimization is based on one of the following two forms:

- **Standard planner.** The standard planner uses the the bottom-up dynamic programming algorithm for join order optimization, originally used in System R, the pioneering relational system developed by IBM research in the 1970s. The System R dynamic programming algorithm is described in detail in Section 16.4.1. The algorithm is used on a single query block at a time.
- **Genetic query optimizer.** When the number of tables in a query block is very large, System R's dynamic programming algorithm becomes very expensive. Unlike other commercial systems that default to greedy or rule-based techniques, PostgreSQL uses a more radical approach: a genetic algorithm that was developed initially to solve traveling-salesman problems. There exists anecdotal evidence of the successful use of genetic query optimization in production systems for queries with around 45 tables.

Since the planner operates in a bottom-up fashion on query blocks, it is able to perform certain transformations on the query plan as it is being built. One example is the common subquery-to-join transformation that is present in many commercial systems (usually implemented in the rewrite phase). When PostgreSQL encounters a noncorrelated subquery (such as one caused by a query on a view), it is generally possible to “pull up” the planned subquery and merge it into the upper-level query block. However, transformations that push duplicate elimination into lower-level query blocks are generally not possible in PostgreSQL.

The query-optimization phase results in a query plan that is a tree of relational operators. Each operator represents a specific operation on one or more sets of tuples. The operators can be unary (for example, sort, aggregation), binary (for example, nested-loop join), or n -ary (for example, set union).

Crucial to the cost model is an accurate estimate of the total number of tuples that will be processed at each operator in the plan. This is inferred by the optimizer on the basis of statistics that are maintained on each relation in the system. These indicate the total number of tuples for each relation and specific information on each column of a relation, such as the column cardinality, a list of most common values in the table and the number of occurrences, and a histogram that divides the column's values into groups of equal population (that is, an equi-depth histogram, described in Section 16.3.1). In addition, PostgreSQL also maintains a statistical correlation between the physical and logical row orderings of a column's values—this indicates the cost of an index scan to retrieve tuples that pass predicates on the column. The DBA must ensure that these statistics are current by running the **analyze** command periodically.

32.6.3 Query Executor

The executor module is responsible for processing a query plan produced by the optimizer. The executor follows the *iterator* model with a set of four functions implemented for each operator (**open**, **next**, **rescan**, and **close**). Iterators are also discussed as part of demand-driven pipelining in Section 15.7.2.1. PostgreSQL iterators have an extra function, **rescan**, which is used to reset a subplan (say for an inner loop of a join) with parameters such as index key ranges.

Some of the important operators of the executor can be categorized as follows:

1. **Access methods.** The actual access methods that are used to retrieve data from on-disk objects in PostgreSQL are sequential scans of heap files, index scans, and bitmap index scans.
 - **Sequential scans.** The tuples of a relation are scanned sequentially from the first to last blocks of the file. Each tuple is returned to the caller only if it is “visible” according to the transaction isolation rules in Section 32.4.1.3.
 - **Index scans.** Given a search condition such as a range or equality predicate, this access method returns a set of matching tuples from the associated heap file. The operator processes one tuple at a time, starting by reading an entry from the index and then fetching the corresponding tuple from the heap file. This can result in a random page fetch for each tuple in the worst case.
 - **Bitmap index scans.** A bitmap index scan reduces the danger of excessive random page fetches in index scans. This is achieved by processing tuples in two phases. The first phase reads all index entries and stores the heap tuple IDs in a bitmap, and the second phase fetches the matching heap tuples in sequential order. This guarantees that each heap page is accessed only once, and increases the chance of sequential page fetches. Moreover, bitmaps from multiple indexes can be merged and intersected to evaluate complex Boolean predicates before accessing the heap.

2. **Join methods.** PostgreSQL supports three join methods: sorted merge joins, nested-loop joins (including index-nested loop variants for the inner), and a hybrid hash join (Section 15.5).
3. **Sort.** External sorting is implemented in PostgreSQL by algorithms explained in Section 15.4. The input is divided into sorted runs that are then merged in a polyphase merge. The initial runs are formed using replacement selection, using a priority tree instead of a data structure that fixes the number of in-memory records. This is because PostgreSQL may deal with tuples that vary considerably in size and tries to ensure full utilization of the configured sort memory space.
4. **Aggregation.** Grouped aggregation in PostgreSQL can be either sort-based or hash-based. When the estimated number of distinct groups is very large the former is used and otherwise the hash-based approach is preferred.

32.6.4 Triggers and Constraints

In PostgreSQL (unlike some commercial systems) active-database features such as triggers and constraints are not implemented in the rewrite phase. Instead they are implemented as part of the query executor. When the triggers and constraints are registered by the user, the details are associated with the catalog information for each appropriate relation and index. The executor processes an **update**, **delete**, and **insert** statement by repeatedly generating tuple changes for a relation. For each row modification, the executor explicitly identifies, fires, and enforces candidate triggers and constraints, before or after the change as required.

32.7 System Architecture

The PostgreSQL system architecture follows the process-per-transaction model. A running PostgreSQL site is managed by a central coordinating process, called the **postmaster**. The postmaster process is responsible for initializing and shutting down the server and also for handling connection requests from new clients. The postmaster assigns each new connecting client to a back-end server process that is responsible for executing the queries on behalf of the client and for returning the results to the client. This architecture is depicted in Figure 32.10.

Client applications can connect to the PostgreSQL server and submit queries through one of the many database application programmer interfaces supported by PostgreSQL (libpq, JDBC, ODBC, Perl DBD) that are provided as client-side libraries. An example client application is the command-line `psql` program, included in the standard PostgreSQL distribution. The postmaster is responsible for handling the initial client connections. For this, it constantly listens for new connections on a known port. After performing initialization steps such as user authentication, the postmaster will spawn a new back-end server process to handle the new client. After this initial connection, the client interacts only with the back-end server process, submitting queries

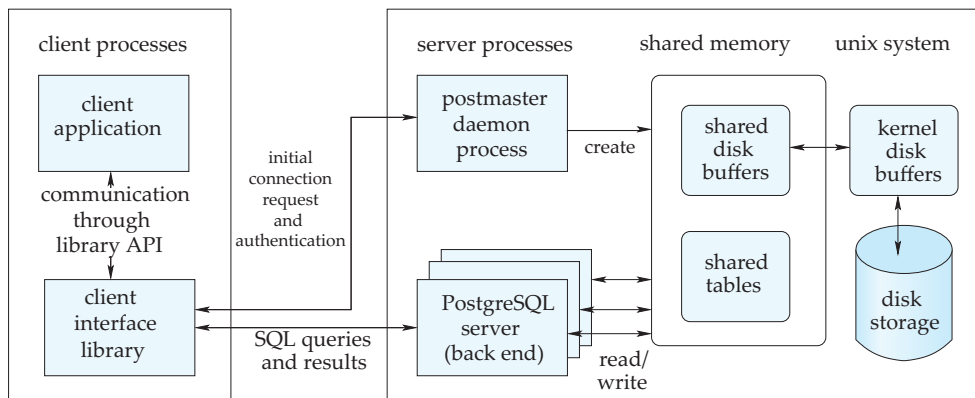


Figure 32.10 The PostgreSQL system architecture.

and receiving query results. This is the essence of the process-per-connection model adopted by PostgreSQL.

The back-end server process is responsible for executing the queries submitted by the client by performing the necessary query-execution steps, including parsing, optimization, and execution. Each back-end server process can handle only a single query at a time. In order to execute more than one query in parallel, an application must maintain multiple connections to the server.

At any given time, there may be multiple clients connected to the system and thus multiple back-end server processes may be executing concurrently. The back-end server processes access database data through the main-memory buffer pool, which is placed in shared memory, so that all the processes have the same view of the data. Shared memory is also used to implement other forms of synchronization between the server processes, for example, the locking of data items.

The use of shared memory as a communication medium suggests that a PostgreSQL server should run on a single machine; a single-server site cannot be spread across multiple machines without the assistance of third-party packages, such as the Slony replication tool. However, it is possible to build a shared-nothing parallel database system with an instance of PostgreSQL running on each node; in fact, several commercial parallel database systems have been built with exactly this architecture, as described in

Bibliographical Notes

There is extensive online documentation of PostgreSQL at www.postgresql.org. This Web site is the authoritative source for information on new releases of PostgreSQL, which occur on a frequent basis. Until PostgreSQL version 8, the only way to run PostgreSQL under Microsoft Windows was by using Cygwin. Cygwin is a Linux-like environment that allows rebuilding of Linux applications from source to run under Windows. Details are at www.cygwin.com. Books on PostgreSQL include [Douglas

and Douglas (2003)] and [Stinson (2002)]. Rules as used in PostgreSQL are presented in [Stonebraker et al. (1990)]. The GiST structure is described in [Hellerstein et al. (1995)].

Many tools and extensions for PostgreSQL are documented by the pgFoundry at www.pgfoundry.org. These include the `pgtcl` library and the pgAccess administration tool mentioned in this chapter. The pgAdmin tool is described on the Web at www.pgadmin.org. The database-design tools, TORA and Data Architect, are described at tora.sourceforge.net and www.thekompany.com/products/dataarchitect, respectively. The report-generation tools, GNU Report Generator and GNU Enterprise, are described at www.gnu.org/software/grg and www.gnuenterprise.org, respectively. The open-source Mondrian OLAP server is described at mondrian.pentaho.org.

An open-source alternative to PostgreSQL is MySQL, which is available for non-commercial use under the GNU General Public License. MySQL may be embedded in commercial software that does not have freely distributed source code, but this requires a special license to be purchased. Comparisons between the most recent versions of the two systems are readily available on the Web.

Bibliography

- [Douglas and Douglas (2003)] K. Douglas and S. Douglas, *PostgreSQL*, Sam's Publishing (2003).
- [Hellerstein et al. (1995)] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems", In *Proc. of the International Conf. on Very Large Databases* (1995), pages 562–573.
- [Stinson (2002)] B. Stinson, *PostgreSQL Essential Reference*, New Riders (2002).
- [Stonebraker et al. (1990)] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedure, Caching and Views in Database Systems", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 281–290.