PostgreSQL 17.4 Documentation

The PostgreSQL Global Development Group

PostgreSQL 17.4 Documentation

The PostgreSQL Global Development Group

Legal Notice

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

This book is the official documentation of PostgreSQL. It has been written by the PostgreSQL developers and other volunteers in parallel to the development of the PostgreSQL software. It describes all the functionality that the current version of PostgreSQL officially supports.

To make the large amount of information about PostgreSQL manageable, this book has been organized in several parts. Each part is targeted at a different class of users, or at users in different stages of their PostgreSQL experience:

- Part I is an informal introduction for new users.
- Part II documents the SQL query language environment, including data types and functions, as well as user-level performance tuning. Every PostgreSQL user should read this.
- Part III describes the installation and administration of the server. Everyone who runs a PostgreSQL server, be it for private use or for others, should read this part.
- Part IV describes the programming interfaces for PostgreSQL client programs.
- Part V contains information for advanced users about the extensibility capabilities of the server. Topics include user-defined data types and functions.
- Part VI contains reference information about SQL commands, client and server programs. This part supports the other parts with structured information sorted by command or program.
- Part VII contains assorted information that might be of use to PostgreSQL developers.

1. What Is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2[1], developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

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

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

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

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

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

2. A Brief History of PostgreSQL

[1] https://dsf.berkeley.edu/postgres.html

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

2.1. The Berkeley POSTGRES Project

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

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

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix[2], which is now owned by IBM[3]) picked up the code and commercialized it. In late 1992, POSTGRES became the primary data manager for the Sequoia 2000 scientific computing project described in [ston92].

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

2.2. Postgres95

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

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

- The query language PostQUEL was replaced with SQL (implemented in the server). (Interface library libpq was named after PostQUEL.) Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregate functions were re-implemented. Support for the `GROUP BY` query clause was also added.

- A new program (psql) was provided for interactive SQL queries, which used GNU Readline. This largely superseded the old monitor program.

- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, `pgtclsh`, provided new Tcl commands to interface Tcl programs with the Postgres95 server.

[2] https://www.ibm.com/analytics/informix
[3] https://www.ibm.com/

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

2.3. PostgreSQL

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

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

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

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

3. Conventions

The following conventions are used in the synopsis of a command: brackets (`[` and `]`) indicate optional parts. Braces (`{` and `}`) and vertical lines (`|`) indicate that you must choose one alternative. Dots (`...`) mean that the preceding element can be repeated. All other symbols, including parentheses, should be taken literally.

Where it enhances the clarity, SQL commands are preceded by the prompt `=>`, and shell commands are preceded by the prompt `$`. Normally, prompts are not shown, though.

An *administrator* is generally a person who is in charge of installing and running the server. A *user* could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should not be interpreted too narrowly; this book does not have fixed presumptions about system administration procedures.

4. Further Information

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

Wiki

The PostgreSQL wiki[4] contains the project's FAQ[5] (Frequently Asked Questions) list, TODO[6] list, and detailed information about many more topics.

Web Site

The PostgreSQL web site[7] carries details on the latest release and other information to make your work or play with PostgreSQL more productive.

[4] https://wiki.postgresql.org
[5] https://wiki.postgresql.org/wiki/Frequently_Asked_Questions
[6] https://wiki.postgresql.org/wiki/Todo
[7] https://www.postgresql.org

Mailing Lists

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

Yourself!

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

5. Bug Reporting Guidelines

When you find a bug in PostgreSQL we want to hear about it. Your bug reports play an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but doing so tends to be to everyone's advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

5.1. Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that a program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a “disk full” message, since you have to fix that yourself.)
- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.
- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here “program” refers to any executable, not only the backend process.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to the SQL standard is not necessarily a bug either, unless compliance for the specific feature is explicitly claimed.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

5.2. What to Report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what "it seemed to do", or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit. And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps *from program start-up* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare `SELECT` statement without the preceding `CREATE TABLE` and `INSERT` statements, if the output should depend on the data in the tables. We do not have the time to reverse-engineer your database schema, and if we are supposed to make up our own data we would probably miss the problem.

  The best format for a test case for SQL-related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your `~/.psqlrc` start-up file.) An easy way to create this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files; do not guess that the problem happens for "large files" or "midsize databases", etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it "didn't work" or "crashed". If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

  > **Note**
  >
  > If you are reporting an error message, please obtain the most verbose form of the message. In psql, say `\set VERBOSITY verbose` beforehand. If you are extracting the message from the server log, set the run-time parameter log_error_verbosity to `verbose` so that all details are logged.

  > **Note**
  >
  > In case of fatal errors, the error message reported by the client might not contain all the information available. Please also look at the log output of the database server. If you do not keep your server's log output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write "This command gives me that output." or "This is not what I expected.", we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking,

nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)

- Any command line options and other start-up options, including any relevant environment variables or configuration files that you changed from the default. Again, please provide exact information. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.

- Anything you did at all differently from the installation instructions.

- The PostgreSQL version. You can run the command `SELECT version();` to find out the version of the server you are connected to. Most executable programs also support a `--version` option; at least `postgres --version` and `psql --version` should work. If the function or the options do not exist then your version is more than old enough to warrant an upgrade. If you run a prepackaged version, such as RPMs, say so, including any subversion the package might have. If you are talking about a Git snapshot, mention that, including the commit hash.

  If your version is older than 17.4 we will almost certainly tell you to upgrade. There are many bug fixes and improvements in each new release, so it is quite possible that a bug you have encountered in an older release of PostgreSQL has already been fixed. We can only provide limited support for sites using older releases of PostgreSQL; if you require more than we can provide, consider acquiring a commercial support contract.

- Platform information. This includes the kernel name and version, C library, processor, memory information, and so on. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly “Debian” contains or that everyone runs on x86_64. If you have installation problems then information about the toolchain on your machine (compiler, make, and so on) is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It is better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it. Here is an article[8] that outlines some more tips on reporting bugs.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug cannot be fixed right away, you will still have time to find and share your work-around. Also, once again, do not waste your time guessing why the bug exists. We will find that out soon enough.

When writing a bug report, please avoid confusing terminology. The software package in total is called “PostgreSQL”, sometimes “Postgres” for short. If you are specifically talking about the backend process, mention that, do not just say “PostgreSQL crashes”. A crash of a single backend process is quite different from crash of the parent “postgres” process; please don't say “the server crashed” when you mean a single backend process went down, nor vice versa. Also, client programs such as the interactive frontend “psql” are completely separate from the backend. Please try to be specific about whether the problem is on the client or server side.

5.3. Where to Report Bugs

In general, send bug reports to the bug report mailing list at `<pgsql-bugs@lists.postgresql.org>`. You are requested to use a descriptive subject for your email message, perhaps parts of the error message.

Another method is to fill in the bug report web-form available at the project's web site[9]. Entering a bug report this way causes it to be mailed to the `<pgsql-bugs@lists.postgresql.org>` mailing list.

[8] https://www.chiark.greenend.org.uk/~sgtatham/bugs.html
[9] https://www.postgresql.org/

If your bug report has security implications and you'd prefer that it not become immediately visible in public archives, don't send it to `pgsql-bugs`. Security issues can be reported privately to `<security@postgresql.org>`.

Do not send bug reports to any of the user mailing lists, such as `<pgsql-sql@lists.postgresql.org>` or `<pgsql-general@lists.postgresql.org>`. These mailing lists are for answering user questions, and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do *not* send reports to the developers' mailing list `<pgsql-hackers@lists.postgresql.org>`. This list is for discussing the development of PostgreSQL, and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on `pgsql-hackers`, if the problem needs more review.

If you have a problem with the documentation, the best place to report it is the documentation mailing list `<pgsql-docs@lists.postgresql.org>`. Please be specific about what part of the documentation you are unhappy with.

If your bug is a portability problem on a non-supported platform, send mail to `<pgsql-hackers@lists.postgresql.org>`, so we (and you) can work on porting PostgreSQL to your platform.

> **Note**
>
> Due to the unfortunate amount of spam going around, all of the above lists will be moderated unless you are subscribed. That means there will be some delay before the email is delivered. If you wish to subscribe to the lists, please visit https://lists.postgresql.org/ for instructions.

Part I. Tutorial

Welcome to the PostgreSQL Tutorial. The tutorial is intended to give an introduction to PostgreSQL, relational database concepts, and the SQL language. We assume some general knowledge about how to use computers and no particular Unix or programming experience is required. This tutorial is intended to provide hands-on experience with important aspects of the PostgreSQL system. It makes no attempt to be a comprehensive treatment of the topics it covers.

After you have successfully completed this tutorial you will want to read the Part II section to gain a better understanding of the SQL language, or Part IV for information about developing applications with PostgreSQL. Those who provision and manage their own PostgreSQL installation should also read Part III.

Table of Contents

Chapter 1. Getting Started

1.1. Installation

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

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

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

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

1.2. Architectural Fundamentals

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

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

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

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

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

1.3. Creating a Database

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

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

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

```
$ createdb mydb
```

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

If you see a message similar to:

```
createdb: command not found
```

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

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

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

Another response could be this:

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

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

Another response could be this:

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

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL user account for you. (PostgreSQL user accounts are distinct from operating system user accounts.) If you are the administrator, see Chapter 21 for help creating accounts. You will need to become the operating system user under which PostgreSQL was installed (usually `postgres`) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your operating system user name; in that case you need to use the `-U` switch or set the `PGUSER` environment variable to specify your PostgreSQL user name.

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

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

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

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

```
$ createdb
```

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

```
$ dropdb mydb
```

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

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

1.4. Accessing a Database

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

- Running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC or JDBC support to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in Part IV.

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

```
$ psql mydb
```

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

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

```
psql (17.4)
Type "help" for help.

mydb=>
```

The last line could also be:

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

```
mydb=#
```

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

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

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

```
mydb=> SELECT version();
                                             version
-------------------------------------------------------------------------
----------------------
 PostgreSQL 17.4 on x86_64-pc-linux-gnu, compiled by gcc (Debian
 4.9.2-10) 4.9.2, 64-bit
(1 row)

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

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

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

```
mydb=> \h
```

To get out of `psql`, type:

```
mydb=> \q
```

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

Chapter 2. The SQL Language

2.1. Introduction

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

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

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

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

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

```
$ psql -s mydb

...

mydb=> \i basics.sql
```

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

2.2. Concepts

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

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

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

2.3. Creating a New Table

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

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

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

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

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

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

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

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

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

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

```
DROP TABLE tablename;
```

2.4. Populating a Table With Rows

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5. Querying a Table

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

```
SELECT * FROM weather;
```

Here `*` is a shorthand for “all columns”. [1] So the same result would be had with:

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

The output should be:

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

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

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

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

This should give:

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

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

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

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

Result:

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

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

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

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

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

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

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

```
SELECT DISTINCT city
    FROM weather;
```

```
     city
---------------
```

```
 Hayward
 San Francisco
(2 rows)
```

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together: [2]

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

2.6. Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. Queries that access multiple tables (or multiple instances of the same table) at one time are called *join* queries. They combine rows from one table with rows from a second table, with an expression specifying which rows are to be paired. For example, to return all the weather records together with the location of the associated city, the database needs to compare the `city` column of each row of the `weather` table with the `name` column of all rows in the `cities` table, and select the pairs of rows where these values match.[3] This would be accomplished by the following query:

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

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

Observe two things about the result set:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
SELECT w1.city, w1.temp_lo AS low, w1.temp_hi AS high,
       w2.city, w2.temp_lo AS low, w2.temp_hi AS high
    FROM weather w1 JOIN weather w2
        ON w1.temp_lo < w2.temp_lo AND w1.temp_hi > w2.temp_hi;
```

```
     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

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

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

You will encounter this style of abbreviating quite frequently.

2.7. Aggregate Functions

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

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

```
SELECT max(temp_lo) FROM weather;
```

```
 max
-----
  46
(1 row)
```

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

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

but this will not work since the aggregate `max` cannot be used in the `WHERE` clause. (This restriction exists because the `WHERE` clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a *subquery*:

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

```
     city
---------------
 San Francisco
(1 row)
```

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

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

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

```
     city      | count | max
---------------+-------+-----
 Hayward       |     1 |  37
 San Francisco |     2 |  46
(2 rows)
```

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

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

```
  city   | count | max
---------+-------+-----
 Hayward |     1 |  37
(1 row)
```

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

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

```
     city      | count | max
---------------+-------+-----
 San Francisco |     2 |  46
(1 row)
```

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

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

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

Another way to select the rows that go into an aggregate computation is to use `FILTER`, which is a per-aggregate option:

```
SELECT city, count(*) FILTER (WHERE temp_lo < 45), max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | count | max
---------------+-------+-----
 Hayward       |     1 |  37
 San Francisco |     1 |  46
(2 rows)
```

`FILTER` is much like `WHERE`, except that it removes rows only from the input of the particular aggregate function that it is attached to. Here, the `count` aggregate counts only rows with `temp_lo` below 45; but the `max` aggregate is still applied to all rows, so it still finds the reading of 46.

2.8. Updates

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

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

Look at the new state of the data:

```
SELECT * FROM weather;
```

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

2.9. Deletions

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

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

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;
```

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

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

One should be wary of statements of the form

```
DELETE FROM tablename;
```

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

Chapter 3. Advanced Features

3.1. Introduction

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

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

3.2. Views

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

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

SELECT * FROM myview;
```

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

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

3.3. Foreign Keys

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

The new declaration of the tables would look like this:

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

CREATE TABLE weather (
        city      varchar(80) references cities(name),
        temp_lo   int,
```

```
        temp_hi   int,
        prcp      real,
        date      date
);
```

Now try inserting an invalid record:

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

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

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

3.4. Transactions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
```

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

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

3.5. Window Functions

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. However, window functions do not cause rows to become grouped into a single output row like non-window aggregate calls would. Instead, the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

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

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY
 depname) FROM empsalary;
```

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

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

A window function call always contains an `OVER` clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a normal function or non-window aggregate. The `OVER` clause determines exactly how the rows of the query are split up for processing by the window function. The `PARTITION BY` clause within `OVER` divides the rows into groups, or partitions, that share the same values of the `PARTITION BY` expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

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

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

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

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

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

There is another important concept associated with window functions: for each row, there is a set of rows within its partition called its *window frame*. Some window functions act only on the rows of the window frame, rather than of the whole partition. By default, if `ORDER BY` is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the `ORDER BY` clause. When `ORDER BY` is omitted the default frame consists of all rows in the partition. [1] Here is an example using `sum`:

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

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

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

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

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

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically execute after the processing of those clauses. Also, window functions execute after non-window aggregate functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

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

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

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

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

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

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

3.6. Inheritance

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

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

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

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

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

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

A better solution is this:

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

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

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

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

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

which returns:

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

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

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

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

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

> **Note**
>
> Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness. See Section 5.11 for more detail.

3.7. Conclusion

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

If you feel you need more introductory material, please visit the PostgreSQL web site[2] for links to more resources.

[2] https://www.postgresql.org

Part II. The SQL Language

This part describes the use of the SQL language in PostgreSQL. We start with describing the general syntax of SQL, then how to create tables, how to populate the database, and how to query it. The middle part lists the available data types and functions for use in SQL commands. Lastly, we address several aspects of importance for tuning a database.

The information is arranged so that a novice user can follow it from start to end and gain a full understanding of the topics without having to refer forward too many times. The chapters are intended to be self-contained, so that advanced users can read the chapters individually as they choose. The information is presented in narrative form with topical units. Readers looking for a complete description of a particular command are encouraged to review the Part VI.

Readers should know how to connect to a PostgreSQL database and issue SQL commands. Readers that are unfamiliar with these issues are encouraged to read Part I first. SQL commands are typically entered using the PostgreSQL interactive terminal psql, but other programs that have similar functionality can be used as well.

Table of Contents

Chapter 4. SQL Syntax

This chapter describes the syntax of SQL. It forms the foundation for understanding the following chapters which will go into detail about how SQL commands are applied to define and modify data.

We also advise users who are already familiar with SQL to read this chapter carefully because it contains several rules and concepts that are implemented inconsistently among SQL databases or that are specific to PostgreSQL.

4.1. Lexical Structure

SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (";"). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

Additionally, *comments* can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a "SELECT", an "UPDATE", and an "INSERT" command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` in order to be complete. The precise syntax rules for each command are described in Part VI.

4.1.1. Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens `MY_TABLE` and `A` are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix C.

SQL identifiers and key words must begin with a letter (`a-z`, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or key word can be letters, underscores, digits (`0-9`), or dollar signs ($). Note that dollar signs are not allowed in identifiers according to the letter of the SQL standard, so their use might render applications less portable. The SQL standard will not define a key word that contains digits or starts or ends with an underscore, so identifiers of this form are safe against possible conflict with future extensions of the standard.

The system uses no more than NAMEDATALEN-1 bytes of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 64 so the maximum identifier length is 63 bytes. If this limit is problematic, it can be raised by changing the NAMEDATALEN constant in src/include/pg_config_manual.h.

Key words and unquoted identifiers are case-insensitive. Therefore:

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.:

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So "select" could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by PostgreSQL, but "Foo" and "FOO" are different from these three and each other. (The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, foo should be equivalent to "FOO" not "foo" according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.)

A variant of quoted identifiers allows including escaped Unicode characters identified by their code points. This variant starts with U& (upper or lower case U followed by ampersand) immediately before the opening double quote, without any spaces in between, for example U&"foo". (Note that this creates an ambiguity with the operator &. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the identifier "data" could be written as

```
U&"d\0061t\+000061"
```

The following less trivial example writes the Russian word "slon" (elephant) in Cyrillic letters:

```
U&"\0441\043B\043E\043D"
```

If a different escape character than backslash is desired, it can be specified using the UESCAPE clause after the string, for example:

```
U&"d!0061t!+000061" UESCAPE '!'
```

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character. Note that the escape character is written in single quotes, not double quotes, after UESCAPE.

To include the escape character in the identifier literally, write it twice.

Either the 4-digit or the 6-digit escape form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (Surrogate pairs are not stored directly, but are combined into a single code point.)

If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that's not possible.

4.1.2. Constants

There are three kinds of *implicitly-typed constants* in PostgreSQL: strings, bit strings, and numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

4.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes ('), for example 'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g., 'Dianne''s horse'. Note that this is *not* the same as a double-quote character (").

Two string constants that are only separated by whitespace *with at least one newline* are concatenated and effectively treated as if the string had been written as one constant. For example:

```
SELECT 'foo'
'bar';
```

is equivalent to:

```
SELECT 'foobar';
```

but:

```
SELECT 'foo'      'bar';
```

is not valid syntax. (This slightly bizarre behavior is specified by SQL; PostgreSQL is following the standard.)

4.1.2.2. String Constants with C-Style Escapes

PostgreSQL also accepts “escape” string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter E (upper or lower case) just before the opening single quote, e.g., E'foo'. (When continuing an escape string constant across lines, write E only before the first opening quote.) Within an escape string, a backslash character (\) begins a C-like *backslash escape* sequence, in which the combination of backslash and following character(s) represent a special byte value, as shown in Table 4.1.

Table 4.1. Backslash Escape Sequences

Backslash Escape Sequence	Interpretation
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\`*o*, `\`*oo*, `\`*ooo* (*o* = 0–7)	octal byte value
`\x`*h*, `\x`*hh* (*h* = 0–9, A–F)	hexadecimal byte value
`\u`*xxxx*, `\U`*xxxxxxxx* (*x* = 0–9, A–F)	16 or 32-bit hexadecimal Unicode character value

Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (`\\`). Also, a single quote can be included in an escape string by writing `\'`, in addition to the normal way of `''`.

It is your responsibility that the byte sequences you create, especially when using the octal or hexadecimal escapes, compose valid characters in the server character set encoding. A useful alternative is to use Unicode escapes or the alternative Unicode escape syntax, explained in Section 4.1.2.3; then the server will check that the character conversion is possible.

> **Caution**
>
> If the configuration parameter standard_conforming_strings is `off`, then PostgreSQL recognizes backslash escapes in both regular and escape string constants. However, as of PostgreSQL 9.1, the default is `on`, meaning that backslash escapes are recognized only in escape string constants. This behavior is more standards-compliant, but might break applications which rely on the historical behavior, where backslash escapes were always recognized. As a workaround, you can set this parameter to `off`, but it is better to migrate away from using backslash escapes. If you need to use a backslash escape to represent a special character, write the string constant with an `E`.
>
> In addition to `standard_conforming_strings`, the configuration parameters escape_string_warning and backslash_quote govern treatment of backslashes in string constants.

The character with the code zero cannot be in a string constant.

4.1.2.3. String Constants with Unicode Escapes

PostgreSQL also supports another type of escape syntax for strings that allows specifying arbitrary Unicode characters by code point. A Unicode escape string constant starts with `U&` (upper or lower case letter U followed by ampersand) immediately before the opening quote, without any spaces in between, for example `U&'foo'`. (Note that this creates an ambiguity with the operator `&`. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the string `'data'` could be written as

```
U&'d\0061t\+000061'
```

The following less trivial example writes the Russian word “slon” (elephant) in Cyrillic letters:

```
U&'\0441\043B\043E\043D'
```

If a different escape character than backslash is desired, it can be specified using the UESCAPE clause after the string, for example:

```
U&'d!0061t!+000061' UESCAPE '!'
```

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character.

To include the escape character in the string literally, write it twice.

Either the 4-digit or the 6-digit escape form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (Surrogate pairs are not stored directly, but are combined into a single code point.)

If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that's not possible.

Also, the Unicode escape syntax for string constants only works when the configuration parameter standard_conforming_strings is turned on. This is because otherwise this syntax could confuse clients that parse the SQL statements to the point that it could lead to SQL injections and similar security issues. If the parameter is set to off, this syntax will be rejected with an error message.

4.1.2.4. Dollar-Quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes, since each of those must be doubled. To allow more readable queries in such situations, PostgreSQL provides another way, called “dollar quoting”, to write string constants. A dollar-quoted string constant consists of a dollar sign ($), an optional “tag” of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. For example, here are two different ways to specify the string “Dianne's horse” using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Notice that inside the dollar-quoted string, single quotes can be used without needing to be escaped. Indeed, no characters inside a dollar-quoted string are ever escaped: the string content is always written literally. Backslashes are not special, and neither are dollar signs, unless they are part of a sequence matching the opening tag.

It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level. This is most commonly used in writing function definitions. For example:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Here, the sequence `$q$[\t\r\n\v\\]$q$` represents a dollar-quoted literal string `[\t\r\n\v\\]`, which will be recognized when the function body is executed by PostgreSQL. But since the sequence does not match the outer dollar quoting delimiter `$function$`, it is just some more characters within the constant so far as the outer string is concerned.

The tag, if any, of a dollar-quoted string follows the same rules as an unquoted identifier, except that it cannot contain a dollar sign. Tags are case sensitive, so `$tag$String content$tag$` is correct, but `$TAG$String content$tag$` is not.

A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax. It is particularly useful when representing string constants inside other constants, as is often needed in procedural function definitions. With single-quote syntax, each backslash in the above example would have to be written as four backslashes, which would be reduced to two backslashes in parsing the original string constant, and then to one when the inner string constant is re-parsed during function execution.

4.1.2.5. Bit-String Constants

Bit-string constants look like regular string constants with a `B` (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`. The only characters allowed within bit-string constants are `0` and `1`.

Alternatively, bit-string constants can be specified in hexadecimal notation, using a leading `X` (upper or lower case), e.g., `X'1FF'`. This notation is equivalent to a bit-string constant with four binary digits for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string constants. Dollar quoting cannot be used in a bit-string constant.

4.1.2.6. Numeric Constants

Numeric constants are accepted in these general forms:

digits
digits.[*digits*][e[+-]*digits*]
[*digits*].*digits*[e[+-]*digits*]
*digits*e[+-]*digits*

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (e), if one is present. There cannot be any spaces or other characters embedded in the constant, except for underscores, which can be used for visual grouping as described below. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42
3.5
4.
.001
5e2
1.925e-3

Additionally, non-decimal integer constants are accepted in these forms:

0x*hexdigits*
0o*octdigits*
0b*bindigits*

where *hexdigits* is one or more hexadecimal digits (0-9, A-F), *octdigits* is one or more octal digits (0-7), and *bindigits* is one or more binary digits (0 or 1). Hexadecimal digits and the radix prefixes can be in upper or lower case. Note that only integers can have non-decimal forms, not numbers with fractional parts.

These are some examples of valid non-decimal integer constants:

```
0b100101
0B10011001
0o273
0O755
0x42f
0XFFFF
```

For visual grouping, underscores can be inserted between digits. These have no further effect on the value of the constant. For example:

```
1_500_000_000
0b10001000_00000000
0o_1_755
0xFFFF_FFFF
1.618_034
```

Underscores are not allowed at the start or end of a numeric constant or a group of digits (that is, immediately before or after the decimal point or the exponent marker), and more than one underscore in a row is not allowed.

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type `integer` if its value fits in type `integer` (32 bits); otherwise it is presumed to be type `bigint` if its value fits in type `bigint` (64 bits); otherwise it is taken to be type `numeric`. Constants that contain decimal points and/or exponents are always initially presumed to be type `numeric`.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it. For example, you can force a numeric value to be treated as type `real` (`float4`) by writing:

```
REAL '1.23'  -- string style
1.23::REAL   -- PostgreSQL (historical) style
```

These are actually just special cases of the general casting notations discussed next.

4.1.2.7. Constants of Other Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

The string constant's text is passed to the input conversion routine for the type called *type*. The result is a constant of the indicated type. The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

The string constant can be written using either regular SQL notation or dollar-quoting.

It is also possible to specify a type coercion using a function-like syntax:

```
typename ( 'string' )
```

but not all type names can be used in this way; see Section 4.2.9 for details.

The `::`, `CAST()`, and function-call syntaxes can also be used to specify run-time type conversions of arbitrary expressions, as discussed in Section 4.2.9. To avoid syntactic ambiguity, the *`type 'string'`* syntax can only be used to specify the type of a simple literal constant. Another restriction on the *`type 'string'`* syntax is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

The `CAST()` syntax conforms to SQL. The *`type 'string'`* syntax is a generalization of the standard: SQL specifies this syntax only for a few data types, but PostgreSQL allows it for all types. The syntax with `::` is historical PostgreSQL usage, as is the function-call syntax.

4.1.3. Operators

An operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list:

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

There are a few restrictions on operator names, however:

- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in `+` or `-`, unless the name also contains at least one of these characters:

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a prefix operator named `@`, you cannot write `X*@Y`; you must write `X* @Y` to ensure that PostgreSQL reads it as two operator names not one.

4.1.4. Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign (`$`) followed by digits is used to represent a positional parameter in the body of a function definition or a prepared statement. In other contexts the dollar sign can be part of an identifier or a dollar-quoted string constant.
- Parentheses (`()`) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets (`[]`) are used to select the elements of an array. See Section 8.15 for more information on arrays.

- Commas (`,`) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (`;`) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (`:`) is used to select "slices" from arrays. (See Section 8.15.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (`*`) is used in some contexts to denote all the fields of a table row or composite value. It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
- The period (`.`) is used in numeric constants, and to separate schema, table, and column names.

4.1.5. Comments

A comment is a sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with `/*` and extends to the matching occurrence of `*/`. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that might contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

4.1.6. Operator Precedence

Table 4.2 shows the precedence and associativity of the operators in PostgreSQL. Most operators have the same precedence and are left-associative. The precedence and associativity of the operators is hard-wired into the parser. Add parentheses if you want an expression with multiple operators to be parsed in some other way than what the precedence rules imply.

Table 4.2. Operator Precedence (highest to lowest)

Operator/Element	Associativity	Description
`.`	left	table/column name separator
`::`	left	PostgreSQL-style typecast
`[ ]`	left	array element selection
`+ -`	right	unary plus, unary minus
`COLLATE`	left	collation selection
`AT`	left	`AT TIME ZONE`, `AT LOCAL`
`^`	left	exponentiation
`* / %`	left	multiplication, division, modulo
`+ -`	left	addition, subtraction

Operator/Element	Associativity	Description
(any other operator)	left	all other native and user-defined operators
BETWEEN IN LIKE ILIKE SIMILAR		range containment, set membership, string matching
< > = <= >= <>		comparison operators
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc.
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a "+" operator for some custom data type it will have the same precedence as the built-in "+" operator, no matter what yours does.

When a schema-qualified operator name is used in the OPERATOR syntax, as for example in:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

the OPERATOR construct is taken to have the default precedence shown in Table 4.2 for "any other operator". This is true no matter which specific operator appears inside OPERATOR().

Note

PostgreSQL versions before 9.5 used slightly different operator precedence rules. In particular, <= >= and <> used to be treated as generic operators; IS tests used to have higher priority; and NOT BETWEEN and related constructs acted inconsistently, being taken in some cases as having the precedence of NOT rather than BETWEEN. These rules were changed for better compliance with the SQL standard and to reduce confusion from inconsistent treatment of logically equivalent constructs. In most cases, these changes will result in no behavioral change, or perhaps in "no such operator" failures which can be resolved by adding parentheses. However there are corner cases in which a query might change behavior without any parsing error being reported.

4.2. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command, as new column values in INSERT or UPDATE, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called *scalar expressions* (or even simply *expressions*). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value
- A column reference
- A positional parameter reference, in the body of a function definition or prepared statement
- A subscripted expression

- A field selection expression
- An operator invocation
- A function call
- An aggregate expression
- A window function call
- A type cast
- A collation expression
- A scalar subquery
- An array constructor
- A row constructor
- Another value expression in parentheses (used to group subexpressions and override precedence)

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 9. An example is the `IS NULL` clause.

We have already discussed constants in Section 4.1.2. The following sections discuss the remaining options.

4.2.1. Column References

A column can be referenced in the form:

correlation.*columnname*

correlation is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a `FROM` clause. The correlation name and separating dot can be omitted if the column name is unique across all the tables being used in the current query. (See also Chapter 7.)

4.2.2. Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

$*number*

For example, consider the definition of a function, `dept`, as:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here the `$1` references the value of the first function argument whenever the function is invoked.

4.2.3. Subscripts

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing

```
expression[subscript]
```

or multiple adjacent elements (an “array slice”) can be extracted by writing

```
expression[lower_subscript:upper_subscript]
```

(Here, the brackets `[ ]` are meant to appear literally.) Each *`subscript`* is itself an expression, which will be rounded to the nearest integer value.

In general the array *`expression`* must be parenthesized, but the parentheses can be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example:

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

The parentheses in the last example are required. See Section 8.15 for more about arrays.

4.2.4. Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing

```
expression.fieldname
```

In general the row *`expression`* must be parenthesized, but the parentheses can be omitted when the expression to be selected from is just a table reference or positional parameter. For example:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(Thus, a qualified column reference is actually just a special case of the field selection syntax.) An important special case is extracting a field from a table column that is of a composite type:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

The parentheses are required here to show that `compositecol` is a column name not a table name, or that `mytable` is a table name not a schema name in the second case.

You can ask for all fields of a composite value by writing `.*`:

```
(compositecol).*
```

This notation behaves differently depending on context; see Section 8.16.5 for details.

4.2.5. Operator Invocations

There are two possible syntaxes for an operator invocation:

expression operator expression (binary infix operator)
operator expression (unary prefix operator)

where the *operator* token follows the syntax rules of Section 4.1.3, or is one of the key words AND, OR, and NOT, or is a qualified operator name in the form:

```
OPERATOR(schema.operatorname)
```

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. Chapter 9 describes the built-in operators.

4.2.6. Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function_name ([expression [, expression ... ]] )
```

For example, the following computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is in Chapter 9. Other functions can be added by the user.

When issuing queries in a database where some users mistrust other users, observe security precautions from Section 10.3 when writing function calls.

The arguments can optionally have names attached. See Section 4.3 for details.

> **Note**
>
> A function that takes a single argument of composite type can optionally be called using field-selection syntax, and conversely field selection can be written in functional style. That is, the notations `col(table)` and `table.col` are interchangeable. This behavior is not SQL-standard but is provided in PostgreSQL because it allows use of functions to emulate "computed fields". For more information see Section 8.16.5.

4.2.7. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER
 ( WHERE filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] )
 [ FILTER ( WHERE filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] )
 [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP
 ( order_by_clause ) [ FILTER ( WHERE filter_clause ) ]
```

where *aggregate_name* is a previously defined aggregate (possibly qualified with a schema name) and *expression* is any value expression that does not itself contain an aggregate expression or a window function call. The optional *order_by_clause* and *filter_clause* are described below.

The first form of aggregate expression invokes the aggregate once for each input row. The second form is the same as the first, since ALL is the default. The third form invokes the aggregate once for each distinct value of the expression (or distinct set of values, for multiple expressions) found in the input rows. The fourth form invokes the aggregate once for each input row; since no particular input value is specified, it is generally only useful for the count(*) aggregate function. The last form is used with *ordered-set* aggregate functions, which are described below.

Most aggregate functions ignore null inputs, so that rows in which one or more of the expression(s) yield null are discarded. This can be assumed to be true, unless otherwise specified, for all built-in aggregates.

For example, count(*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-null, since count ignores nulls; and count(distinct f1) yields the number of distinct non-null values of f1.

Ordinarily, the input rows are fed to the aggregate function in an unspecified order. In many cases this does not matter; for example, min produces the same result no matter what order it receives the inputs in. However, some aggregate functions (such as array_agg and string_agg) produce results that depend on the ordering of the input rows. When using such an aggregate, the optional *order_by_clause* can be used to specify the desired ordering. The *order_by_clause* has the same syntax as for a query-level ORDER BY clause, as described in Section 7.5, except that its expressions are always just expressions and cannot be output-column names or numbers. For example:

```
WITH vals (v) AS ( VALUES (1),(3),(4),(3),(2) )
SELECT array_agg(v ORDER BY v DESC) FROM vals;
  array_agg
-------------
 {4,3,3,2,1}
```

Since jsonb only keeps the last matching key, ordering of its keys can be significant:

```
WITH vals (k, v) AS ( VALUES ('key0','1'), ('key1','3'),
 ('key1','2') )
SELECT jsonb_object_agg(k, v ORDER BY v) FROM vals;
      jsonb_object_agg
---------------------------
 {"key0": "1", "key1": "3"}
```

When dealing with multiple-argument aggregate functions, note that the ORDER BY clause goes after all the aggregate arguments. For example, write this:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

not this:

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

The latter is syntactically valid, but it represents a call of a single-argument aggregate function with two ORDER BY keys (the second one being rather useless since it's a constant).

If DISTINCT is specified with an *order_by_clause*, ORDER BY expressions can only reference columns in the DISTINCT list. For example:

```
WITH vals (v) AS ( VALUES (1),(3),(4),(3),(2) )
SELECT array_agg(DISTINCT v ORDER BY v DESC) FROM vals;
 array_agg
-----------
 {4,3,2,1}
```

Placing `ORDER BY` within the aggregate's regular argument list, as described so far, is used when ordering the input rows for general-purpose and statistical aggregates, for which ordering is optional. There is a subclass of aggregate functions called *ordered-set aggregates* for which an *`order_by_clause`* is *required*, usually because the aggregate's computation is only sensible in terms of a specific ordering of its input rows. Typical examples of ordered-set aggregates include rank and percentile calculations. For an ordered-set aggregate, the *`order_by_clause`* is written inside `WITHIN GROUP (...)`, as shown in the final syntax alternative above. The expressions in the *`order_by_clause`* are evaluated once per input row just like regular aggregate arguments, sorted as per the *`order_by_clause`*'s requirements, and fed to the aggregate function as input arguments. (This is unlike the case for a non-`WITHIN GROUP` *`order_by_clause`*, which is not treated as argument(s) to the aggregate function.) The argument expressions preceding `WITHIN GROUP`, if any, are called *direct arguments* to distinguish them from the *aggregated arguments* listed in the *`order_by_clause`*. Unlike regular aggregate arguments, direct arguments are evaluated only once per aggregate call, not once per input row. This means that they can contain variables only if those variables are grouped by `GROUP BY`; this restriction is the same as if the direct arguments were not inside an aggregate expression at all. Direct arguments are typically used for things like percentile fractions, which only make sense as a single value per aggregation calculation. The direct argument list can be empty; in this case, write just `()` not `(*)`. (PostgreSQL will actually accept either spelling, but only the first way conforms to the SQL standard.)

An example of an ordered-set aggregate call is:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM
 households;
 percentile_cont
-----------------
           50489
```

which obtains the 50th percentile, or median, value of the `income` column from table `households`. Here, `0.5` is a direct argument; it would make no sense for the percentile fraction to be a value varying across rows.

If `FILTER` is specified, then only the input rows for which the *`filter_clause`* evaluates to true are fed to the aggregate function; other rows are discarded. For example:

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

The predefined aggregate functions are described in Section 9.21. Other aggregate functions can be added by the user.

An aggregate expression can only appear in the result list or `HAVING` clause of a `SELECT` command. It is forbidden in other clauses, such as `WHERE`, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see Section 4.2.11 and Section 9.24), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments (and *filter_clause* if any) contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or HAVING clause applies with respect to the query level that the aggregate belongs to.

4.2.8. Window Function Calls

A *window function call* represents the application of an aggregate-like function over some portion of the rows selected by a query. Unlike non-window aggregate calls, this is not tied to grouping of the selected rows into a single output row — each row remains separate in the query output. However the window function has access to all the rows that would be part of the current row's group according to the grouping specification (PARTITION BY list) of the window function call. The syntax of a window function call is one of the following:

```
function_name ([expression [, expression ... ]]) [ FILTER
 ( WHERE filter_clause ) ] OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER
 ( WHERE filter_clause ) ] OVER ( window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ]
 OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER
 ( window_definition )
```

where *window_definition* has the syntax

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS
 { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

The optional *frame_clause* can be one of

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end
 [ frame_exclusion ]
```

where *frame_start* and *frame_end* can be one of

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

and *frame_exclusion* can be one of

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

Here, *expression* represents any value expression that does not itself contain window function calls.

window_name is a reference to a named window specification defined in the query's WINDOW clause. Alternatively, a full *window_definition* can be given within parentheses, using the same syntax as for defining a named window in the WINDOW clause; see the SELECT reference page for details. It's worth pointing out that OVER wname is not exactly equivalent to OVER (wname ...); the latter implies copying and modifying the window definition, and will be rejected if the referenced window specification includes a frame clause.

The PARTITION BY clause groups the rows of the query into *partitions*, which are processed separately by the window function. PARTITION BY works similarly to a query-level GROUP BY clause, except that its expressions are always just expressions and cannot be output-column names or numbers. Without PARTITION BY, all rows produced by the query are treated as a single partition. The ORDER BY clause determines the order in which the rows of a partition are processed by the window function. It works similarly to a query-level ORDER BY clause, but likewise cannot use output-column names or numbers. Without ORDER BY, rows are processed in an unspecified order.

The *frame_clause* specifies the set of rows constituting the *window frame*, which is a subset of the current partition, for those window functions that act on the frame instead of the whole partition. The set of rows in the frame can vary depending on which row is the current row. The frame can be specified in RANGE, ROWS or GROUPS mode; in each case, it runs from the *frame_start* to the *frame_end*. If *frame_end* is omitted, the end defaults to CURRENT ROW.

A *frame_start* of UNBOUNDED PRECEDING means that the frame starts with the first row of the partition, and similarly a *frame_end* of UNBOUNDED FOLLOWING means that the frame ends with the last row of the partition.

In RANGE or GROUPS mode, a *frame_start* of CURRENT ROW means the frame starts with the current row's first *peer* row (a row that the window's ORDER BY clause sorts as equivalent to the current row), while a *frame_end* of CURRENT ROW means the frame ends with the current row's last peer row. In ROWS mode, CURRENT ROW simply means the current row.

In the *offset* PRECEDING and *offset* FOLLOWING frame options, the *offset* must be an expression not containing any variables, aggregate functions, or window functions. The meaning of the *offset* depends on the frame mode:

- In ROWS mode, the *offset* must yield a non-null, non-negative integer, and the option means that the frame starts or ends the specified number of rows before or after the current row.
- In GROUPS mode, the *offset* again must yield a non-null, non-negative integer, and the option means that the frame starts or ends the specified number of *peer groups* before or after the current row's peer group, where a peer group is a set of rows that are equivalent in the ORDER BY ordering. (There must be an ORDER BY clause in the window definition to use GROUPS mode.)
- In RANGE mode, these options require that the ORDER BY clause specify exactly one column. The *offset* specifies the maximum difference between the value of that column in the current row and its value in preceding or following rows of the frame. The data type of the *offset* expression varies depending on the data type of the ordering column. For numeric ordering columns it is typically of the same type as the ordering column, but for datetime ordering columns it is an interval. For example, if the ordering column is of type date or timestamp, one could write RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING. The *offset* is still required to be non-null and non-negative, though the meaning of “non-negative” depends on its data type.

In any case, the distance to the end of the frame is limited by the distance to the end of the partition, so that for rows near the partition ends the frame might contain fewer rows than elsewhere.

Notice that in both ROWS and GROUPS mode, 0 PRECEDING and 0 FOLLOWING are equivalent to CURRENT ROW. This normally holds in RANGE mode as well, for an appropriate data-type-specific meaning of “zero”.

The *frame_exclusion* option allows rows around the current row to be excluded from the frame, even if they would be included according to the frame start and frame end options. EXCLUDE CURRENT ROW excludes the current row from the frame. EXCLUDE GROUP excludes the current row and its ordering peers from the frame. EXCLUDE TIES excludes any peers of the current row from the frame, but not the current row itself. EXCLUDE NO OTHERS simply specifies explicitly the default behavior of not excluding the current row or its peers.

The default framing option is RANGE UNBOUNDED PRECEDING, which is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. With ORDER BY, this sets the frame to be all rows from the partition start up through the current row's last ORDER BY peer. Without ORDER BY, this means all rows of the partition are included in the window frame, since all rows become peers of the current row.

Restrictions are that *frame_start* cannot be UNBOUNDED FOLLOWING, *frame_end* cannot be UNBOUNDED PRECEDING, and the *frame_end* choice cannot appear earlier in the above list of *frame_start* and *frame_end* options than the *frame_start* choice does — for example RANGE BETWEEN CURRENT ROW AND *offset* PRECEDING is not allowed. But, for example, ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING is allowed, even though it would never select any rows.

If FILTER is specified, then only the input rows for which the *filter_clause* evaluates to true are fed to the window function; other rows are discarded. Only window functions that are aggregates accept a FILTER clause.

The built-in window functions are described in Table 9.65. Other window functions can be added by the user. Also, any built-in or user-defined general-purpose or statistical aggregate can be used as a window function. (Ordered-set and hypothetical-set aggregates cannot presently be used as window functions.)

The syntaxes using * are used for calling parameter-less aggregate functions as window functions, for example count(*) OVER (PARTITION BY x ORDER BY y). The asterisk (*) is customarily not used for window-specific functions. Window-specific functions do not allow DISTINCT or ORDER BY to be used within the function argument list.

Window function calls are permitted only in the SELECT list and the ORDER BY clause of the query.

More information about window functions can be found in Section 3.5, Section 9.22, and Section 7.2.5.

4.2.9. Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

```
CAST ( expression AS type )
expression::type
```

The CAST syntax conforms to SQL; the syntax with :: is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. Notice that this is subtly different from the use of casts with constants, as shown in Section 4.1.2.7. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast can usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked “OK

to apply implicitly" in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

It is also possible to specify a type cast using a function-like syntax:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, `double precision` cannot be used this way, but the equivalent `float8` can. Also, the names `interval`, `time`, and `timestamp` can only be used in this fashion if they are double-quoted, because of syntactic conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided.

> **Note**
>
> The function-like syntax is in fact just a function call. When one of the two standard cast syntaxes is used to do a run-time conversion, it will internally invoke a registered function to perform the conversion. By convention, these conversion functions have the same name as their output type, and thus the "function-like syntax" is nothing more than a direct invocation of the underlying conversion function. Obviously, this is not something that a portable application should rely on. For further details see CREATE CAST.

4.2.10. Collation Expressions

The `COLLATE` clause overrides the collation of an expression. It is appended to the expression it applies to:

```
expr COLLATE collation
```

where *`collation`* is a possibly schema-qualified identifier. The `COLLATE` clause binds tighter than operators; parentheses can be used when necessary.

If no collation is explicitly specified, the database system either derives a collation from the columns involved in the expression, or it defaults to the default collation of the database if no column is involved in the expression.

The two common uses of the `COLLATE` clause are overriding the sort order in an `ORDER BY` clause, for example:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

and overriding the collation of a function or operator call that has locale-sensitive results, for example:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

Note that in the latter case the `COLLATE` clause is attached to an input argument of the operator we wish to affect. It doesn't matter which argument of the operator or function call the `COLLATE` clause is attached to, because the collation that is applied by the operator or function is derived by considering all arguments, and an explicit `COLLATE` clause will override the collations of all other arguments. (Attaching non-matching `COLLATE` clauses to more than one argument, however, is an error. For more details see Section 23.2.) Thus, this gives the same result as the previous example:

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

But this is an error:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

because it attempts to apply a collation to the result of the > operator, which is of the non-collatable data type `boolean`.

4.2.11. Scalar Subqueries

A scalar subquery is an ordinary `SELECT` query in parentheses that returns exactly one row with one column. (See Chapter 7 for information about writing queries.) The `SELECT` query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be null.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also Section 9.24 for other expressions involving subqueries.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state =
 states.name)
    FROM states;
```

4.2.12. Array Constructors

An array constructor is an expression that builds an array value using values for its member elements. A simple array constructor consists of the key word `ARRAY`, a left square bracket `[`, a list of expressions (separated by commas) for the array element values, and finally a right square bracket `]`. For example:

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

By default, the array element type is the common type of the member expressions, determined using the same rules as for `UNION` or `CASE` constructs (see Section 10.5). You can override this by explicitly casting the array constructor to the desired type, for example:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

This has the same effect as casting each expression to the array element type individually. For more on casting, see Section 4.2.9.

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word `ARRAY` can be omitted. For example, these produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
```

```
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions. Any cast applied to the outer ARRAY constructor propagates automatically to all the inner constructors.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

You can construct an empty array, but since it's impossible to have an array with no type, you must explicitly cast your empty array to the desired type. For example:

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word ARRAY followed by a parenthesized (not bracketed) subquery. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                               array
-----------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS
 a(i));
             array
---------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

The subquery must return a single column. If the subquery's output column is of a non-array type, the resulting one-dimensional array will have an element for each row in the subquery result, with an

element type matching that of the subquery's output column. If the subquery's output column is of an array type, the result will be an array of the same type but one higher dimension; in this case all the subquery rows must yield arrays of identical dimensionality, else the result would not be rectangular.

The subscripts of an array value built with ARRAY always begin with one. For more information about arrays, see Section 8.15.

4.2.13. Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) using values for its member fields. A row constructor consists of the key word ROW, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example:

```
SELECT ROW(1,2.5,'this is a test');
```

The key word ROW is optional when there is more than one expression in the list.

A row constructor can include the syntax *rowvalue*.*, which will be expanded to a list of the elements of the row value, just as occurs when the .* syntax is used at the top level of a SELECT list (see Section 8.16.5). For example, if table t has columns f1 and f2, these are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Note**
>
> Before PostgreSQL 8.2, the .* syntax was not expanded in row constructors, so that writing ROW(t.*, 42) created a two-field row whose first field was another row value. The new behavior is usually more useful. If you need the old behavior of nested row values, write the inner row value without .*, for instance ROW(t, 42).

By default, the value created by a ROW expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with CREATE TYPE AS. An explicit cast might be needed to avoid ambiguity. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1'
 LANGUAGE SQL;

-- No cast needed since only one getf1() exists
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1'
 LANGUAGE SQL;

-- Now we need a cast to indicate which function to call:
```

```
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter. Also, it is possible to test rows using the standard comparison operators as described in Section 9.2, to compare one row against another as described in Section 9.25, and to use them in connection with subqueries, as discussed in Section 9.24,

4.2.14. Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

```
SELECT true OR somefunc();
```

then `somefunc()` would (probably) not be called at all. The same would be the case if one wrote:

```
SELECT somefunc() OR true;
```

Note that this is not the same as the left-to-right “short-circuiting” of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in `WHERE` and `HAVING` clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (`AND`/`OR`/`NOT` combinations) in those clauses can be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a `CASE` construct (see Section 9.18) can be used. For example, this is an untrustworthy way of trying to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

But this is safe:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

A `CASE` construct used in this fashion will defeat optimization attempts, so it should only be done when necessary. (In this particular example, it would be better to sidestep the problem by writing `y > 1.5*x` instead.)

CASE is not a cure-all for such issues, however. One limitation of the technique illustrated above is that it does not prevent early evaluation of constant subexpressions. As described in Section 36.7, functions and operators marked IMMUTABLE can be evaluated when the query is planned rather than when it is executed. Thus for example

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

is likely to result in a division-by-zero failure due to the planner trying to simplify the constant subexpression, even if every row in the table has x > 0 so that the ELSE arm would never be entered at run time.

While that particular example might seem silly, related cases that don't obviously involve constants can occur in queries executed within functions, since the values of function arguments and local variables can be inserted into queries as constants for planning purposes. Within PL/pgSQL functions, for example, using an IF-THEN-ELSE statement to protect a risky computation is much safer than just nesting it in a CASE expression.

Another limitation of the same kind is that a CASE cannot prevent evaluation of an aggregate expression contained within it, because aggregate expressions are computed before other expressions in a SELECT list or HAVING clause are considered. For example, the following query can cause a division-by-zero error despite seemingly having protected against it:

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

The min() and avg() aggregates are computed concurrently over all the input rows, so if any row has employees equal to zero, the division-by-zero error will occur before there is any opportunity to test the result of min(). Instead, use a WHERE or FILTER clause to prevent problematic input rows from reaching an aggregate function in the first place.

4.3. Calling Functions

PostgreSQL allows functions that have named parameters to be called using either *positional* or *named* notation. Named notation is especially useful for functions that have a large number of parameters, since it makes the associations between parameters and actual arguments more explicit and reliable. In positional notation, a function call is written with its argument values in the same order as they are defined in the function declaration. In named notation, the arguments are matched to the function parameters by name and can be written in any order. For each notation, also consider the effect of function argument types, documented in Section 10.3.

In either notation, parameters that have default values given in the function declaration need not be written in the call at all. But this is particularly useful in named notation, since any combination of parameters can be omitted; while in positional notation parameters can only be omitted from right to left.

PostgreSQL also supports *mixed* notation, which combines positional and named notation. In this case, positional parameters are written first and named parameters appear after them.

The following examples will illustrate the usage of all three notations, using the following function definition:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase
 boolean DEFAULT false)
RETURNS text
```

```
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Function `concat_lower_or_upper` has two mandatory parameters, `a` and `b`. Additionally there is one optional parameter `uppercase` which defaults to `false`. The `a` and `b` inputs will be concatenated, and forced to either upper or lower case depending on the `uppercase` parameter. The remaining details of this function definition are not important here (see Chapter 36 for more information).

4.3.1. Using Positional Notation

Positional notation is the traditional mechanism for passing arguments to functions in PostgreSQL. An example is:

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

All arguments are specified in order. The result is upper case since `uppercase` is specified as `true`. Another example is:

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Here, the `uppercase` parameter is omitted, so it receives its default value of `false`, resulting in lower case output. In positional notation, arguments can be omitted from right to left so long as they have defaults.

4.3.2. Using Named Notation

In named notation, each argument's name is specified using `=>` to separate it from the argument expression. For example:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Again, the argument `uppercase` was omitted so it is set to `false` implicitly. One advantage of using named notation is that the arguments may be specified in any order, for example:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase
 => true);
 concat_lower_or_upper
```

```
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b =>
 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

An older syntax based on ":=" is supported for backward compatibility:

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b :=
 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. Using Mixed Notation

The mixed notation combines positional and named notation. However, as already mentioned, named arguments cannot precede positional arguments. For example:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

In the above query, the arguments `a` and `b` are specified positionally, while `uppercase` is specified by name. In this example, that adds little except documentation. With a more complex function having numerous parameters that have default values, named or mixed notation can save a great deal of writing and reduce chances for error.

> **Note**
>
> Named and mixed call notations currently cannot be used when calling an aggregate function (but they do work when an aggregate function is used as a window function).

Chapter 5. Data Definition

This chapter covers how one creates the database structures that will hold one's data. In a relational database, the raw data is stored in tables, so the majority of this chapter is devoted to explaining how tables are created and modified and what features are available to control what data is stored in the tables. Subsequently, we discuss how tables can be organized into schemas, and how privileges can be assigned to tables. Finally, we will briefly look at other features that affect the data storage, such as inheritance, table partitioning, views, functions, and triggers.

5.1. Table Basics

A table in a relational database is much like a table on paper: It consists of rows and columns. The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. SQL does not make any guarantees about the order of the rows in a table. When a table is read, the rows will appear in an unspecified order, unless sorting is explicitly requested. This is covered in Chapter 7. Furthermore, SQL does not assign unique identifiers to rows, so it is possible to have several completely identical rows in a table. This is a consequence of the mathematical model that underlies SQL but is usually not desirable. Later in this chapter we will see how to deal with this issue.

Each column has a data type. The data type constrains the set of possible values that can be assigned to a column and assigns semantics to the data stored in the column so that it can be used for computations. For instance, a column declared to be of a numerical type will not accept arbitrary text strings, and the data stored in such a column can be used for mathematical computations. By contrast, a column declared to be of a character string type will accept almost any kind of data but it does not lend itself to mathematical calculations, although other operations such as string concatenation are available.

PostgreSQL includes a sizable set of built-in data types that fit many applications. Users can also define their own data types. Most built-in data types have obvious names and semantics, so we defer a detailed explanation to Chapter 8. Some of the frequently used data types are `integer` for whole numbers, `numeric` for possibly fractional numbers, `text` for character strings, `date` for dates, `time` for time-of-day values, and `timestamp` for values containing both date and time.

To create a table, you use the aptly named CREATE TABLE command. In this command you specify at least a name for the new table, the names of the columns and the data type of each column. For example:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

This creates a table named `my_first_table` with two columns. The first column is named `first_column` and has a data type of `text`; the second column has the name `second_column` and the type `integer`. The table and column names follow the identifier syntax explained in Section 4.1.1. The type names are usually also identifiers, but there are some exceptions. Note that the column list is comma-separated and surrounded by parentheses.

Of course, the previous example was heavily contrived. Normally, you would give names to your tables and columns that convey what kind of data they store. So let's look at a more realistic example:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
```

```
);
```

(The `numeric` type can store fractional components, as would be typical of monetary amounts.)

> **Tip**
>
> When you create many interrelated tables it is wise to choose a consistent naming pattern for the tables and columns. For instance, there is a choice of using singular or plural nouns for table names, both of which are favored by some theorist or other.

There is a limit on how many columns a table can contain. Depending on the column types, it is between 250 and 1600. However, defining a table with anywhere near this many columns is highly unusual and often a questionable design.

If you no longer need a table, you can remove it using the DROP TABLE command. For example:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Attempting to drop a table that does not exist is an error. Nevertheless, it is common in SQL script files to unconditionally try to drop each table before creating it, ignoring any error messages, so that the script works whether or not the table exists. (If you like, you can use the `DROP TABLE IF EXISTS` variant to avoid the error messages, but this is not standard SQL.)

If you need to modify a table that already exists, see Section 5.7 later in this chapter.

With the tools discussed so far you can create fully functional tables. The remainder of this chapter is concerned with adding features to the table definition to ensure data integrity, security, or convenience. If you are eager to fill your tables with data now you can skip ahead to Chapter 6 and read the rest of this chapter later.

5.2. Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is. (Details about data manipulation commands are in Chapter 6.)

If no default value is declared explicitly, the default value is the null value. This usually makes sense because a null value can be considered to represent unknown data.

In a table definition, default values are listed after the column data type. For example:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

The default value can be an expression, which will be evaluated whenever the default value is inserted (*not* when the table is created). A common example is for a `timestamp` column to have a default of `CURRENT_TIMESTAMP`, so that it gets set to the time of row insertion. Another common example is generating a “serial number” for each row. In PostgreSQL this is typically done by something like:

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

where the `nextval()` function supplies successive values from a *sequence object* (see Section 9.17). This arrangement is sufficiently common that there's a special shorthand for it:

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

The `SERIAL` shorthand is discussed further in Section 8.1.4.

5.3. Identity Columns

An identity column is a special column that is generated automatically from an implicit sequence. It can be used to generate key values.

To create an identity column, use the `GENERATED ... AS IDENTITY` clause in `CREATE TABLE`, for example:

```
CREATE TABLE people (
    id bigint GENERATED ALWAYS AS IDENTITY,
    ...,
);
```

or alternatively

```
CREATE TABLE people (
    id bigint GENERATED BY DEFAULT AS IDENTITY,
    ...,
);
```

See CREATE TABLE for more details.

If an `INSERT` command is executed on the table with the identity column and no value is explicitly specified for the identity column, then a value generated by the implicit sequence is inserted. For example, with the above definitions and assuming additional appropriate columns, writing

```
INSERT INTO people (name, address) VALUES ('A', 'foo');
INSERT INTO people (name, address) VALUES ('B', 'bar');
```

would generate values for the `id` column starting at 1 and result in the following table data:

```
 id | name | address
----+------+---------
  1 | A    | foo
  2 | B    | bar
```

Alternatively, the keyword `DEFAULT` can be specified in place of a value to explicitly request the sequence-generated value, like

```
INSERT INTO people (id, name, address) VALUES (DEFAULT, 'C',
 'baz');
```

Similarly, the keyword DEFAULT can be used in UPDATE commands.

Thus, in many ways, an identity column behaves like a column with a default value.

The clauses ALWAYS and BY DEFAULT in the column definition determine how explicitly user-specified values are handled in INSERT and UPDATE commands. In an INSERT command, if ALWAYS is selected, a user-specified value is only accepted if the INSERT statement specifies OVERRIDING SYSTEM VALUE. If BY DEFAULT is selected, then the user-specified value takes precedence. Thus, using BY DEFAULT results in a behavior more similar to default values, where the default value can be overridden by an explicit value, whereas ALWAYS provides some more protection against accidentally inserting an explicit value.

The data type of an identity column must be one of the data types supported by sequences. (See CREATE SEQUENCE.) The properties of the associated sequence may be specified when creating an identity column (see CREATE TABLE) or changed afterwards (see ALTER TABLE).

An identity column is automatically marked as NOT NULL. An identity column, however, does not guarantee uniqueness. (A sequence normally returns unique values, but a sequence could be reset, or values could be inserted manually into the identity column, as discussed above.) Uniqueness would need to be enforced using a PRIMARY KEY or UNIQUE constraint.

In table inheritance hierarchies, identity columns and their properties in a child table are independent of those in its parent tables. A child table does not inherit identity columns or their properties automatically from the parent. During INSERT or UPDATE, a column is treated as an identity column if that column is an identity column in the table named in the statement, and the corresponding identity properties are applied.

Partitions inherit identity columns from the partitioned table. They cannot have their own identity columns. The properties of a given identity column are consistent across all the partitions in the partition hierarchy.

5.4. Generated Columns

A generated column is a special column that is always computed from other columns. Thus, it is for columns what a view is for tables. There are two kinds of generated columns: stored and virtual. A stored generated column is computed when it is written (inserted or updated) and occupies storage as if it were a normal column. A virtual generated column occupies no storage and is computed when it is read. Thus, a virtual generated column is similar to a view and a stored generated column is similar to a materialized view (except that it is always updated automatically). PostgreSQL currently implements only stored generated columns.

To create a generated column, use the GENERATED ALWAYS AS clause in CREATE TABLE, for example:

```
CREATE TABLE people (
    ...,
    height_cm numeric,
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

The keyword STORED must be specified to choose the stored kind of generated column. See CREATE TABLE for more details.

A generated column cannot be written to directly. In INSERT or UPDATE commands, a value cannot be specified for a generated column, but the keyword DEFAULT may be specified.

Consider the differences between a column with a default and a generated column. The column default is evaluated once when the row is first inserted if no other value was provided; a generated column is updated whenever the row changes and cannot be overridden. A column default may not refer to other columns of the table; a generation expression would normally do so. A column default can use volatile functions, for example `random()` or functions referring to the current time; this is not allowed for generated columns.

Several restrictions apply to the definition of generated columns and tables involving generated columns:

- The generation expression can only use immutable functions and cannot use subqueries or reference anything other than the current row in any way.
- A generation expression cannot reference another generated column.
- A generation expression cannot reference a system column, except `tableoid`.
- A generated column cannot have a column default or an identity definition.
- A generated column cannot be part of a partition key.
- Foreign tables can have generated columns. See CREATE FOREIGN TABLE for details.
- For inheritance and partitioning:
  - If a parent column is a generated column, its child column must also be a generated column; however, the child column can have a different generation expression. The generation expression that is actually applied during insert or update of a row is the one associated with the table that the row is physically in. (This is unlike the behavior for column defaults: for those, the default value associated with the table named in the query applies.)
  - If a parent column is not a generated column, its child column must not be generated either.
  - For inherited tables, if you write a child column definition without any `GENERATED` clause in `CREATE TABLE ... INHERITS`, then its `GENERATED` clause will automatically be copied from the parent. `ALTER TABLE ... INHERIT` will insist that parent and child columns already match as to generation status, but it will not require their generation expressions to match.
  - Similarly for partitioned tables, if you write a child column definition without any `GENERATED` clause in `CREATE TABLE ... PARTITION OF`, then its `GENERATED` clause will automatically be copied from the parent. `ALTER TABLE ... ATTACH PARTITION` will insist that parent and child columns already match as to generation status, but it will not require their generation expressions to match.
  - In case of multiple inheritance, if one parent column is a generated column, then all parent columns must be generated columns. If they do not all have the same generation expression, then the desired expression for the child must be specified explicitly.

Additional considerations apply to the use of generated columns.

- Generated columns maintain access privileges separately from their underlying base columns. So, it is possible to arrange it so that a particular role can read from a generated column but not from the underlying base columns.
- Generated columns are, conceptually, updated after `BEFORE` triggers have run. Therefore, changes made to base columns in a `BEFORE` trigger will be reflected in generated columns. But conversely, it is not allowed to access generated columns in `BEFORE` triggers.
- Generated columns are skipped for logical replication and cannot be specified in a `CREATE PUBLICATION` column list.

5.5. Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

5.5.1. Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word `CHECK` followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

You can also give the constraint a separate name. This clarifies error messages and allows you to refer to the constraint when you need to change it. The syntax is:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

So, to specify a named constraint, use the key word `CONSTRAINT` followed by an identifier followed by the constraint definition. (If you don't specify a constraint name in this way, the system chooses a name for you.)

A check constraint can also refer to several columns. Say you store a regular price and a discounted price, and you want to ensure that the discounted price is lower than the regular price:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

The first two constraints should look familiar. The third one uses a new syntax. It is not attached to a particular column, instead it appears as a separate item in the comma-separated column list. Column definitions and these constraint definitions can be listed in mixed order.

We say that the first two constraints are column constraints, whereas the third one is a table constraint because it is written separately from any one column definition. Column constraints can also be written as table constraints, while the reverse is not necessarily possible, since a column constraint is supposed to refer to only the column it is attached to. (PostgreSQL doesn't enforce that rule, but you should follow it if you want your table definitions to work with other database systems.) The above example could also be written as:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

or even:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

It's a matter of taste.

Names can be assigned to table constraints in the same way as column constraints:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

It should be noted that a check constraint is satisfied if the check expression evaluates to true or the null value. Since most expressions will evaluate to the null value if any operand is null, they will not prevent null values in the constrained columns. To ensure that a column does not contain null values, the not-null constraint described in the next section can be used.

> **Note**
>
> PostgreSQL does not support `CHECK` constraints that reference table data other than the new or updated row being checked. While a `CHECK` constraint that violates this rule may appear to work in simple tests, it cannot guarantee that the database will not reach a state in which

the constraint condition is false (due to subsequent changes of the other row(s) involved). This would cause a database dump and restore to fail. The restore could fail even when the complete database state is consistent with the constraint, due to rows not being loaded in an order that will satisfy the constraint. If possible, use `UNIQUE`, `EXCLUDE`, or `FOREIGN KEY` constraints to express cross-row and cross-table restrictions.

If what you desire is a one-time check against other rows at row insertion, rather than a continuously-maintained consistency guarantee, a custom trigger can be used to implement that. (This approach avoids the dump/restore problem because pg_dump does not reinstall triggers until after restoring data, so that the check will not be enforced during a dump/restore.)

Note

PostgreSQL assumes that `CHECK` constraints' conditions are immutable, that is, they will always give the same result for the same input row. This assumption is what justifies examining `CHECK` constraints only when rows are inserted or updated, and not at other times. (The warning above about not referencing other table data is really a special case of this restriction.)

An example of a common way to break this assumption is to reference a user-defined function in a `CHECK` expression, and then change the behavior of that function. PostgreSQL does not disallow that, but it will not notice if there are rows in the table that now violate the `CHECK` constraint. That would cause a subsequent database dump and restore to fail. The recommended way to handle such a change is to drop the constraint (using `ALTER TABLE`), adjust the function definition, and re-add the constraint, thereby rechecking it against all table rows.

5.5.2. Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint `CHECK (`*`column_name`* `IS NOT NULL)`, but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created this way.

Of course, a column can have more than one constraint. Just write the constraints one after another:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

The order doesn't matter. It does not necessarily determine in which order the constraints are checked.

The `NOT NULL` constraint has an inverse: the `NULL` constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply selects the default behavior that the column might be null. The `NULL` constraint is not present in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with some other database

systems.) Some users, however, like it because it makes it easy to toggle the constraint in a script file. For example, you could start with:

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

and then insert the `NOT` key word where desired.

> **Tip**
>
> In most database designs the majority of columns should be marked not null.

5.5.3. Unique Constraints

Unique constraints ensure that the data contained in a column, or a group of columns, is unique among all the rows in the table. The syntax is:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

when written as a column constraint, and:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

when written as a table constraint.

To define a unique constraint for a group of columns, write it as a table constraint with the column names separated by commas:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

This specifies that the combination of values in the indicated columns is unique across the whole table, though any one of the columns need not be (and ordinarily isn't) unique.

You can assign your own name for a unique constraint, in the usual way:

```
CREATE TABLE products (
```

```
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

Adding a unique constraint will automatically create a unique B-tree index on the column or group of columns listed in the constraint. A uniqueness restriction covering only some rows cannot be written as a unique constraint, but it is possible to enforce such a restriction by creating a unique partial index.

In general, a unique constraint is violated if there is more than one row in the table where the values of all of the columns included in the constraint are equal. By default, two null values are not considered equal in this comparison. That means even in the presence of a unique constraint it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. This behavior can be changed by adding the clause `NULLS NOT DISTINCT`, like

```
CREATE TABLE products (
    product_no integer UNIQUE NULLS NOT DISTINCT,
    name text,
    price numeric
);
```

or

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE NULLS NOT DISTINCT (product_no)
);
```

The default behavior can be specified explicitly using `NULLS DISTINCT`. The default null treatment in unique constraints is implementation-defined according to the SQL standard, and other implementations have a different behavior. So be careful when developing applications that are intended to be portable.

5.5.4. Primary Keys

A primary key constraint indicates that a column, or group of columns, can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null. So, the following two table definitions accept the same data:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Primary keys can span more than one column; the syntax is similar to unique constraints:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

Adding a primary key will automatically create a unique B-tree index on the column or group of columns listed in the primary key, and will force the column(s) to be marked `NOT NULL`.

A table can have at most one primary key. (There can be any number of unique and not-null constraints, which are functionally almost the same thing, but only one can be identified as the primary key.) Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, but it is usually best to follow it.

Primary keys are useful both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely. There are also various ways in which the database system makes use of a primary key if one has been declared; for example, the primary key defines the default target column(s) for foreign keys referencing its table.

5.5.5. Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the *referential integrity* between two related tables.

Say you have the product table that we have used several times already:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

Now it is impossible to create orders with non-NULL `product_no` entries that do not appear in the products table.

We say that in this situation the orders table is the *referencing* table and the products table is the *referenced* table. Similarly, there are referencing and referenced columns.

You can also shorten the above command to:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
```

```
    product_no integer REFERENCES products,
    quantity integer
);
```

because in absence of a column list the primary key of the referenced table is used as the referenced column(s).

You can assign your own name for a foreign key constraint, in the usual way.

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form. Here is a contrived syntax example:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Of course, the number and type of the constrained columns need to match the number and type of the referenced columns.

Sometimes it is useful for the “other table” of a foreign key constraint to be the same table; this is called a *self-referential* foreign key. For example, if you want rows of a table to represent nodes of a tree structure, you could write

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text,
    ...
);
```

A top-level node would have NULL `parent_id`, while non-NULL `parent_id` entries would be constrained to reference valid rows of the table.

A table can have more than one foreign key constraint. This is used to implement many-to-many relationships between tables. Say you have tables about products and orders, but now you want to allow one order to contain possibly many products (which the structure above did not allow). You could use this table structure:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
```

```
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Notice that the primary key overlaps with the foreign keys in the last table.

We know that the foreign keys disallow creation of orders that do not relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to handle that as well. Intuitively, we have a few options:

- Disallow deleting a referenced product
- Delete the orders as well
- Something else?

To illustrate this, let's implement the following policy on the many-to-many relationship example above: when someone wants to remove a product that is still referenced by an order (via `order_items`), we disallow it. If someone removes an order, the order items are removed as well:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Restricting and cascading deletes are the two most common options. `RESTRICT` prevents deletion of a referenced row. `NO ACTION` means that if any referencing rows still exist when the constraint is checked, an error is raised; this is the default behavior if you do not specify anything. (The essential difference between these two choices is that `NO ACTION` allows the check to be deferred until later in the transaction, whereas `RESTRICT` does not.) `CASCADE` specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well. There are two other options: `SET NULL` and `SET DEFAULT`. These cause the referencing column(s) in the referencing row(s) to be set to nulls or their default values, respectively, when the referenced row is deleted. Note that these do not excuse you from observing any constraints. For example, if an action specifies `SET DEFAULT` but the default value would not satisfy the foreign key constraint, the operation will fail.

The appropriate choice of `ON DELETE` action depends on what kinds of objects the related tables represent. When the referencing table represents something that is a component of what is represented by the referenced table and cannot exist independently, then `CASCADE` could be appropriate. If the two tables represent independent objects, then `RESTRICT` or `NO ACTION` is more appropriate; an application that actually wants to delete both objects would then have to be explicit about this and run two delete commands. In the above example, order items are part of an order, and it is convenient if they are deleted automatically if an order is deleted. But products and orders are different things, and so making a deletion of a product automatically cause the deletion of some order items could be considered problematic. The actions `SET NULL` or `SET DEFAULT` can be appropriate if a foreign-key

relationship represents optional information. For example, if the products table contained a reference to a product manager, and the product manager entry gets deleted, then setting the product's product manager to null or a default might be useful.

The actions SET NULL and SET DEFAULT can take a column list to specify which columns to set. Normally, all columns of the foreign-key constraint are set; setting only a subset is useful in some special cases. Consider the following example:

```
CREATE TABLE tenants (
    tenant_id integer PRIMARY KEY
);

CREATE TABLE users (
    tenant_id integer REFERENCES tenants ON DELETE CASCADE,
    user_id integer NOT NULL,
    PRIMARY KEY (tenant_id, user_id)
);

CREATE TABLE posts (
    tenant_id integer REFERENCES tenants ON DELETE CASCADE,
    post_id integer NOT NULL,
    author_id integer,
    PRIMARY KEY (tenant_id, post_id),
    FOREIGN KEY (tenant_id, author_id) REFERENCES users ON DELETE
 SET NULL (author_id)
);
```

Without the specification of the column, the foreign key would also set the column tenant_id to null, but that column is still required as part of the primary key.

Analogous to ON DELETE there is also ON UPDATE which is invoked when a referenced column is changed (updated). The possible actions are the same, except that column lists cannot be specified for SET NULL and SET DEFAULT. In this case, CASCADE means that the updated values of the referenced column(s) should be copied into the referencing row(s).

Normally, a referencing row need not satisfy the foreign key constraint if any of its referencing columns are null. If MATCH FULL is added to the foreign key declaration, a referencing row escapes satisfying the constraint only if all its referencing columns are null (so a mix of null and non-null values is guaranteed to fail a MATCH FULL constraint). If you don't want referencing rows to be able to avoid satisfying the foreign key constraint, declare the referencing column(s) as NOT NULL.

A foreign key must reference columns that either are a primary key or form a unique constraint, or are columns from a non-partial unique index. This means that the referenced columns always have an index to allow efficient lookups on whether a referencing row has a match. Since a DELETE of a row from the referenced table or an UPDATE of a referenced column will require a scan of the referencing table for rows matching the old value, it is often a good idea to index the referencing columns too. Because this is not always needed, and there are many choices available on how to index, the declaration of a foreign key constraint does not automatically create an index on the referencing columns.

More information about updating and deleting data is in Chapter 6. Also see the description of foreign key constraint syntax in the reference documentation for CREATE TABLE.

5.5.6. Exclusion Constraints

Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null. The syntax is:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

See also `CREATE TABLE ... CONSTRAINT ... EXCLUDE` for details.

Adding an exclusion constraint will automatically create an index of the type specified in the constraint declaration.

5.6. System Columns

Every table has several *system columns* that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns. (Note that these restrictions are separate from whether the name is a key word or not; quoting a name will not allow you to escape these restrictions.) You do not really need to be concerned about these columns; just know they exist.

`tableoid`

The OID of the table containing this row. This column is particularly handy for queries that select from partitioned tables (see Section 5.12) or inheritance hierarchies (see Section 5.11), since without it, it's difficult to tell which individual table a row came from. The `tableoid` can be joined against the `oid` column of `pg_class` to obtain the table name.

`xmin`

The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)

`cmin`

The command identifier (starting at zero) within the inserting transaction.

`xmax`

The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

`cmax`

The command identifier within the deleting transaction, or zero.

`ctid`

The physical location of the row version within its table. Note that although the `ctid` can be used to locate the row version very quickly, a row's `ctid` will change if it is updated or moved by `VACUUM FULL`. Therefore `ctid` is useless as a long-term row identifier. A primary key should be used to identify logical rows.

Transaction identifiers are also 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see Chapter 24 for details. It is unwise, however, to depend on the uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of 2^{32} (4 billion) SQL commands within a single transaction. In practice this limit is not a problem — note that the limit is on the number of SQL commands, not the number of rows processed. Also, only commands that actually modify the database contents will consume a command identifier.

5.7. Modifying Tables

When you create a table and you realize that you made a mistake, or the requirements of the application change, you can drop the table and create it again. But this is not a convenient option if the table is already filled with data, or if the table is referenced by other database objects (for instance a foreign key constraint). Therefore PostgreSQL provides a family of commands to make modifications to existing tables. Note that this is conceptually distinct from altering the data contained in the table: here we are interested in altering the definition, or structure, of the table.

You can:

- Add columns
- Remove columns
- Add constraints
- Remove constraints
- Change default values
- Change column data types
- Rename columns
- Rename tables

All these actions are performed using the ALTER TABLE command, whose reference page contains details beyond those given here.

5.7.1. Adding a Column

To add a column, use a command like:

```
ALTER TABLE products ADD COLUMN description text;
```

The new column is initially filled with whatever default value is given (null if you don't specify a `DEFAULT` clause).

> **Tip**
>
> From PostgreSQL 11, adding a column with a constant default value no longer means that each row of the table needs to be updated when the `ALTER TABLE` statement is executed. Instead, the default value will be returned the next time the row is accessed, and applied when the table is rewritten, making the `ALTER TABLE` very fast even on large tables.
>
> However, if the default value is volatile (e.g., `clock_timestamp()`) each row will need to be updated with the value calculated at the time `ALTER TABLE` is executed. To avoid a potentially lengthy update operation, particularly if you intend to fill the column with mostly nondefault values anyway, it may be preferable to add the column with no default, insert the correct values using `UPDATE`, and then add any desired default as described below.

You can also define constraints on the column at the same time, using the usual syntax:

```
ALTER TABLE products ADD COLUMN description text CHECK (description
 <> '');
```

In fact all the options that can be applied to a column description in `CREATE TABLE` can be used here. Keep in mind however that the default value must satisfy the given constraints, or the `ADD` will fail. Alternatively, you can add constraints later (see below) after you've filled in the new column correctly.

5.7.2. Removing a Column

To remove a column, use a command like:

```
ALTER TABLE products DROP COLUMN description;
```

Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQL will not silently drop that constraint. You can authorize dropping everything that depends on the column by adding CASCADE:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

See Section 5.15 for a description of the general mechanism behind this.

5.7.3. Adding a Constraint

To add a constraint, the table constraint syntax is used. For example:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES
 product_groups;
```

To add a not-null constraint, which cannot be written as a table constraint, use this syntax:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

The constraint will be checked immediately, so the table data must satisfy the constraint before it can be added.

5.7.4. Removing a Constraint

To remove a constraint you need to know its name. If you gave it a name then that's easy. Otherwise the system assigned a generated name, which you need to find out. The psql command `\d` *`table-name`* can be helpful here; other interfaces might also provide a way to inspect table details. Then the command is:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(If you are dealing with a generated constraint name like `$2`, don't forget that you'll need to double-quote it to make it a valid identifier.)

As with dropping a column, you need to add CASCADE if you want to drop a constraint that something else depends on. An example is that a foreign key constraint depends on a unique or primary key constraint on the referenced column(s).

This works the same for all constraint types except not-null constraints. To drop a not-null constraint use:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Recall that not-null constraints do not have names.)

5.7.5. Changing a Column's Default Value

To set a new default for a column, use a command like:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Note that this doesn't affect any existing rows in the table, it just changes the default for future `INSERT` commands.

To remove any default value, use:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

This is effectively the same as setting the default to null. As a consequence, it is not an error to drop a default where one hadn't been defined, because the default is implicitly the null value.

5.7.6. Changing a Column's Data Type

To convert a column to a different data type, use a command like:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

This will succeed only if each existing entry in the column can be converted to the new type by an implicit cast. If a more complex conversion is needed, you can add a `USING` clause that specifies how to compute the new values from the old.

PostgreSQL will attempt to convert the column's default value (if any) to the new type, as well as any constraints that involve the column. But these conversions might fail, or might produce surprising results. It's often best to drop any constraints on the column before altering its type, and then add back suitably modified constraints afterwards.

5.7.7. Renaming a Column

To rename a column:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.7.8. Renaming a Table

To rename a table:

```
ALTER TABLE products RENAME TO items;
```

5.8. Privileges

When an object is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, *privileges* must be granted.

There are different kinds of privileges: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, `USAGE`, `SET`, `ALTER SYSTEM`, and `MAINTAIN`. The privileges applicable to a particular object vary depending on the object's type

(table, function, etc.). More detail about the meanings of these privileges appears below. The following sections and chapters will also show you how these privileges are used.

The right to modify or destroy an object is inherent in being the object's owner, and cannot be granted or revoked in itself. (However, like all privileges, that right can be inherited by members of the owning role; see Section 21.3.)

An object can be assigned to a new owner with an ALTER command of the appropriate kind for the object, for example

```
ALTER TABLE table_name OWNER TO new_owner;
```

Superusers can always do this; ordinary roles can only do it if they are both the current owner of the object (or inherit the privileges of the owning role) and able to SET ROLE to the new owning role.

To assign privileges, the GRANT command is used. For example, if joe is an existing role, and accounts is an existing table, the privilege to update the table can be granted with:

```
GRANT UPDATE ON accounts TO joe;
```

Writing ALL in place of a specific privilege grants all privileges that are relevant for the object type.

The special “role” name PUBLIC can be used to grant a privilege to every role on the system. Also, “group” roles can be set up to help manage privileges when there are many users of a database — for details see Chapter 21.

To revoke a previously-granted privilege, use the fittingly named REVOKE command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Ordinarily, only the object's owner (or a superuser) can grant or revoke privileges on an object. However, it is possible to grant a privilege “with grant option”, which gives the recipient the right to grant it in turn to others. If the grant option is subsequently revoked then all who received the privilege from that recipient (directly or through a chain of grants) will lose the privilege. For details see the GRANT and REVOKE reference pages.

An object's owner can choose to revoke their own ordinary privileges, for example to make a table read-only for themselves as well as others. But owners are always treated as holding all grant options, so they can always re-grant their own privileges.

The available privileges are:

SELECT

Allows SELECT from any column, or specific column(s), of a table, view, materialized view, or other table-like object. Also allows use of COPY TO. This privilege is also needed to reference existing column values in UPDATE, DELETE, or MERGE. For sequences, this privilege also allows use of the currval function. For large objects, this privilege allows the object to be read.

INSERT

Allows INSERT of a new row into a table, view, etc. Can be granted on specific column(s), in which case only those columns may be assigned to in the INSERT command (other columns will therefore receive default values). Also allows use of COPY FROM.

UPDATE

Allows UPDATE of any column, or specific column(s), of a table, view, etc. (In practice, any nontrivial UPDATE command will require SELECT privilege as well, since it must reference

table columns to determine which rows to update, and/or to compute new values for columns.) `SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE` also require this privilege on at least one column, in addition to the `SELECT` privilege. For sequences, this privilege allows use of the `nextval` and `setval` functions. For large objects, this privilege allows writing or truncating the object.

`DELETE`

Allows `DELETE` of a row from a table, view, etc. (In practice, any nontrivial `DELETE` command will require `SELECT` privilege as well, since it must reference table columns to determine which rows to delete.)

`TRUNCATE`

Allows `TRUNCATE` on a table.

`REFERENCES`

Allows creation of a foreign key constraint referencing a table, or specific column(s) of a table.

`TRIGGER`

Allows creation of a trigger on a table, view, etc.

`CREATE`

For databases, allows new schemas and publications to be created within the database, and allows trusted extensions to be installed within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object *and* have this privilege for the containing schema.

For tablespaces, allows tables, indexes, and temporary files to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace.

Note that revoking this privilege will not alter the existence or location of existing objects.

`CONNECT`

Allows the grantee to connect to the database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by `pg_hba.conf`).

`TEMPORARY`

Allows temporary tables to be created while using the database.

`EXECUTE`

Allows calling a function or procedure, including use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions and procedures.

`USAGE`

For procedural languages, allows use of the language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to “look up” objects within the schema. Without this permission, it is still possible to see the object names, e.g., by querying system catalogs. Also, after revoking this permission, existing sessions might have statements that have previously performed this lookup, so this is not a completely secure way to prevent object access.

For sequences, allows use of the `currval` and `nextval` functions.

For types and domains, allows use of the type or domain in the creation of tables, functions, and other schema objects. (Note that this privilege does not control all "usage" of the type, such as values of the type appearing in queries. It only prevents objects from being created that depend on the type. The main purpose of this privilege is controlling which users can create dependencies on a type, which could prevent the owner from changing the type later.)

For foreign-data wrappers, allows creation of new servers using the foreign-data wrapper.

For foreign servers, allows creation of foreign tables using the server. Grantees may also create, alter, or drop their own user mappings associated with that server.

`SET`

Allows a server configuration parameter to be set to a new value within the current session. (While this privilege can be granted on any parameter, it is meaningless except for parameters that would normally require superuser privilege to set.)

`ALTER SYSTEM`

Allows a server configuration parameter to be configured to a new value using the ALTER SYSTEM command.

`MAINTAIN`

Allows `VACUUM`, `ANALYZE`, `CLUSTER`, `REFRESH MATERIALIZED VIEW`, `REINDEX`, and `LOCK TABLE` on a relation.

The privileges required by other commands are listed on the reference page of the respective command.

PostgreSQL grants privileges on some types of objects to `PUBLIC` by default when the objects are created. No privileges are granted to `PUBLIC` by default on tables, table columns, sequences, foreign data wrappers, foreign servers, large objects, schemas, tablespaces, or configuration parameters. For other types of objects, the default privileges granted to `PUBLIC` are as follows: `CONNECT` and `TEMPORARY` (create temporary tables) privileges for databases; `EXECUTE` privilege for functions and procedures; and `USAGE` privilege for languages and data types (including domains). The object owner can, of course, `REVOKE` both default and expressly granted privileges. (For maximum security, issue the `REVOKE` in the same transaction that creates the object; then there is no window in which another user can use the object.) Also, these default privilege settings can be overridden using the ALTER DEFAULT PRIVILEGES command.

Table 5.1 shows the one-letter abbreviations that are used for these privilege types in *ACL* (Access Control List) values. You will see these letters in the output of the psql commands listed below, or when looking at ACL columns of system catalogs.

Table 5.1. ACL Privilege Abbreviations

Privilege	Abbreviation	Applicable Object Types
`SELECT`	`r` ("read")	`LARGE OBJECT`, `SEQUENCE`, `TABLE` (and table-like objects), table column
`INSERT`	`a` ("append")	`TABLE`, table column
`UPDATE`	`w` ("write")	`LARGE OBJECT`, `SEQUENCE`, `TABLE`, table column
`DELETE`	`d`	`TABLE`
`TRUNCATE`	`D`	`TABLE`
`REFERENCES`	`x`	`TABLE`, table column
`TRIGGER`	`t`	`TABLE`

Privilege	Abbreviation	Applicable Object Types
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE
SET	s	PARAMETER
ALTER SYSTEM	A	PARAMETER
MAINTAIN	m	TABLE

Table 5.2 summarizes the privileges available for each type of SQL object, using the abbreviations shown above. It also shows the psql command that can be used to examine privilege settings for each object type.

Table 5.2. Summary of Access Privileges

Object Type	All Privileges	Default PUBLIC Privileges	psql Command
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	\dl+
PARAMETER	sA	none	\dconfig+
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp
TABLE (and table-like objects)	arwdDxtm	none	\dp
Table column	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

The privileges that have been granted for a particular object are displayed as a list of aclitem entries, each having the format:

grantee=privilege-abbreviation[*]*.../grantor*

Each aclitem lists all the permissions of one grantee that have been granted by a particular grantor. Specific privileges are represented by one-letter abbreviations from Table 5.1, with * appended if the privilege was granted with grant option. For example, calvin=r*w/hobbes specifies that the role calvin has the privilege SELECT (r) with grant option (*) as well as the non-grantable privilege UPDATE (w), both granted by the role hobbes. If calvin also has some privileges on the same object granted by a different grantor, those would appear as a separate aclitem entry. An empty grantee field in an aclitem stands for PUBLIC.

As an example, suppose that user miriam creates table mytable and does:

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

Then psql's `\dp` command would show:

```
=> \dp mytable
                                  Access privileges
 Schema |  Name   | Type  |   Access privileges    |   Column
 privileges   | Policies
--------+---------+-------+------------------------
+-----------------------+----------
 public | mytable | table | miriam=arwdDxtm/miriam+| col1:
       +|
        |         |       | =r/miriam             +|
 miriam_rw=rw/miriam |
        |         |       | admin=arw/miriam       |
        |
(1 row)
```

If the “Access privileges” column is empty for a given object, it means the object has default privileges (that is, its privileges entry in the relevant system catalog is null). Default privileges always include all privileges for the owner, and can include some privileges for `PUBLIC` depending on the object type, as explained above. The first `GRANT` or `REVOKE` on an object will instantiate the default privileges (producing, for example, `miriam=arwdDxt/miriam`) and then modify them per the specified request. Similarly, entries are shown in “Column privileges” only for columns with nondefault privileges. (Note: for this purpose, “default privileges” always means the built-in default privileges for the object's type. An object whose privileges have been affected by an `ALTER DEFAULT PRIVILEGES` command will always be shown with an explicit privilege entry that includes the effects of the `ALTER`.)

Notice that the owner's implicit grant options are not marked in the access privileges display. A `*` will appear only when grant options have been explicitly granted to someone.

The “Access privileges” column shows `(none)` when the object's privileges entry is non-null but empty. This means that no privileges are granted at all, even to the object's owner — a rare situation. (The owner still has implicit grant options in this case, and so could re-grant her own privileges; but she has none at the moment.)

5.9. Row Security Policies

In addition to the SQL-standard privilege system available through GRANT, tables can have *row security policies* that restrict, on a per-user basis, which rows can be returned by normal queries or inserted, updated, or deleted by data modification commands. This feature is also known as *Row-Level Security*. By default, tables do not have any policies, so that if a user has access privileges to a table according to the SQL privilege system, all rows within it are equally available for querying or updating.

When row security is enabled on a table (with ALTER TABLE ... ENABLE ROW LEVEL SECURITY), all normal access to the table for selecting rows or modifying rows must be allowed by a row security policy. (However, the table's owner is typically not subject to row security policies.) If no policy exists for the table, a default-deny policy is used, meaning that no rows are visible or can be modified. Operations that apply to the whole table, such as `TRUNCATE` and `REFERENCES`, are not subject to row security.

Row security policies can be specific to commands, or to roles, or to both. A policy can be specified to apply to `ALL` commands, or to `SELECT`, `INSERT`, `UPDATE`, or `DELETE`. Multiple roles can be assigned to a given policy, and normal role membership and inheritance rules apply.

To specify which rows are visible or modifiable according to a policy, an expression is required that returns a Boolean result. This expression will be evaluated for each row prior to any conditions or functions coming from the user's query. (The only exceptions to this rule are `leakproof` functions, which are guaranteed to not leak information; the optimizer may choose to apply such functions ahead of the row-security check.) Rows for which the expression does not return `true` will not be processed. Separate expressions may be specified to provide independent control over the rows which are visible and the rows which are allowed to be modified. Policy expressions are run as part of the query and with the privileges of the user running the query, although security-definer functions can be used to access data not available to the calling user.

Superusers and roles with the `BYPASSRLS` attribute always bypass the row security system when accessing a table. Table owners normally bypass row security as well, though a table owner can choose to be subject to row security with ALTER TABLE ... FORCE ROW LEVEL SECURITY.

Enabling and disabling row security, as well as adding policies to a table, is always the privilege of the table owner only.

Policies are created using the CREATE POLICY command, altered using the ALTER POLICY command, and dropped using the DROP POLICY command. To enable and disable row security for a given table, use the ALTER TABLE command.

Each policy has a name and multiple policies can be defined for a table. As policies are table-specific, each policy for a table must have a unique name. Different tables may have policies with the same name.

When multiple policies apply to a given query, they are combined using either `OR` (for permissive policies, which are the default) or using `AND` (for restrictive policies). This is similar to the rule that a given role has the privileges of all roles that they are a member of. Permissive vs. restrictive policies are discussed further below.

As a simple example, here is how to create a policy on the `account` relation to allow only members of the `managers` role to access rows, and only rows of their accounts:

```
CREATE TABLE accounts (manager text, company text, contact_email
 text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

The policy above implicitly provides a `WITH CHECK` clause identical to its `USING` clause, so that the constraint applies both to rows selected by a command (so a manager cannot `SELECT`, `UPDATE`, or `DELETE` existing rows belonging to a different manager) and to rows modified by a command (so rows belonging to a different manager cannot be created via `INSERT` or `UPDATE`).

If no role is specified, or the special user name `PUBLIC` is used, then the policy applies to all users on the system. To allow all users to access only their own row in a `users` table, a simple policy can be used:

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

This works similarly to the previous example.

To use a different policy for rows that are being added to the table compared to those rows that are visible, multiple policies can be combined. This pair of policies would allow all users to view all rows in the `users` table, but only modify their own:

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

In a `SELECT` command, these two policies are combined using `OR`, with the net effect being that all rows can be selected. In other command types, only the second policy applies, so that the effects are the same as before.

Row security can also be disabled with the `ALTER TABLE` command. Disabling row security does not remove any policies that are defined on the table; they are simply ignored. Then all rows in the table are visible and modifiable, subject to the standard SQL privileges system.

Below is a larger example of how this feature can be used in production environments. The table `passwd` emulates a Unix password file:

```
-- Simple passwd-file based example
CREATE TABLE passwd (
  user_name             text UNIQUE NOT NULL,
  pwhash                text,
  uid                   int  PRIMARY KEY,
  gid                   int  NOT NULL,
  real_name             text NOT NULL,
  home_phone            text,
  extra_info            text,
  home_dir              text NOT NULL,
  shell                 text NOT NULL
);

CREATE ROLE admin;  -- Administrator
CREATE ROLE bob;    -- Normal user
CREATE ROLE alice;  -- Normal user

-- Populate the table
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/
dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/
zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/
bin/zsh');

-- Be sure to enable row-level security on the table
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- Create policies
-- Administrator can see all rows and add any rows
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK
 (true);
-- Normal users can view all rows
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- Normal users can update their own records, but
-- limit which shells a normal user is allowed to set
CREATE POLICY user_mod ON passwd FOR UPDATE
```

```
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/
tcsh')
  );

-- Allow admin all normal rights
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- Users only get select access on public columns
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info,
 home_dir, shell)
  ON passwd TO public;
-- Allow users to update certain columns
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

As with any security settings, it's important to test and ensure that the system is behaving as expected. Using the example above, this demonstrates that the permission system is working properly.

```
-- admin can view all rows and fields
postgres=> set role admin;
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  |
 extra_info | home_dir    |   shell
-----------+--------+-----+-----+-----------+--------------
+------------+-------------+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |
     | /root       | /bin/dash
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |
     | /home/bob   | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |
     | /home/alice | /bin/zsh
(3 rows)

-- Test what Alice is able to do
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for table passwd
postgres=> select
 user_name,real_name,home_phone,extra_info,home_dir,shell from
 passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |
  shell
-----------+-----------+--------------+------------+-------------
+-----------
 admin     | Admin     | 111-222-3333 |            | /root
 | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob
 | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice
 | /bin/zsh
(3 rows)
```

```
postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for table passwd
-- Alice is allowed to change her own real_name, but no others
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name
 = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for table passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for table passwd
-- Alice can change her own password; RLS silently prevents
 updating other rows
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

All of the policies constructed thus far have been permissive policies, meaning that when multiple policies are applied they are combined using the "OR" Boolean operator. While permissive policies can be constructed to only allow access to rows in the intended cases, it can be simpler to combine permissive policies with restrictive policies (which the records must pass and which are combined using the "AND" Boolean operator). Building on the example above, we add a restrictive policy to require the administrator to be connected over a local Unix socket to access the records of the `passwd` table:

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

We can then see that an administrator connecting over a network will not see any records, due to the restrictive policy:

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone |
 extra_info | home_dir | shell
-----------+--------+-----+-----+-----------+------------
+------------+----------+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

Referential integrity checks, such as unique or primary key constraints and foreign key references, always bypass row security to ensure that data integrity is maintained. Care must be taken when developing schemas and row level policies to avoid “covert channel” leaks of information through such referential integrity checks.

In some contexts it is important to be sure that row security is not being applied. For example, when taking a backup, it could be disastrous if row security silently caused some rows to be omitted from the backup. In such a situation, you can set the row_security configuration parameter to `off`. This does not in itself bypass row security; what it does is throw an error if any query's results would get filtered by a policy. The reason for the error can then be investigated and fixed.

In the examples above, the policy expressions consider only the current values in the row to be accessed or updated. This is the simplest and best-performing case; when possible, it's best to design row security applications to work this way. If it is necessary to consult other rows or other tables to make a policy decision, that can be accomplished using sub-`SELECT`s, or functions that contain `SELECT`s, in the policy expressions. Be aware however that such accesses can create race conditions that could allow information leakage if care is not taken. As an example, consider the following table design:

```
-- definition of privilege groups
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice is the administrator
GRANT SELECT ON groups TO public;

-- definition of users' privilege levels
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- table holding the information to be protected
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- a row should be visible to/updatable by users whose security
 group_id is
-- greater than or equal to the row's group_id
CREATE POLICY fp_s ON information FOR SELECT
```

```
  USING (group_id <= (SELECT group_id FROM users WHERE user_name =
 current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name =
 current_user));

-- we rely only on RLS to protect the information table
GRANT ALL ON information TO public;
```

Now suppose that `alice` wishes to change the "slightly secret" information, but decides that `mallory` should not be trusted with the new content of that row, so she does:

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id
 = 2;
COMMIT;
```

That looks safe; there is no window wherein `mallory` should be able to see the "secret from mallory" string. However, there is a race condition here. If `mallory` is concurrently doing, say,

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

and her transaction is in `READ COMMITTED` mode, it is possible for her to see "secret from mallory". That happens if her transaction reaches the `information` row just after `alice`'s does. It blocks waiting for `alice`'s transaction to commit, then fetches the updated row contents thanks to the `FOR UPDATE` clause. However, it does *not* fetch an updated row for the implicit `SELECT` from `users`, because that sub-`SELECT` did not have `FOR UPDATE`; instead the `users` row is read with the snapshot taken at the start of the query. Therefore, the policy expression tests the old value of `mallory`'s privilege level and allows her to see the updated row.

There are several ways around this problem. One simple answer is to use `SELECT ... FOR SHARE` in sub-`SELECT`s in row security policies. However, that requires granting `UPDATE` privilege on the referenced table (here `users`) to the affected users, which might be undesirable. (But another row security policy could be applied to prevent them from actually exercising that privilege; or the sub-`SELECT` could be embedded into a security definer function.) Also, heavy concurrent use of row share locks on the referenced table could pose a performance problem, especially if updates of it are frequent. Another solution, practical if updates of the referenced table are infrequent, is to take an `ACCESS EXCLUSIVE` lock on the referenced table when updating it, so that no concurrent transactions could be examining old row values. Or one could just wait for all concurrent transactions to end after committing an update of the referenced table and before making changes that rely on the new security situation.

For additional details see CREATE POLICY and ALTER TABLE.

5.10. Schemas

A PostgreSQL database cluster contains one or more named databases. Roles and a few other object types are shared across the entire cluster. A client connection to the server can only access data in a single database, the one specified in the connection request.

> **Note**
>
> Users of a cluster do not necessarily have the privilege to access every database in the cluster. Sharing of role names means that there cannot be different roles named, say, `joe` in two

databases in the same cluster; but the system can be configured to allow `joe` access to only some of the databases.

A database contains one or more named *schemas*, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. Within one schema, two objects of the same type cannot have the same name. Furthermore, tables, sequences, indexes, views, materialized views, and foreign tables share the same namespace, so that, for example, an index and a table must have different names if they are in the same schema. The same object name can be used in different schemas without conflict; for example, both `schema1` and `myschema` can contain tables named `mytable`. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database they are connected to, if they have privileges to do so.

There are several reasons why one might want to use schemas:

- To allow many users to use one database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

5.10.1. Creating a Schema

To create a schema, use the CREATE SCHEMA command. Give the schema a name of your choice. For example:

```
CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a *qualified name* consisting of the schema name and table name separated by a dot:

```
schema.table
```

This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters. (For brevity we will speak of tables only, but the same ideas apply to other kinds of named objects, such as types and functions.)

Actually, the even more general syntax

```
database.schema.table
```

can be used too, but at present this is just for pro forma compliance with the SQL standard. If you write a database name, it must be the same as the database you are connected to.

So to create a table in the new schema, use:

```
CREATE TABLE myschema.mytable (
 ...
```

```
);
```

To drop a schema if it's empty (all objects in it have been dropped), use:

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

See Section 5.15 for a description of the general mechanism behind this.

Often you will want to create a schema owned by someone else (since this is one of the ways to restrict the activities of your users to well-defined namespaces). The syntax for that is:

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

You can even omit the schema name, in which case the schema name will be the same as the user name. See Section 5.10.6 for how this can be useful.

Schema names beginning with `pg_` are reserved for system purposes and cannot be created by users.

5.10.2. The Public Schema

In the previous sections we created tables without specifying any schema names. By default such tables (and other objects) are automatically put into a schema named “public”. Every new database contains such a schema. Thus, the following are equivalent:

```
CREATE TABLE products ( ... );
```

and:

```
CREATE TABLE public.products ( ... );
```

5.10.3. The Schema Search Path

Qualified names are tedious to write, and it's often best not to wire a particular schema name into applications anyway. Therefore tables are often referred to by *unqualified names*, which consist of just the table name. The system determines which table is meant by following a *search path*, which is a list of schemas to look in. The first matching table in the search path is taken to be the one wanted. If there is no match in the search path, an error is reported, even if matching table names exist in other schemas in the database.

The ability to create like-named objects in different schemas complicates writing a query that references precisely the same objects every time. It also opens up the potential for users to change the behavior of other users' queries, maliciously or accidentally. Due to the prevalence of unqualified names in queries and their use in PostgreSQL internals, adding a schema to `search_path` effectively trusts all users having `CREATE` privilege on that schema. When you run an ordinary query, a malicious user able to create objects in a schema of your search path can take control and execute arbitrary SQL functions as though you executed them.

The first schema named in the search path is called the current schema. Aside from being the first schema searched, it is also the schema in which new tables will be created if the `CREATE TABLE` command does not specify a schema name.

To show the current search path, use the following command:

```
SHOW search_path;
```

In the default setup this returns:

```
 search_path
--------------
 "$user", public
```

The first element specifies that a schema with the same name as the current user is to be searched. If no such schema exists, the entry is ignored. The second element refers to the public schema that we have seen already.

The first schema in the search path that exists is the default location for creating new objects. That is the reason that by default objects are created in the public schema. When objects are referenced in any other context without schema qualification (table modification, data modification, or query commands) the search path is traversed until a matching object is found. Therefore, in the default configuration, any unqualified access again can only refer to the public schema.

To put our new schema in the path, we use:

```
SET search_path TO myschema,public;
```

(We omit the `$user` here because we have no immediate need for it.) And then we can access the table without schema qualification:

```
DROP TABLE mytable;
```

Also, since `myschema` is the first element in the path, new objects would by default be created in it.

We could also have written:

```
SET search_path TO myschema;
```

Then we no longer have access to the public schema without explicit qualification. There is nothing special about the public schema except that it exists by default. It can be dropped, too.

See also Section 9.27 for other ways to manipulate the schema search path.

The search path works in the same way for data type names, function names, and operator names as it does for table names. Data type and function names can be qualified in exactly the same way as table names. If you need to write a qualified operator name in an expression, there is a special provision: you must write

```
OPERATOR(schema.operator)
```

This is needed to avoid syntactic ambiguity. An example is:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

In practice one usually relies on the search path for operators, so as not to have to write anything so ugly as that.

5.10.4. Schemas and Privileges

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema must grant the `USAGE` privilege on the schema. By default, everyone has that privilege on the schema `public`. To allow users to make use of the objects in a schema, additional privileges might need to be granted, as appropriate for the object.

A user can also be allowed to create objects in someone else's schema. To allow that, the `CREATE` privilege on the schema needs to be granted. In databases upgraded from PostgreSQL 14 or earlier, everyone has that privilege on the schema `public`. Some usage patterns call for revoking that privilege:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(The first “public” is the schema, the second “public” means “every user”. In the first sense it is an identifier, in the second sense it is a key word, hence the different capitalization; recall the guidelines from Section 4.1.1.)

5.10.5. The System Catalog Schema

In addition to `public` and user-created schemas, each database contains a `pg_catalog` schema, which contains the system tables and all the built-in data types, functions, and operators. `pg_catalog` is always effectively part of the search path. If it is not named explicitly in the path then it is implicitly searched *before* searching the path's schemas. This ensures that built-in names will always be findable. However, you can explicitly place `pg_catalog` at the end of your search path if you prefer to have user-defined names override built-in names.

Since system table names begin with `pg_`, it is best to avoid such names to ensure that you won't suffer a conflict if some future version defines a system table named the same as your table. (With the default search path, an unqualified reference to your table name would then be resolved as the system table instead.) System tables will continue to follow the convention of having names beginning with `pg_`, so that they will not conflict with unqualified user-table names so long as users avoid the `pg_` prefix.

5.10.6. Usage Patterns

Schemas can be used to organize your data in many ways. A *secure schema usage pattern* prevents untrusted users from changing the behavior of other users' queries. When a database does not use a secure schema usage pattern, users wishing to securely query that database would take protective action at the beginning of each session. Specifically, they would begin each session by setting `search_path` to the empty string or otherwise removing schemas that are writable by non-superusers from `search_path`. There are a few usage patterns easily supported by the default configuration:

- Constrain ordinary users to user-private schemas. To implement this pattern, first ensure that no schemas have public `CREATE` privileges. Then, for every user needing to create non-temporary objects, create a schema with the same name as that user, for example `CREATE SCHEMA alice AUTHORIZATION alice`. (Recall that the default search path starts with `$user`, which resolves to the user name. Therefore, if each user has a separate schema, they access their own schemas by default.) This pattern is a secure schema usage pattern unless an untrusted user is the database owner or has been granted `ADMIN OPTION` on a relevant role, in which case no secure schema usage pattern exists.

In PostgreSQL 15 and later, the default configuration supports this usage pattern. In prior versions, or when using a database that has been upgraded from a prior version, you will need to remove the public `CREATE` privilege from the `public` schema (issue `REVOKE CREATE ON SCHEMA public FROM PUBLIC`). Then consider auditing the `public` schema for objects named like objects in schema `pg_catalog`.

- Remove the public schema from the default search path, by modifying `postgresql.conf` or by issuing `ALTER ROLE ALL SET search_path = "$user"`. Then, grant privileges to create in the public schema. Only qualified names will choose public schema objects. While qualified table references are fine, calls to functions in the public schema will be unsafe or unreliable. If you create functions or extensions in the public schema, use the first pattern instead. Otherwise, like the first pattern, this is secure unless an untrusted user is the database owner or has been granted `ADMIN OPTION` on a relevant role.

- Keep the default search path, and grant privileges to create in the public schema. All users access the public schema implicitly. This simulates the situation where schemas are not available at all, giving a smooth transition from the non-schema-aware world. However, this is never a secure pattern. It is acceptable only when the database has a single user or a few mutually-trusting users. In databases upgraded from PostgreSQL 14 or earlier, this is the default.

For any pattern, to install shared applications (tables to be used by everyone, additional functions provided by third parties, etc.), put them into separate schemas. Remember to grant appropriate privileges to allow the other users to access them. Users can then refer to these additional objects by qualifying the names with a schema name, or they can put the additional schemas into their search path, as they choose.

5.10.7. Portability

In the SQL standard, the notion of objects in the same schema being owned by different users does not exist. Moreover, some implementations do not allow you to create schemas that have a different name than their owner. In fact, the concepts of schema and user are nearly equivalent in a database system that implements only the basic schema support specified in the standard. Therefore, many users consider qualified names to really consist of *`user_name.table_name`*. This is how PostgreSQL will effectively behave if you create a per-user schema for every user.

Also, there is no concept of a `public` schema in the SQL standard. For maximum conformance to the standard, you should not use the `public` schema.

Of course, some SQL database systems might not implement schemas at all, or provide namespace support by allowing (possibly limited) cross-database access. If you need to work with those systems, then maximum portability would be achieved by not using schemas at all.

5.11. Inheritance

PostgreSQL implements table inheritance, which can be a useful tool for database designers. (SQL:1999 and later define a type inheritance feature, which differs in many respects from the features described here.)

Let's start with an example: suppose we are trying to build a data model for cities. Each state has many cities, but only one capital. We want to be able to quickly retrieve the capital city for any particular state. This can be done by creating two tables, one for state capitals and one for cities that are not capitals. However, what happens when we want to ask for data about a city, regardless of whether it is a capital or not? The inheritance feature can help to resolve this problem. We define the `capitals` table so that it inherits from `cities`:

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- in feet
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, the `capitals` table *inherits* all the columns of its parent table, `cities`. State capitals also have an extra column, `state`, that shows their state.

In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendant tables. The latter behavior is the default. For example, the following query finds the names of all cities, including state capitals, that are located at an elevation over 500 feet:

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

Given the sample data from the PostgreSQL tutorial (see Section 2.1), this returns:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

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

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

   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

Here the `ONLY` keyword indicates that the query should apply only to `cities`, and not any tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed — `SELECT`, `UPDATE` and `DELETE` — support the `ONLY` keyword.

You can also write the table name with a trailing `*` to explicitly specify that descendant tables are included:

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

Writing `*` is not necessary, since this behavior is always the default. However, this syntax is still supported for compatibility with older releases where the default could be changed.

In some cases you might wish to know which table a particular row originated from. There is a system column called `tableoid` in each table which can tell you the originating table:

```
SELECT c.tableoid, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

which returns:

```
 tableoid |   name    | elevation
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

(If you try to reproduce this example, you will probably get different numeric OIDs.) By doing a join with `pg_class` you can see the actual table names:

```
SELECT p.relname, c.name, c.elevation
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

which returns:

```
 relname  |   name    | elevation
----------+-----------+-----------
 cities   | Las Vegas |      2174
 cities   | Mariposa  |      1953
 capitals | Madison   |       845
```

Another way to get the same effect is to use the `regclass` alias type, which will print the table OID symbolically:

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

Inheritance does not automatically propagate data from `INSERT` or `COPY` commands to other tables in the inheritance hierarchy. In our example, the following `INSERT` statement will fail:

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

We might hope that the data would somehow be routed to the `capitals` table, but this does not happen: `INSERT` always inserts into exactly the table specified. In some cases it is possible to redirect the insertion using a rule (see Chapter 39). However that does not help for the above case because the `cities` table does not contain the column `state`, and so the command will be rejected before the rule can be applied.

All check constraints and not-null constraints on a parent table are automatically inherited by its children, unless explicitly specified otherwise with `NO INHERIT` clauses. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

A table can inherit from more than one parent table, in which case it has the union of the columns defined by the parent tables. Any columns declared in the child table's definition are added to these.

If the same column name appears in multiple parent tables, or in both a parent table and the child's definition, then these columns are “merged” so that there is only one such column in the child table. To be merged, columns must have the same data types, else an error is raised. Inheritable check constraints and not-null constraints are merged in a similar fashion. Thus, for example, a merged column will be marked not-null if any one of the column definitions it came from is marked not-null. Check constraints are merged if they have the same name, and the merge will fail if their conditions are different.

Table inheritance is typically established when the child table is created, using the `INHERITS` clause of the `CREATE TABLE` statement. Alternatively, a table which is already defined in a compatible way can have a new parent relationship added, using the `INHERIT` variant of `ALTER TABLE`. To do this the new child table must already include columns with the same names and types as the columns of the parent. It must also include check constraints with the same names and check expressions as those of the parent. Similarly an inheritance link can be removed from a child using the `NO INHERIT` variant of `ALTER TABLE`. Dynamically adding and removing inheritance links like this can be useful when the inheritance relationship is being used for table partitioning (see Section 5.12).

One convenient way to create a compatible table that will later be made a new child is to use the `LIKE` clause in `CREATE TABLE`. This creates a new table with the same columns as the source table. If there are any `CHECK` constraints defined on the source table, the `INCLUDING CONSTRAINTS` option to `LIKE` should be specified, as the new child must have constraints matching the parent to be considered compatible.

A parent table cannot be dropped while any of its children remain. Neither can columns or check constraints of child tables be dropped or altered if they are inherited from any parent tables. If you wish to remove a table and all of its descendants, one easy way is to drop the parent table with the `CASCADE` option (see Section 5.15).

`ALTER TABLE` will propagate any changes in column data definitions and check constraints down the inheritance hierarchy. Again, dropping columns that are depended on by other tables is only possible when using the `CASCADE` option. `ALTER TABLE` follows the same rules for duplicate column merging and rejection that apply during `CREATE TABLE`.

Inherited queries perform access permission checks on the parent table only. Thus, for example, granting `UPDATE` permission on the `cities` table implies permission to update rows in the `capitals` table as well, when they are accessed through `cities`. This preserves the appearance that the data is (also) in the parent table. But the `capitals` table could not be updated directly without an additional grant. In a similar way, the parent table's row security policies (see Section 5.9) are applied to rows coming from child tables during an inherited query. A child table's policies, if any, are applied only when it is the table explicitly named in the query; and in that case, any policies attached to its parent(s) are ignored.

Foreign tables (see Section 5.13) can also be part of inheritance hierarchies, either as parent or child tables, just as regular tables can be. If a foreign table is part of an inheritance hierarchy then any operations not supported by the foreign table are not supported on the whole hierarchy either.

5.11.1. Caveats

Note that not all SQL commands are able to work on inheritance hierarchies. Commands that are used for data querying, data modification, or schema modification (e.g., `SELECT`, `UPDATE`, `DELETE`, most variants of `ALTER TABLE`, but not `INSERT` or `ALTER TABLE ... RENAME`) typically default to including child tables and support the `ONLY` notation to exclude them. Commands that do database maintenance and tuning (e.g., `REINDEX`, `VACUUM`) typically only work on individual, physical tables and do not support recursing over inheritance hierarchies. The respective behavior of each individual command is documented in its reference page (SQL Commands).

A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint. Thus, in the terms of the above example:

- If we declared `cities.name` to be `UNIQUE` or a `PRIMARY KEY`, this would not stop the `capitals` table from having rows with names duplicating rows in `cities`. And those duplicate rows would by default show up in queries from `cities`. In fact, by default `capitals` would have no unique constraint at all, and so could contain multiple rows with the same name. You could add a unique constraint to `capitals`, but this would not prevent duplication compared to `cities`.

- Similarly, if we were to specify that `cities.name` `REFERENCES` some other table, this constraint would not automatically propagate to `capitals`. In this case you could work around it by manually adding the same `REFERENCES` constraint to `capitals`.

- Specifying that another table's column `REFERENCES cities(name)` would allow the other table to contain city names, but not capital names. There is no good workaround for this case.

Some functionality not implemented for inheritance hierarchies is implemented for declarative partitioning. Considerable care is needed in deciding whether partitioning with legacy inheritance is useful for your application.

5.12. Table Partitioning

PostgreSQL supports basic table partitioning. This section describes why and how to implement partitioning as part of your database design.

5.12.1. Overview

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning effectively substitutes for the upper tree levels of indexes, making it more likely that the heavily-used parts of the indexes fit in memory.

- When queries or updates access a large percentage of a single partition, performance can be improved by using a sequential scan of that partition instead of using an index, which would require random-access reads scattered across the whole table.

- Bulk loads and deletes can be accomplished by adding or removing partitions, if the usage pattern is accounted for in the partitioning design. Dropping an individual partition using `DROP TABLE`, or doing `ALTER TABLE DETACH PARTITION`, is far faster than a bulk operation. These commands also entirely avoid the `VACUUM` overhead caused by a bulk `DELETE`.

- Seldom-used data can be migrated to cheaper and slower storage media.

These benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of the database server.

PostgreSQL offers built-in support for the following forms of partitioning:

Range Partitioning

> The table is partitioned into “ranges” defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example, one might partition by date ranges, or by ranges of identifiers for particular business objects. Each range's bounds are understood as being inclusive at the lower end and exclusive at the upper end. For example, if one partition's range is from `1` to `10`, and the next one's range is from `10` to `20`, then value `10` belongs to the second partition not the first.

List Partitioning

The table is partitioned by explicitly listing which key value(s) appear in each partition.

Hash Partitioning

The table is partitioned by specifying a modulus and a remainder for each partition. Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.

If your application needs to use other forms of partitioning not listed above, alternative methods such as inheritance and `UNION ALL` views can be used instead. Such methods offer flexibility but do not have some of the performance benefits of built-in declarative partitioning.

5.12.2. Declarative Partitioning

PostgreSQL allows you to declare that a table is divided into partitions. The table that is divided is referred to as a *partitioned table*. The declaration includes the *partitioning method* as described above, plus a list of columns or expressions to be used as the *partition key*.

The partitioned table itself is a "virtual" table having no storage of its own. Instead, the storage belongs to *partitions*, which are otherwise-ordinary tables associated with the partitioned table. Each partition stores a subset of the data as defined by its *partition bounds*. All rows inserted into a partitioned table will be routed to the appropriate one of the partitions based on the values of the partition key column(s). Updating the partition key of a row will cause it to be moved into a different partition if it no longer satisfies the partition bounds of its original partition.

Partitions may themselves be defined as partitioned tables, resulting in *sub-partitioning*. Although all partitions must have the same columns as their partitioned parent, partitions may have their own indexes, constraints and default values, distinct from those of other partitions. See CREATE TABLE for more details on creating partitioned tables and partitions.

It is not possible to turn a regular table into a partitioned table or vice versa. However, it is possible to add an existing regular or partitioned table as a partition of a partitioned table, or remove a partition from a partitioned table turning it into a standalone table; this can simplify and speed up many maintenance processes. See ALTER TABLE to learn more about the `ATTACH PARTITION` and `DETACH PARTITION` sub-commands.

Partitions can also be foreign tables, although considerable care is needed because it is then the user's responsibility that the contents of the foreign table satisfy the partitioning rule. There are some other restrictions as well. See CREATE FOREIGN TABLE for more information.

5.12.2.1. Example

Suppose we are constructing a database for a large ice cream company. The company measures peak temperatures every day as well as ice cream sales in each region. Conceptually, we want a table like:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

We know that most queries will access just the last week's, month's or quarter's data, since the main use of this table will be to prepare online reports for management. To reduce the amount of old data that needs to be stored, we decide to keep only the most recent 3 years worth of data. At the beginning

of each month we will remove the oldest month's data. In this situation we can use partitioning to help us meet all of our different requirements for the measurements table.

To use declarative partitioning in this case, use the following steps:

1. Create the `measurement` table as a partitioned table by specifying the `PARTITION BY` clause, which includes the partitioning method (`RANGE` in this case) and the list of column(s) to use as the partition key.

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

2. Create partitions. Each partition's definition must specify bounds that correspond to the partitioning method and partition key of the parent. Note that specifying bounds such that the new partition's values would overlap with those in one or more existing partitions will cause an error.

   Partitions thus created are in every way normal PostgreSQL tables (or, possibly, foreign tables). It is possible to specify a tablespace and storage parameters for each partition separately.

   For our example, each partition should hold one month's worth of data, to match the requirement of deleting one month's data at a time. So the commands might look like:

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

CREATE TABLE measurement_y2006m03 PARTITION OF measurement
    FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

...
CREATE TABLE measurement_y2007m11 PARTITION OF measurement
    FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement_y2007m12 PARTITION OF measurement
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

   (Recall that adjacent partitions can share a bound value, since range upper bounds are treated as exclusive bounds.)

   If you wish to implement sub-partitioning, again specify the `PARTITION BY` clause in the commands used to create individual partitions, for example:

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

   After creating partitions of `measurement_y2006m02`, any data inserted into `measurement` that is mapped to `measurement_y2006m02` (or data that is directly inserted into `measurement_y2006m02`, which is allowed provided its partition constraint is satisfied) will be further

redirected to one of its partitions based on the `peaktemp` column. The partition key specified may overlap with the parent's partition key, although care should be taken when specifying the bounds of a sub-partition such that the set of data it accepts constitutes a subset of what the partition's own bounds allow; the system does not try to check whether that's really the case.

Inserting data into the parent table that does not map to one of the existing partitions will cause an error; an appropriate partition must be added manually.

It is not necessary to manually create table constraints describing the partition boundary conditions for partitions. Such constraints will be created automatically.

3. Create an index on the key column(s), as well as any other indexes you might want, on the partitioned table. (The key index is not strictly necessary, but in most scenarios it is helpful.) This automatically creates a matching index on each partition, and any partitions you create or attach later will also have such an index. An index or unique constraint declared on a partitioned table is “virtual” in the same way that the partitioned table is: the actual data is in child indexes on the individual partition tables.

   ```
   CREATE INDEX ON measurement (logdate);
   ```

4. Ensure that the enable_partition_pruning configuration parameter is not disabled in `postgresql.conf`. If it is, queries will not be optimized as desired.

In the above example we would be creating a new partition each month, so it might be wise to write a script that generates the required DDL automatically.

5.12.2.2. Partition Maintenance

Normally the set of partitions established when initially defining the table is not intended to remain static. It is common to want to remove partitions holding old data and periodically add new partitions for new data. One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around.

The simplest option for removing old data is to drop the partition that is no longer necessary:

```
DROP TABLE measurement_y2006m02;
```

This can very quickly delete millions of records because it doesn't have to individually delete every record. Note however that the above command requires taking an `ACCESS EXCLUSIVE` lock on the parent table.

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right. This has two forms:

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02
 CONCURRENTLY;
```

These allow further operations to be performed on the data before it is dropped. For example, this is often a useful time to back up the data using `COPY`, pg_dump, or similar tools. It might also be a useful time to aggregate data into smaller formats, perform other data manipulations, or run reports. The first form of the command requires an `ACCESS EXCLUSIVE` lock on the parent table. Adding the `CONCURRENTLY` qualifier as in the second form allows the detach operation to require only `SHARE UPDATE EXCLUSIVE` lock on the parent table, but see `ALTER TABLE ... DETACH PARTITION` for details on the restrictions.

Similarly we can add a new partition to handle new data. We can create an empty partition in the partitioned table just as the original partitions were created above:

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

As an alternative to creating a new partition, it is sometimes more convenient to create a new table separate from the partition structure and attach it as a partition later. This allows new data to be loaded, checked, and transformed prior to it appearing in the partitioned table. Moreover, the `ATTACH PARTITION` operation requires only a `SHARE UPDATE EXCLUSIVE` lock on the partitioned table rather than the `ACCESS EXCLUSIVE` lock required by `CREATE TABLE ... PARTITION OF`, so it is more friendly to concurrent operations on the partitioned table; see `ALTER TABLE ... ATTACH PARTITION` for additional details. The `CREATE TABLE ... LIKE` option can be helpful to avoid tediously repeating the parent table's definition; for example:

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE
 '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

Note that when running the `ATTACH PARTITION` command, the table will be scanned to validate the partition constraint while holding an `ACCESS EXCLUSIVE` lock on that partition. As shown above, it is recommended to avoid this scan by creating a `CHECK` constraint matching the expected partition constraint on the table prior to attaching it. Once the `ATTACH PARTITION` is complete, it is recommended to drop the now-redundant `CHECK` constraint. If the table being attached is itself a partitioned table, then each of its sub-partitions will be recursively locked and scanned until either a suitable `CHECK` constraint is encountered or the leaf partitions are reached.

Similarly, if the partitioned table has a `DEFAULT` partition, it is recommended to create a `CHECK` constraint which excludes the to-be-attached partition's constraint. If this is not done, the `DEFAULT` partition will be scanned to verify that it contains no records which should be located in the partition being attached. This operation will be performed whilst holding an `ACCESS EXCLUSIVE` lock on the `DEFAULT` partition. If the `DEFAULT` partition is itself a partitioned table, then each of its partitions will be recursively checked in the same way as the table being attached, as mentioned above.

As mentioned earlier, it is possible to create indexes on partitioned tables so that they are applied automatically to the entire hierarchy. This can be very convenient as not only will all existing partitions be indexed, but any future partitions will be as well. However, one limitation when creating new indexes on partitioned tables is that it is not possible to use the `CONCURRENTLY` qualifier, which could lead to long lock times. To avoid this, you can use `CREATE INDEX ON ONLY` the partitioned table, which creates the new index marked as invalid, preventing automatic application to existing partitions. Instead, indexes can then be created individually on each partition using `CONCURRENTLY` and *attached* to the partitioned index on the parent using `ALTER INDEX ... ATTACH PARTITION`. Once indexes for all the partitions are attached to the parent index, the parent index will be marked valid automatically. Example:

```
CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX CONCURRENTLY measurement_usls_200602_idx
```

```
    ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
    ATTACH PARTITION measurement_usls_200602_idx;
...
```

This technique can be used with UNIQUE and PRIMARY KEY constraints too; the indexes are created implicitly when the constraint is created. Example:

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
    ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...
```

5.12.2.3. Limitations

The following limitations apply to partitioned tables:

- To create a unique or primary key constraint on a partitioned table, the partition keys must not include any expressions or function calls and the constraint's columns must include all of the partition key columns. This limitation exists because the individual indexes making up the constraint can only directly enforce uniqueness within their own partitions; therefore, the partition structure itself must guarantee that there are not duplicates in different partitions.

- Similarly an exclusion constraint must include all the partition key columns. Furthermore the constraint must compare those columns for equality (not e.g. &&). Again, this limitation stems from not being able to enforce cross-partition restrictions. The constraint may include additional columns that aren't part of the partition key, and it may compare those with any operators you like.

- BEFORE ROW triggers on INSERT cannot change which partition is the final destination for a new row.

- Mixing temporary and permanent relations in the same partition tree is not allowed. Hence, if the partitioned table is permanent, so must be its partitions and likewise if the partitioned table is temporary. When using temporary relations, all members of the partition tree have to be from the same session.

Individual partitions are linked to their partitioned table using inheritance behind-the-scenes. However, it is not possible to use all of the generic features of inheritance with declaratively partitioned tables or their partitions, as discussed below. Notably, a partition cannot have any parents other than the partitioned table it is a partition of, nor can a table inherit from both a partitioned table and a regular table. That means partitioned tables and their partitions never share an inheritance hierarchy with regular tables.

Since a partition hierarchy consisting of the partitioned table and its partitions is still an inheritance hierarchy, tableoid and all the normal rules of inheritance apply as described in Section 5.11, with a few exceptions:

- Partitions cannot have columns that are not present in the parent. It is not possible to specify columns when creating partitions with CREATE TABLE, nor is it possible to add columns to partitions after-the-fact using ALTER TABLE. Tables may be added as a partition with ALTER TABLE ... ATTACH PARTITION only if their columns exactly match the parent.

- Both CHECK and NOT NULL constraints of a partitioned table are always inherited by all its partitions. CHECK constraints that are marked NO INHERIT are not allowed to be created on partitioned tables. You cannot drop a NOT NULL constraint on a partition's column if the same constraint is present in the parent table.

- Using `ONLY` to add or drop a constraint on only the partitioned table is supported as long as there are no partitions. Once partitions exist, using `ONLY` will result in an error for any constraints other than `UNIQUE` and `PRIMARY KEY`. Instead, constraints on the partitions themselves can be added and (if they are not present in the parent table) dropped.
- As a partitioned table does not have any data itself, attempts to use `TRUNCATE ONLY` on a partitioned table will always return an error.

5.12.3. Partitioning Using Inheritance

While the built-in declarative partitioning is suitable for most common use cases, there are some circumstances where a more flexible approach may be useful. Partitioning can be implemented using table inheritance, which allows for several features not supported by declarative partitioning, such as:

- For declarative partitioning, partitions must have exactly the same set of columns as the partitioned table, whereas with table inheritance, child tables may have extra columns not present in the parent.
- Table inheritance allows for multiple inheritance.
- Declarative partitioning only supports range, list and hash partitioning, whereas table inheritance allows data to be divided in a manner of the user's choosing. (Note, however, that if constraint exclusion is unable to prune child tables effectively, query performance might be poor.)

5.12.3.1. Example

This example builds a partitioning structure equivalent to the declarative partitioning example above. Use the following steps:

1. Create the “root” table, from which all of the “child” tables will inherit. This table will contain no data. Do not define any check constraints on this table, unless you intend them to be applied equally to all child tables. There is no point in defining any indexes or unique constraints on it, either. For our example, the root table is the `measurement` table as originally defined:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

2. Create several “child” tables that each inherit from the root table. Normally, these tables will not add any columns to the set inherited from the root. Just as with declarative partitioning, these tables are in every way normal PostgreSQL tables (or foreign tables).

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. Add non-overlapping table constraints to the child tables to define the allowed key values in each.

   Typical examples would be:

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire',
 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

Ensure that the constraints guarantee that there is no overlap between the key values permitted in different child tables. A common mistake is to set up range constraints like:

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

This is wrong since it is not clear which child table the key value 200 belongs in. Instead, ranges should be defined in this style:

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE
 '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE
 '2006-04-01' )
) INHERITS (measurement);

...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE
 '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE
 '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE
 '2008-02-01' )
) INHERITS (measurement);
```

4. For each child table, create an index on the key column(s), as well as any other indexes you might want.

```
CREATE INDEX measurement_y2006m02_logdate ON
 measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON
 measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON
 measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON
 measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON
 measurement_y2008m01 (logdate);
```

5. We want our application to be able to say `INSERT INTO measurement ...` and have the data be redirected into the appropriate child table. We can arrange that by attaching a suitable trigger function to the root table. If data will be added only to the latest child, we can use a very simple trigger function:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
```

```
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

After creating the function, we create a trigger which calls the trigger function:

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
```

We must redefine the trigger function each month so that it always inserts into the current child table. The trigger definition does not need to be updated, however.

We might want to insert data and have the server automatically locate the child table into which the row should be added. We could do this with a more complex trigger function, for example:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the
 measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
```

```
LANGUAGE plpgsql;
```

The trigger definition is the same as before. Note that each `IF` test must exactly match the `CHECK` constraint for its child table.

While this function is more complex than the single-month case, it doesn't need to be updated as often, since branches can be added in advance of being needed.

> **Note**
>
> In practice, it might be best to check the newest child first, if most inserts go into that child. For simplicity, we have shown the trigger's tests in the same order as in other parts of this example.

A different approach to redirecting inserts into the appropriate child table is to set up rules, instead of a trigger, on the root table. For example:

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE
 '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE
 '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

A rule has significantly more overhead than a trigger, but the overhead is paid once per query rather than once per row, so this method might be advantageous for bulk-insert situations. In most cases, however, the trigger method will offer better performance.

Be aware that `COPY` ignores rules. If you want to use `COPY` to insert data, you'll need to copy into the correct child table rather than directly into the root. `COPY` does fire triggers, so you can use it normally if you use the trigger approach.

Another disadvantage of the rule approach is that there is no simple way to force an error if the set of rules doesn't cover the insertion date; the data will silently go into the root table instead.

6. Ensure that the constraint_exclusion configuration parameter is not disabled in `postgresql.conf`; otherwise child tables may be accessed unnecessarily.

As we can see, a complex table hierarchy could require a substantial amount of DDL. In the above example we would be creating a new child table each month, so it might be wise to write a script that generates the required DDL automatically.

5.12.3.2. Maintenance for Inheritance Partitioning

To remove old data quickly, simply drop the child table that is no longer necessary:

```
DROP TABLE measurement_y2006m02;
```

To remove the child table from the inheritance hierarchy table but retain access to it as a table in its own right:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

To add a new child table to handle new data, create an empty child table just as the original children were created above:

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE
 '2008-03-01' )
) INHERITS (measurement);
```

Alternatively, one may want to create and populate the new child table before adding it to the table hierarchy. This could allow data to be loaded, checked, and transformed before being made visible to queries on the parent table.

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE
 '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.12.3.3. Caveats

The following caveats apply to partitioning implemented using inheritance:

- There is no automatic way to verify that all of the `CHECK` constraints are mutually exclusive. It is safer to create code that generates child tables and creates and/or modifies associated objects than to write each by hand.
- Indexes and foreign key constraints apply to single tables and not to their inheritance children, hence they have some caveats to be aware of.
- The schemes shown here assume that the values of a row's key column(s) never change, or at least do not change enough to require it to move to another partition. An `UPDATE` that attempts to do that will fail because of the `CHECK` constraints. If you need to handle such cases, you can put suitable update triggers on the child tables, but it makes management of the structure much more complicated.
- If you are using manual `VACUUM` or `ANALYZE` commands, don't forget that you need to run them on each child table individually. A command like:

  ```
  ANALYZE measurement;
  ```

  will only process the root table.
- `INSERT` statements with `ON CONFLICT` clauses are unlikely to work as expected, as the `ON CONFLICT` action is only taken in case of unique violations on the specified target relation, not its child relations.
- Triggers or rules will be needed to route rows to the desired child table, unless the application is explicitly aware of the partitioning scheme. Triggers may be complicated to write, and will be much slower than the tuple routing performed internally by declarative partitioning.

5.12.4. Partition Pruning

Partition pruning is a query optimization technique that improves performance for declaratively partitioned tables. As an example:

```
SET enable_partition_pruning = on;                 -- the default
SELECT count(*) FROM measurement WHERE logdate >= DATE
 '2008-01-01';
```

Without partition pruning, the above query would scan each of the partitions of the `measurement` table. With partition pruning enabled, the planner will examine the definition of each partition and prove that the partition need not be scanned because it could not contain any rows meeting the query's `WHERE` clause. When the planner can prove this, it excludes (*prunes*) the partition from the query plan.

By using the EXPLAIN command and the enable_partition_pruning configuration parameter, it's possible to show the difference between a plan for which partitions have been pruned and one for which they have not. A typical unoptimized plan for this type of table setup is:

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE
 '2008-01-01';
                                     QUERY PLAN
-------------------------------------------------------------------------
---------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

Some or all of the partitions might use index scans instead of full-table sequential scans, but the point here is that there is no need to scan the older partitions at all to answer this query. When we enable partition pruning, we get a significantly cheaper plan that will deliver the same answer:

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE
 '2008-01-01';
                                     QUERY PLAN
-------------------------------------------------------------------------
---------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617
 width=0)
         Filter: (logdate >= '2008-01-01'::date)
```

Note that partition pruning is driven only by the constraints defined implicitly by the partition keys, not by the presence of indexes. Therefore it isn't necessary to define indexes on the key columns. Whether an index needs to be created for a given partition depends on whether you expect that queries that scan the partition will generally scan a large part of the partition or just a small part. An index will be helpful in the latter case but not the former.

Partition pruning can be performed not only during the planning of a given query, but also during its execution. This is useful as it can allow more partitions to be pruned when clauses contain expressions whose values are not known at query planning time, for example, parameters defined in a PREPARE statement, using a value obtained from a subquery, or using a parameterized value on the inner side of a nested loop join. Partition pruning during execution can be performed at any of the following times:

- During initialization of the query plan. Partition pruning can be performed here for parameter values which are known during the initialization phase of execution. Partitions which are pruned during this stage will not show up in the query's EXPLAIN or EXPLAIN ANALYZE. It is possible to determine the number of partitions which were removed during this phase by observing the “Subplans Removed” property in the EXPLAIN output.

- During actual execution of the query plan. Partition pruning may also be performed here to remove partitions using values which are only known during actual query execution. This includes values from subqueries and values from execution-time parameters such as those from parameterized nested loop joins. Since the value of these parameters may change many times during the execution of the query, partition pruning is performed whenever one of the execution parameters being used by partition pruning changes. Determining if partitions were pruned during this phase requires careful inspection of the loops property in the EXPLAIN ANALYZE output. Subplans corresponding to different partitions may have different values for it depending on how many times each of them was pruned during execution. Some may be shown as (never executed) if they were pruned every time.

Partition pruning can be disabled using the enable_partition_pruning setting.

5.12.5. Partitioning and Constraint Exclusion

Constraint exclusion is a query optimization technique similar to partition pruning. While it is primarily used for partitioning implemented using the legacy inheritance method, it can be used for other purposes, including with declarative partitioning.

Constraint exclusion works in a very similar way to partition pruning, except that it uses each table's CHECK constraints — which gives it its name — whereas partition pruning uses the table's partition bounds, which exist only in the case of declarative partitioning. Another difference is that constraint exclusion is only applied at plan time; there is no attempt to remove partitions at execution time.

The fact that constraint exclusion uses CHECK constraints, which makes it slow compared to partition pruning, can sometimes be used as an advantage: because constraints can be defined even on declaratively-partitioned tables, in addition to their internal partition bounds, constraint exclusion may be able to elide additional partitions from the query plan.

The default (and recommended) setting of constraint_exclusion is neither on nor off, but an intermediate setting called partition, which causes the technique to be applied only to queries that are likely to be working on inheritance partitioned tables. The on setting causes the planner to examine CHECK constraints in all queries, even simple ones that are unlikely to benefit.

The following caveats apply to constraint exclusion:

- Constraint exclusion is only applied during query planning, unlike partition pruning, which can also be applied during query execution.

- Constraint exclusion only works when the query's WHERE clause contains constants (or externally supplied parameters). For example, a comparison against a non-immutable function such as CUR-

`RENT_TIMESTAMP` cannot be optimized, since the planner cannot know which child table the function's value might fall into at run time.

- Keep the partitioning constraints simple, else the planner may not be able to prove that child tables might not need to be visited. Use simple equality conditions for list partitioning, or simple range tests for range partitioning, as illustrated in the preceding examples. A good rule of thumb is that partitioning constraints should contain only comparisons of the partitioning column(s) to constants using B-tree-indexable operators, because only B-tree-indexable column(s) are allowed in the partition key.

- All constraints on all children of the parent table are examined during constraint exclusion, so large numbers of children are likely to increase query planning time considerably. So the legacy inheritance based partitioning will work well with up to perhaps a hundred child tables; don't try to use many thousands of children.

5.12.6. Best Practices for Declarative Partitioning

The choice of how to partition a table should be made carefully, as the performance of query planning and execution can be negatively affected by poor design.

One of the most critical design decisions will be the column or columns by which you partition your data. Often the best choice will be to partition by the column or set of columns which most commonly appear in `WHERE` clauses of queries being executed on the partitioned table. `WHERE` clauses that are compatible with the partition bound constraints can be used to prune unneeded partitions. However, you may be forced into making other decisions by requirements for the `PRIMARY KEY` or a `UNIQUE` constraint. Removal of unwanted data is also a factor to consider when planning your partitioning strategy. An entire partition can be detached fairly quickly, so it may be beneficial to design the partition strategy in such a way that all data to be removed at once is located in a single partition.

Choosing the target number of partitions that the table should be divided into is also a critical decision to make. Not having enough partitions may mean that indexes remain too large and that data locality remains poor which could result in low cache hit ratios. However, dividing the table into too many partitions can also cause issues. Too many partitions can mean longer query planning times and higher memory consumption during both query planning and execution, as further described below. When choosing how to partition your table, it's also important to consider what changes may occur in the future. For example, if you choose to have one partition per customer and you currently have a small number of large customers, consider the implications if in several years you instead find yourself with a large number of small customers. In this case, it may be better to choose to partition by `HASH` and choose a reasonable number of partitions rather than trying to partition by `LIST` and hoping that the number of customers does not increase beyond what it is practical to partition the data by.

Sub-partitioning can be useful to further divide partitions that are expected to become larger than other partitions. Another option is to use range partitioning with multiple columns in the partition key. Either of these can easily lead to excessive numbers of partitions, so restraint is advisable.

It is important to consider the overhead of partitioning during query planning and execution. The query planner is generally able to handle partition hierarchies with up to a few thousand partitions fairly well, provided that typical queries allow the query planner to prune all but a small number of partitions. Planning times become longer and memory consumption becomes higher when more partitions remain after the planner performs partition pruning. Another reason to be concerned about having a large number of partitions is that the server's memory consumption may grow significantly over time, especially if many sessions touch large numbers of partitions. That's because each partition requires its metadata to be loaded into the local memory of each session that touches it.

With data warehouse type workloads, it can make sense to use a larger number of partitions than with an OLTP type workload. Generally, in data warehouses, query planning time is less of a concern as the majority of processing time is spent during query execution. With either of these two types of workload, it is important to make the right decisions early, as re-partitioning large quantities of data can be painfully slow. Simulations of the intended workload are often beneficial for optimizing the

partitioning strategy. Never just assume that more partitions are better than fewer partitions, nor vice-versa.

5.13. Foreign Data

PostgreSQL implements portions of the SQL/MED specification, allowing you to access data that resides outside PostgreSQL using regular SQL queries. Such data is referred to as *foreign data*. (Note that this usage is not to be confused with foreign keys, which are a type of constraint within the database.)

Foreign data is accessed with help from a *foreign data wrapper*. A foreign data wrapper is a library that can communicate with an external data source, hiding the details of connecting to the data source and obtaining data from it. There are some foreign data wrappers available as `contrib` modules; see Appendix F. Other kinds of foreign data wrappers might be found as third party products. If none of the existing foreign data wrappers suit your needs, you can write your own; see Chapter 57.

To access foreign data, you need to create a *foreign server* object, which defines how to connect to a particular external data source according to the set of options used by its supporting foreign data wrapper. Then you need to create one or more *foreign tables*, which define the structure of the remote data. A foreign table can be used in queries just like a normal table, but a foreign table has no storage in the PostgreSQL server. Whenever it is used, PostgreSQL asks the foreign data wrapper to fetch data from the external source, or transmit data to the external source in the case of update commands.

Accessing remote data may require authenticating to the external data source. This information can be provided by a *user mapping*, which can provide additional data such as user names and passwords based on the current PostgreSQL role.

For additional information, see CREATE FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE, and IMPORT FOREIGN SCHEMA.

5.14. Other Database Objects

Tables are the central objects in a relational database structure, because they hold your data. But they are not the only objects that exist in a database. Many other kinds of objects can be created to make the use and management of the data more efficient or convenient. They are not discussed in this chapter, but we give you a list here so that you are aware of what is possible:

- Views
- Functions, procedures, and operators
- Data types and domains
- Triggers and rewrite rules

Detailed information on these topics appears in Part V.

5.15. Dependency Tracking

When you create complex database structures involving many tables with foreign key constraints, views, triggers, functions, etc. you implicitly create a net of dependencies between the objects. For instance, a table with a foreign key constraint depends on the table it references.

To ensure the integrity of the entire database structure, PostgreSQL makes sure that you cannot drop objects that other objects still depend on. For example, attempting to drop the products table we considered in Section 5.5.5, with the orders table depending on it, would result in an error message like this:

```
DROP TABLE products;

ERROR:  cannot drop table products because other objects depend on
 it
DETAIL:  constraint orders_product_no_fkey on table orders depends
 on table products
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

The error message contains a useful hint: if you do not want to bother deleting all the dependent objects individually, you can run:

```
DROP TABLE products CASCADE;
```

and all the dependent objects will be removed, as will any objects that depend on them, recursively. In this case, it doesn't remove the orders table, it only removes the foreign key constraint. It stops there because nothing depends on the foreign key constraint. (If you want to check what `DROP ... CASCADE` will do, run `DROP` without `CASCADE` and read the `DETAIL` output.)

Almost all `DROP` commands in PostgreSQL support specifying `CASCADE`. Of course, the nature of the possible dependencies varies with the type of the object. You can also write `RESTRICT` instead of `CASCADE` to get the default behavior, which is to prevent dropping objects that any other objects depend on.

> **Note**
>
> According to the SQL standard, specifying either `RESTRICT` or `CASCADE` is required in a `DROP` command. No database system actually enforces that rule, but whether the default behavior is `RESTRICT` or `CASCADE` varies across systems.

If a `DROP` command lists multiple objects, `CASCADE` is only required when there are dependencies outside the specified group. For example, when saying `DROP TABLE tab1, tab2` the existence of a foreign key referencing `tab1` from `tab2` would not mean that `CASCADE` is needed to succeed.

For a user-defined function or procedure whose body is defined as a string literal, PostgreSQL tracks dependencies associated with the function's externally-visible properties, such as its argument and result types, but *not* dependencies that could only be known by examining the function body. As an example, consider this situation:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

(See Section 36.5 for an explanation of SQL-language functions.) PostgreSQL will be aware that the `get_color_note` function depends on the `rainbow` type: dropping the type would force dropping the function, because its argument type would no longer be defined. But PostgreSQL will not consider `get_color_note` to depend on the `my_colors` table, and so will not drop the function if the table is dropped. While there are disadvantages to this approach, there are also benefits. The function is still valid in some sense if the table is missing, though executing it would cause an error; creating a new table of the same name would allow the function to work again.

On the other hand, for an SQL-language function or procedure whose body is written in SQL-standard style, the body is parsed at function definition time and all dependencies recognized by the parser are stored. Thus, if we write the function above as

```
CREATE FUNCTION get_color_note (rainbow) RETURNS text
BEGIN ATOMIC
  SELECT note FROM my_colors WHERE color = $1;
END;
```

then the function's dependency on the `my_colors` table will be known and enforced by `DROP`.

Chapter 6. Data Manipulation

The previous chapter discussed how to create tables and other structures to hold your data. Now it is time to fill the tables with data. This chapter covers how to insert, update, and delete table data. The chapter after this will finally explain how to extract your long-lost data from the database.

6.1. Inserting Data

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is inserted one row at a time. You can also insert more than one row in a single command, but it is not possible to insert something that is not a complete row. Even if you know only some column values, a complete row must be created.

To create a new row, use the INSERT command. The command requires the table name and column values. For example, consider the products table from Chapter 5:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

An example command to insert a row would be:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns appear in the table, separated by commas. Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid this you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese',
 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese',
 9.99, 1);
```

Many users consider it good practice to always list the column names.

If you don't have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

The second form is a PostgreSQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese',
 DEFAULT);
```

```
INSERT INTO products DEFAULT VALUES;
```

You can insert multiple rows in a single command:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

It is also possible to insert the result of a query (which might be no rows, one row, or many rows):

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

This provides the full power of the SQL query mechanism (Chapter 7) for computing the rows to be inserted.

> **Tip**
>
> When inserting a lot of data at the same time, consider using the COPY command. It is not as flexible as the INSERT command, but is more efficient. Refer to Section 14.4 for more information on improving bulk loading performance.

6.2. Updating Data

The modification of data that is already in the database is referred to as updating. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately; the other columns are not affected.

To update existing rows, use the UPDATE command. This requires three pieces of information:

1. The name of the table and column to update
2. The new value of the column
3. Which row(s) to update

Recall from Chapter 5 that SQL does not, in general, provide a unique identifier for rows. Therefore it is not always possible to directly specify which row to update. Instead, you specify which conditions a row must meet in order to be updated. Only if you have a primary key in the table (independent of whether you declared it or not) can you reliably address individual rows by choosing a condition that matches the primary key. Graphical database access tools rely on this fact to allow you to update rows individually.

For example, this command updates all products that have a price of 5 to have a price of 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

This might cause zero, one, or many rows to be updated. It is not an error to attempt an update that does not match any rows.

Let's look at that command in detail. First is the key word `UPDATE` followed by the table name. As usual, the table name can be schema-qualified, otherwise it is looked up in the path. Next is the key word `SET` followed by the column name, an equal sign, and the new column value. The new column value can be any scalar expression, not just a constant. For example, if you want to raise the price of all products by 10% you could use:

```
UPDATE products SET price = price * 1.10;
```

As you see, the expression for the new value can refer to the existing value(s) in the row. We also left out the WHERE clause. If it is omitted, it means that all rows in the table are updated. If it is present, only those rows that match the WHERE condition are updated. Note that the equals sign in the SET clause is an assignment while the one in the WHERE clause is a comparison, but this does not create any ambiguity. Of course, the WHERE condition does not have to be an equality test. Many other operators are available (see Chapter 9). But the expression needs to evaluate to a Boolean result.

You can update more than one column in an UPDATE command by listing more than one assignment in the SET clause. For example:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Deleting Data

So far we have explained how to add data to tables and how to change data. What remains is to discuss how to remove data that is no longer needed. Just as adding data is only possible in whole rows, you can only remove entire rows from a table. In the previous section we explained that SQL does not provide a way to directly address individual rows. Therefore, removing rows can only be done by specifying conditions that the rows to be removed have to match. If you have a primary key in the table then you can specify the exact row. But you can also remove groups of rows matching a condition, or you can remove all rows in the table at once.

You use the DELETE command to remove rows; the syntax is very similar to the UPDATE command. For instance, to remove all rows from the products table that have a price of 10, use:

```
DELETE FROM products WHERE price = 10;
```

If you simply write:

```
DELETE FROM products;
```

then all rows in the table will be deleted! Caveat programmer.

6.4. Returning Data from Modified Rows

Sometimes it is useful to obtain data from modified rows while they are being manipulated. The INSERT, UPDATE, DELETE, and MERGE commands all have an optional RETURNING clause that supports this. Use of RETURNING avoids performing an extra database query to collect the data, and is especially valuable when it would otherwise be difficult to identify the modified rows reliably.

The allowed contents of a RETURNING clause are the same as a SELECT command's output list (see Section 7.3). It can contain column names of the command's target table, or value expressions using those columns. A common shorthand is RETURNING *, which selects all columns of the target table in order.

In an INSERT, the data available to RETURNING is the row as it was inserted. This is not so useful in trivial inserts, since it would just repeat the data provided by the client. But it can be very handy when relying on computed default values. For example, when using a serial column to provide unique identifiers, RETURNING can return the ID assigned to a new row:

```
CREATE TABLE users (firstname text, lastname text, id serial
 primary key);
```

```
INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool')
 RETURNING id;
```

The RETURNING clause is also very useful with INSERT ... SELECT.

In an UPDATE, the data available to RETURNING is the new content of the modified row. For example:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

In a DELETE, the data available to RETURNING is the content of the deleted row. For example:

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

In a MERGE, the data available to RETURNING is the content of the source row plus the content of the inserted, updated, or deleted target row. Since it is quite common for the source and target to have many of the same columns, specifying RETURNING * can lead to a lot of duplicated columns, so it is often more useful to qualify it so as to return just the source or target row. For example:

```
MERGE INTO products p USING new_products n ON p.product_no =
 n.product_no
  WHEN NOT MATCHED THEN INSERT VALUES (n.product_no, n.name,
 n.price)
  WHEN MATCHED THEN UPDATE SET name = n.name, price = n.price
  RETURNING p.*;
```

If there are triggers (Chapter 37) on the target table, the data available to RETURNING is the row as modified by the triggers. Thus, inspecting columns computed by triggers is another common use-case for RETURNING.

Chapter 7. Queries

The previous chapters explained how to create tables, how to fill them with data, and how to manipulate that data. Now we finally discuss how to retrieve the data from the database.

7.1. Overview

The process of retrieving or the command to retrieve data from a database is called a *query*. In SQL the `SELECT` command is used to specify queries. The general syntax of the `SELECT` command is

```
[WITH with_queries] SELECT select_list FROM table_expression
 [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification. `WITH` queries are treated last since they are an advanced feature.

A simple kind of query has the form:

```
SELECT * FROM table1;
```

Assuming that there is a table called `table1`, this command would retrieve all rows and all user-defined columns from `table1`. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, while client libraries will offer functions to extract individual values from the query result.) The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if `table1` has columns named `a`, `b`, and `c` (and perhaps others) you can make the following query:

```
SELECT a, b + c FROM table1;
```

(assuming that `b` and `c` are of a numerical data type). See Section 7.3 for more details.

`FROM table1` is a simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the `SELECT` command as a calculator:

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

```
SELECT random();
```

7.2. Table Expressions

A *table expression* computes a table. The table expression contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses. Trivial table expressions simply refer to a table on disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional `WHERE`, `GROUP BY`, and `HAVING` clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the `FROM` clause. All these transforma-

tions produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

7.2.1. The FROM Clause

The FROM clause derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference can be a table name (possibly schema-qualified), or a derived table such as a subquery, a JOIN construct, or complex combinations of these. If more than one table reference is listed in the FROM clause, the tables are cross-joined (that is, the Cartesian product of their rows is formed; see below). The result of the FROM list is an intermediate virtual table that can then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

When a table reference names a table that is the parent of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its descendant tables, unless the key word ONLY precedes the table name. However, the reference produces only the columns that appear in the named table — any columns added in subtables are ignored.

Instead of writing ONLY before the table name, you can write * after the table name to explicitly specify that descendant tables are included. There is no real reason to use this syntax any more, because searching descendant tables is now always the default behavior. However, it is supported for compatibility with older releases.

7.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available. The general syntax of a joined table is

```
T1 join_type T2 [ join_condition ]
```

Joins of all types can be chained together, or nested: either or both *T1* and *T2* can be joined tables. Parentheses can be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

Join Types

Cross join

```
T1 CROSS JOIN T2
```

For every possible combination of rows from *T1* and *T2* (i.e., a Cartesian product), the joined table will contain a row consisting of all columns in *T1* followed by all columns in *T2*. If the tables have N and M rows respectively, the joined table will have N * M rows.

FROM *T1* CROSS JOIN *T2* is equivalent to FROM *T1* INNER JOIN *T2* ON TRUE (see below). It is also equivalent to FROM *T1*, *T2*.

> **Note**
>
> This latter equivalence does not hold exactly when more than two tables appear, because JOIN binds more tightly than comma. For example FROM *T1* CROSS JOIN *T2* INNER JOIN *T3* ON *condition* is not the same as FROM *T1*, *T2* INNER JOIN

> *T3* ON *condition* because the *condition* can reference *T1* in the first case but not the second.

Qualified joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING
 ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

The words INNER and OUTER are optional in all forms. INNER is the default; LEFT, RIGHT, and FULL imply an outer join.

The *join condition* is specified in the ON or USING clause, or implicitly by the word NATURAL. The join condition determines which rows from the two source tables are considered to "match", as explained in detail below.

The possible types of qualified join are:

INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

LEFT OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.

RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will always have a row for each row in T2.

FULL OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from *T1* and *T2* match if the ON expression evaluates to true.

The USING clause is a shorthand that allows you to take advantage of the specific situation where both sides of the join use the same name for the joining column(s). It takes a comma-separated list of the shared column names and forms a join condition that includes an equality comparison for each one. For example, joining *T1* and *T2* with USING (a, b) produces the join condition ON *T1*.a = *T2*.a AND *T1*.b = *T2*.b.

Furthermore, the output of JOIN USING suppresses redundant columns: there is no need to print both of the matched columns, since they must have equal values. While JOIN ON produces all columns from *T1* followed by all columns from *T2*, JOIN USING produces one output column for each of the listed column pairs (in the listed order), followed by any remaining columns from *T1*, followed by any remaining columns from *T2*.

Finally, NATURAL is a shorthand form of USING: it forms a USING list consisting of all column names that appear in both input tables. As with USING, these columns appear only once in the output table. If there are no common column names, NATURAL JOIN behaves like CROSS JOIN.

> **Note**
>
> USING is reasonably safe from column changes in the joined relations since only the listed columns are combined. NATURAL is considerably more risky since any schema changes to either relation that cause a new matching column name to be present will cause the join to combine that new column as well.

To put this together, assume we have tables t1:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

and t2:

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

then we get the following results for the various joins:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
```

```
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

The join condition specified with `ON` can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value =
 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Notice that placing the restriction in the `WHERE` clause produces a different result:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value
 = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

This is because a restriction placed in the `ON` clause is processed *before* the join, while a restriction placed in the `WHERE` clause is processed *after* the join. That does not matter with inner joins, but it matters a lot with outer joins.

7.2.1.2. Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a *table alias*.

To create a table alias, write

```
FROM table_reference AS alias
```

or

```
FROM table_reference alias
```

The `AS` key word is optional noise. *`alias`* can be any identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

```
SELECT * FROM some_very_long_table_name s JOIN
 another_fairly_long_name a ON s.id = a.num;
```

The alias becomes the new name of the table reference so far as the current query is concerned — it is not allowed to refer to the table by the original name elsewhere in the query. Thus, this is not valid:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- wrong
```

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.:

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id =
 child.mother_id;
```

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias `b` to the second instance of `my_table`, but the second statement assigns the alias to the result of the join:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Another form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, the alias hides the original name(s) within the JOIN. For example:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid; the table alias a is not visible outside the alias c.

7.2.1.3. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses. They may be assigned a table alias name, and optionally column alias names (as in Section 7.2.1.2). For example:

```
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to FROM table1 AS alias_name. More interesting cases, which cannot be reduced to a plain join, arise when the subquery involves grouping or aggregation.

A subquery can also be a VALUES list:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

Again, a table alias is optional. Assigning alias names to the columns of the VALUES list is optional, but is good practice. For more information see Section 7.7.

According to the SQL standard, a table alias name must be supplied for a subquery. PostgreSQL allows AS and the alias to be omitted, but writing one is good practice in SQL code that might be ported to another system.

7.2.1.4. Table Functions

Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view, or subquery in the FROM clause of a query. Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as columns of a table, view, or subquery.

Table functions may also be combined using the ROWS FROM syntax, with the results returned in parallel columns; the number of result rows in this case is that of the largest function result, with smaller results padded with null values to match.

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias
 [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY]
 [[AS] table_alias [(column_alias [, ... ])]]
```

If the WITH ORDINALITY clause is specified, an additional column of type bigint will be added to the function result columns. This column numbers the rows of the function result set, starting from 1. (This is a generalization of the SQL-standard syntax for UNNEST ... WITH ORDINALITY.)

By default, the ordinal column is called `ordinality`, but a different column name can be assigned to it using an `AS` clause.

The special table function `UNNEST` may be called with any number of array parameters, and it returns a corresponding number of columns, as if `UNNEST` (Section 9.19) had been called on each parameter separately and combined using the `ROWS FROM` construct.

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY]
 [[AS] table_alias [(column_alias [, ... ])]]
```

If no *`table_alias`* is specified, the function name is used as the table name; in the case of a `ROWS FROM()` construct, the first function's name is used.

If column aliases are not supplied, then for a function returning a base data type, the column name is also the same as the function name. For a function returning a composite type, the result columns get the names of the individual attributes of the type.

Some examples:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

In some cases it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudo-type `record` with no `OUT` parameters. When such a function is used in a query, the expected row structure must be specified in the query itself, so that the system can know how to parse and plan the query. This syntax looks like:

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ])
 [, ... ] )
```

When not using the `ROWS FROM()` syntax, the *`column_definition`* list replaces the column alias list that could otherwise be attached to the `FROM` item; the names in the column definitions serve as column aliases. When using the `ROWS FROM()` syntax, a *`column_definition`* list can be attached to each member function separately; or if there is only one member function and no `WITH ORDINALITY` clause, a *`column_definition`* list can be written in place of a column alias list following `ROWS FROM()`.

Consider this example:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM
 pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

The dblink function (part of the dblink module) executes a remote query. It is declared to return `record` since it might be used for any kind of query. The actual column set must be specified in the calling query so that the parser knows, for example, what `*` should expand to.

This example uses `ROWS FROM`:

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},
{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

It joins two functions into a single `FROM` target. `json_to_recordset()` is instructed to return two columns, the first `integer` and the second `text`. The result of `generate_series()` is used directly. The `ORDER BY` clause sorts the column values as integers.

7.2.1.5. LATERAL Subqueries

Subqueries appearing in `FROM` can be preceded by the key word `LATERAL`. This allows them to reference columns provided by preceding `FROM` items. (Without `LATERAL`, each subquery is evaluated independently and so cannot cross-reference any other `FROM` item.)

Table functions appearing in `FROM` can also be preceded by the key word `LATERAL`, but for functions the key word is optional; the function's arguments can contain references to columns provided by preceding `FROM` items in any case.

A `LATERAL` item can appear at the top level in the `FROM` list, or within a `JOIN` tree. In the latter case it can also refer to any items that are on the left-hand side of a `JOIN` that it is on the right-hand side of.

When a `FROM` item contains `LATERAL` cross-references, evaluation proceeds as follows: for each row of the `FROM` item providing the cross-referenced column(s), or set of rows of multiple `FROM` items providing the columns, the `LATERAL` item is evaluated using that row or row set's values of the columns. The resulting row(s) are joined as usual with the rows they were computed from. This is repeated for each row or set of rows from the column source table(s).

A trivial example of `LATERAL` is

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id =
 foo.bar_id) ss;
```

This is not especially useful since it has exactly the same result as the more conventional

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

LATERAL is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined. A common application is providing an argument value for a set-returning function. For example, supposing that vertices(polygon) returns the set of vertices of a polygon, we could identify close-together vertices of polygons stored in a table with:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

This query could also be written

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

or in several other equivalent formulations. (As already mentioned, the LATERAL key word is unnecessary in this example, but we use it for clarity.)

It is often particularly handy to LEFT JOIN to a LATERAL subquery, so that source rows will appear in the result even if the LATERAL subquery produces no rows for them. For example, if get_product_names() returns the names of products made by a manufacturer, but some manufacturers in our table currently produce no products, we could find out which ones those are like this:

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id)
 pname ON true
WHERE pname IS NULL;
```

7.2.2. The WHERE Clause

The syntax of the WHERE clause is

```
WHERE search_condition
```

where *search_condition* is any value expression (see Section 4.2) that returns a value of type boolean.

After the processing of the FROM clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (i.e., if the result is false or null) it is discarded. The search condition typically references at least one column of the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

> **Note**
>
> The join condition of an inner join can be written either in the WHERE clause or in the JOIN clause. For example, these table expressions are equivalent:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

and:

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

or perhaps even:

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Which one of these you use is mainly a matter of style. The `JOIN` syntax in the `FROM` clause is probably not as portable to other SQL database management systems, even though it is in the SQL standard. For outer joins there is no choice: they must be done in the `FROM` clause. The `ON` or `USING` clause of an outer join is *not* equivalent to a `WHERE` condition, because it results in the addition of rows (for unmatched input rows) as well as the removal of rows in the final result.

Here are some examples of `WHERE` clauses:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 =
 fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 =
 fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 >
 fdt.c1)
```

`fdt` is the table derived in the `FROM` clause. Rows that do not meet the search condition of the `WHERE` clause are eliminated from `fdt`. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how `fdt` is referenced in the subqueries. Qualifying `c1` as `fdt.c1` is only necessary if `c1` is also the name of a column in the derived input table of the subquery. But qualifying the column name adds clarity even when it is not needed. This example shows how the column naming scope of an outer query extends into its inner queries.

7.2.3. The `GROUP BY` and `HAVING` Clauses

After passing the `WHERE` filter, the derived input table might be subject to grouping, using the `GROUP BY` clause, and elimination of group rows using the `HAVING` clause.

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference
 [, grouping_column_reference]...
```

The `GROUP BY` clause is used to group together those rows in a table that have the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to

combine each set of rows having common values into one group row that represents all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups. For instance:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

In the second query, we could not have written `SELECT * FROM test1 GROUP BY x`, because there is no single value for the column `y` that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a single value in each group.

In general, if a table is grouped, columns that are not listed in `GROUP BY` cannot be referenced except in aggregate expressions. An example with aggregate expressions is:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Here `sum` is an aggregate function that computes a single value over the entire group. More information about the available aggregate functions can be found in Section 9.21.

> **Tip**
>
> Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the `DISTINCT` clause (see Section 7.3.3).

Here is another example: it calculates the total sales for each product (rather than the total sales of all products):

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

In this example, the columns `product_id`, `p.name`, and `p.price` must be in the `GROUP BY` clause since they are referenced in the query select list (but see below). The column `s.units` does not have to be in the `GROUP BY` list since it is only used in an aggregate expression (`sum(...)`),

which represents the sales of a product. For each product, the query returns a summary row about all sales of the product.

If the products table is set up so that, say, `product_id` is the primary key, then it would be enough to group by `product_id` in the above example, since name and price would be *functionally dependent* on the product ID, and so there would be no ambiguity about which name and price value to return for each product ID group.

In strict SQL, `GROUP BY` can only group by columns of the source table but PostgreSQL extends this to also allow `GROUP BY` to group by columns in the select list. Grouping by value expressions instead of simple column names is also allowed.

If a table has been grouped using `GROUP BY`, but only certain groups are of interest, the `HAVING` clause can be used, much like a `WHERE` clause, to eliminate groups from the result. The syntax is:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ...
 HAVING boolean_expression
```

Expressions in the `HAVING` clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Example:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

Again, a more realistic example:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS
 profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

In the example above, the `WHERE` clause is selecting rows by a column that is not grouped (the expression is only true for sales during the last four weeks), while the `HAVING` clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.

If a query contains aggregate function calls, but no `GROUP BY` clause, grouping still occurs: the result is a single group row (or perhaps no rows at all, if the single row is then eliminated by `HAVING`). The same is true if it contains a `HAVING` clause, even without any aggregate function calls or `GROUP BY` clause.

7.2.4. GROUPING SETS, CUBE, and ROLLUP

More complex grouping operations than those described above are possible using the concept of *grouping sets*. The data selected by the FROM and WHERE clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple GROUP BY clauses, and then the results returned. For example:

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING
 SETS ((brand), (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause. An empty grouping set means that all rows are aggregated down to a single group (which is output even if no input rows were present), as described above for the case of aggregate functions with no GROUP BY clause.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear. To distinguish which grouping a particular output row resulted from, see Table 9.64.

A shorthand notation is provided for specifying two common types of grouping set. A clause of the form

```
ROLLUP ( e1, e2, e3, ... )
```

represents the given list of expressions and all prefixes of the list including the empty list; thus it is equivalent to

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

This is commonly used for analysis over hierarchical data; e.g., total salary by department, division, and company-wide total.

A clause of the form

```
CUBE ( e1, e2, ... )
```

represents the given list and all of its possible subsets (i.e., the power set). Thus

```
CUBE ( a, b, c )
```

is equivalent to

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
)
```

The individual elements of a CUBE or ROLLUP clause may be either individual expressions, or sublists of elements in parentheses. In the latter case, the sublists are treated as single units for the purposes of generating the individual grouping sets. For example:

```
CUBE ( (a, b), (c, d) )
```

is equivalent to

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

and

```
ROLLUP ( a, (b, c), d )
```

is equivalent to

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

The CUBE and ROLLUP constructs can be used either directly in the GROUP BY clause, or nested inside a GROUPING SETS clause. If one GROUPING SETS clause is nested inside another, the effect is the same as if all the elements of the inner clause had been written directly in the outer clause.

If multiple grouping items are specified in a single GROUP BY clause, then the final list of grouping sets is the Cartesian product of the individual items. For example:

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

is equivalent to

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

When specifying multiple grouping items together, the final set of grouping sets might contain duplicates. For example:

```
GROUP BY ROLLUP (a, b), ROLLUP (a, c)
```

is equivalent to

```
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, b),
    (a, c),
    (a),
    (a),
    (a, c),
    (a),
    ()
)
```

If these duplicates are undesirable, they can be removed using the `DISTINCT` clause directly on the `GROUP BY`. Therefore:

```
GROUP BY DISTINCT ROLLUP (a, b), ROLLUP (a, c)
```

is equivalent to

```
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, c),
    (a),
    ()
)
```

This is not the same as using `SELECT DISTINCT` because the output rows may still contain duplicates. If any of the ungrouped columns contains NULL, it will be indistinguishable from the NULL used when that same column is grouped.

> **Note**
>
> The construct `(a, b)` is normally recognized in expressions as a row constructor. Within the `GROUP BY` clause, this does not apply at the top levels of expressions, and `(a, b)` is parsed

as a list of expressions as described above. If for some reason you *need* a row constructor in a grouping expression, use `ROW(a, b)`.

7.2.5. Window Function Processing

If the query contains any window functions (see Section 3.5, Section 9.22 and Section 4.2.8), these functions are evaluated after any grouping, aggregation, and `HAVING` filtering is performed. That is, if the query uses any aggregates, `GROUP BY`, or `HAVING`, then the rows seen by the window functions are the group rows instead of the original table rows from `FROM`/`WHERE`.

When multiple window functions are used, all the window functions having syntactically equivalent `PARTITION BY` and `ORDER BY` clauses in their window definitions are guaranteed to be evaluated in a single pass over the data. Therefore they will see the same sort ordering, even if the `ORDER BY` does not uniquely determine an ordering. However, no guarantees are made about the evaluation of functions having different `PARTITION BY` or `ORDER BY` specifications. (In such cases a sort step is typically required between the passes of window function evaluations, and the sort is not guaranteed to preserve ordering of rows that its `ORDER BY` sees as equivalent.)

Currently, window functions always require presorted data, and so the query output will be ordered according to one or another of the window functions' `PARTITION BY`/`ORDER BY` clauses. It is not recommended to rely on this, however. Use an explicit top-level `ORDER BY` clause if you want to be sure the results are sorted in a particular way.

7.3. Select Lists

As shown in the previous section, the table expression in the `SELECT` command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which *columns* of the intermediate table are actually output.

7.3.1. Select-List Items

The simplest kind of select list is `*` which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 4.2). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...
```

The columns names `a`, `b`, and `c` are either the actual names of the columns of tables referenced in the `FROM` clause, or the aliases given to them as explained in Section 7.2.1.2. The name space available in the select list is the same as in the `WHERE` clause, unless grouping is used, in which case it is the same as in the `HAVING` clause.

If more than one table has a column of the same name, the table name must also be given, as in:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

```
SELECT tbl1.*, tbl2.a FROM ...
```

See Section 8.16.5 for more about the *`table_name`*`.*` notation.

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row's values

substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they can be constant arithmetic expressions, for instance.

7.3.2. Column Labels

The entries in the select list can be assigned names for subsequent processing, such as for use in an ORDER BY clause or for display by the client application. For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified using AS, the system assigns a default column name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

The AS key word is usually optional, but in some cases where the desired column name matches a PostgreSQL key word, you must write AS or double-quote the column name in order to avoid ambiguity. (Appendix C shows which key words require AS to be used as a column label.) For example, FROM is one such key word, so this does not work:

```
SELECT a from, b + c AS sum FROM ...
```

but either of these do:

```
SELECT a AS from, b + c AS sum FROM ...
SELECT a "from", b + c AS sum FROM ...
```

For greatest safety against possible future key word additions, it is recommended that you always either write AS or double-quote the output column name.

> **Note**
>
> The naming of output columns here is different from that done in the FROM clause (see Section 7.2.1.2). It is possible to rename the same column twice, but the name assigned in the select list is the one that will be passed on.

7.3.3. DISTINCT

After the select list has been processed, the result table can optionally be subject to the elimination of duplicate rows. The DISTINCT key word is written directly after SELECT to specify this:

```
SELECT DISTINCT select_list ...
```

(Instead of DISTINCT the key word ALL can be used to specify the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Here *expression* is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the "first row" of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ON processing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subqueries in FROM, this construct can be avoided, but it is often the most convenient alternative.

7.4. Combining Queries (UNION, INTERSECT, EXCEPT)

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

where *query1* and *query2* are queries that can use any of the features discussed up to this point.

UNION effectively appends the result of *query2* to the result of *query1* (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as DISTINCT, unless UNION ALL is used.

INTERSECT returns all rows that are both in the result of *query1* and in the result of *query2*. Duplicate rows are eliminated unless INTERSECT ALL is used.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. (This is sometimes called the *difference* between two queries.) Again, duplicates are eliminated unless EXCEPT ALL is used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be "union compatible", which means that they return the same number of columns and the corresponding columns have compatible data types, as described in Section 10.5.

Set operations can be combined, for example

```
query1 UNION query2 EXCEPT query3
```

which is equivalent to

```
(query1 UNION query2) EXCEPT query3
```

As shown here, you can use parentheses to control the order of evaluation. Without parentheses, UNION and EXCEPT associate left-to-right, but INTERSECT binds more tightly than those two operators. Thus

```
query1 UNION query2 INTERSECT query3
```

means

```
query1 UNION (query2 INTERSECT query3)
```

You can also surround an individual *query* with parentheses. This is important if the *query* needs to use any of the clauses discussed in following sections, such as LIMIT. Without parentheses, you'll get a syntax error, or else the clause will be understood as applying to the output of the set operation rather than one of its inputs. For example,

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

is accepted, but it means

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

not

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

7.5. Sorting Rows (ORDER BY)

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The ORDER BY clause specifies the sort order:

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, sort_expression2 [ASC | DESC] [NULLS { FIRST |
 LAST }] ...]
```

The sort expression(s) can be any expression that would be valid in the query's select list. An example is:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression can be followed by an optional ASC or DESC keyword to set the sort direction to ascending or descending. ASC order is the default. Ascending order puts smaller values first, where "smaller" is defined in terms of the < operator. Similarly, descending order is determined with the > operator. [1]

The NULLS FIRST and NULLS LAST options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if larger than any non-null value; that is, NULLS FIRST is the default for DESC order, and NULLS LAST otherwise.

Note that the ordering options are considered independently for each sort column. For example ORDER BY x, y DESC means ORDER BY x ASC, y DESC, which is not the same as ORDER BY x DESC, y DESC.

[1] Actually, PostgreSQL uses the *default B-tree operator class* for the expression's data type to determine the sort ordering for ASC and DESC. Conventionally, data types will be set up so that the < and > operators correspond to this sort ordering, but a user-defined data type's designer could choose to do something different.

A *sort_expression* can also be the column label or number of an output column, as in:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

both of which sort by the first output column. Note that an output column name has to stand alone, that is, it cannot be used in an expression — for example, this is *not* correct:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          --
 wrong
```

This restriction is made to reduce ambiguity. There is still ambiguity if an ORDER BY item is a simple name that could match either an output column name or a column from the table expression. The output column is used in such cases. This would only cause confusion if you use AS to rename an output column to match some other table column's name.

ORDER BY can be applied to the result of a UNION, INTERSECT, or EXCEPT combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

7.6. LIMIT and OFFSET

LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query:

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

If a limit count is given, no more than that many rows will be returned (but possibly fewer, if the query itself yields fewer rows). LIMIT ALL is the same as omitting the LIMIT clause, as is LIMIT with a NULL argument.

OFFSET says to skip that many rows before beginning to return rows. OFFSET 0 is the same as omitting the OFFSET clause, as is OFFSET with a NULL argument.

If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is important to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows. You might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating query plans, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

The rows skipped by an OFFSET clause still have to be computed inside the server; therefore a large OFFSET might be inefficient.

7.7. VALUES Lists

VALUES provides a way to generate a “constant table” that can be used in a query without having to actually create and populate a table on-disk. The syntax is

```
VALUES ( expression [, ...] ) [, ...]
```

Each parenthesized list of expressions generates a row in the table. The lists must all have the same number of elements (i.e., the number of columns in the table), and corresponding entries in each list must have compatible data types. The actual data type assigned to each column of the result is determined using the same rules as for UNION (see Section 10.5).

As an example:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

will return a table of two columns and three rows. It's effectively equivalent to:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

By default, PostgreSQL assigns the names column1, column2, etc. to the columns of a VALUES table. The column names are not specified by the SQL standard and different database systems do it differently, so it's usually better to override the default names with a table alias list, like this:

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t
 (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

Syntactically, VALUES followed by expression lists is treated as equivalent to:

```
SELECT select_list FROM table_expression
```

and can appear anywhere a SELECT can. For example, you can use it as part of a UNION, or attach a *sort_specification* (ORDER BY, LIMIT, and/or OFFSET) to it. VALUES is most commonly used as the data source in an INSERT command, and next most commonly as a subquery.

For more information see VALUES.

7.8. WITH Queries (Common Table Expressions)

WITH provides a way to write auxiliary statements for use in a larger query. These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, DELETE, or MERGE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, DELETE, or MERGE.

7.8.1. SELECT in WITH

The basic value of SELECT in WITH is to break down complicated queries into simpler parts. An example is:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM
 regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

which displays per-product sales totals in only the top sales regions. The WITH clause defines two auxiliary statements named regional_sales and top_regions, where the output of regional_sales is used in top_regions and the output of top_regions is used in the primary SELECT query. This example could have been written without WITH, but we'd have needed two levels of nested sub-SELECTs. It's a bit easier to follow this way.

7.8.2. Recursive Queries

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output. A very simple example is this query to sum the integers from 1 through 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

The general form of a recursive WITH query is always a *non-recursive term*, then UNION (or UNION ALL), then a *recursive term*, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

Recursive Query Evaluation

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary *working table*.

2. So long as the working table is not empty, repeat these steps:

    a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and

rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary *intermediate table*.

b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

> **Note**
>
> While `RECURSIVE` allows queries to be specified recursively, internally such queries are evaluated iteratively.

In the example above, the working table has just a single row in each step, and it takes on the values from 1 through 100 in successive steps. In the 100th step, there is no output because of the `WHERE` clause, and so the query terminates.

Recursive queries are typically used to deal with hierarchical or tree-structured data. A useful example is this query to find all the direct and indirect sub-parts of a product, given only a table that shows immediate inclusions:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part =
 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

7.8.2.1. Search Order

When computing a tree traversal using a recursive query, you might want to order the results in either depth-first or breadth-first order. This can be done by computing an ordering column alongside the other data columns and using that to sort the results at the end. Note that this does not actually control in which order the query evaluation visits the rows; that is as always in SQL implementation-dependent. This approach merely provides a convenient way to order the results afterwards.

To create a depth-first order, we compute for each result row an array of rows that we have visited so far. For example, consider the following query that searches a table `tree` using a `link` field:

```
WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree;
```

To add depth-first ordering information, you can write this:

```
WITH RECURSIVE search_tree(id, link, data, path) AS (
    SELECT t.id, t.link, t.data, ARRAY[t.id]
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, path || t.id
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY path;
```

In the general case where more than one field needs to be used to identify a row, use an array of rows. For example, if we needed to track fields `f1` and `f2`:

```
WITH RECURSIVE search_tree(id, link, data, path) AS (
    SELECT t.id, t.link, t.data, ARRAY[ROW(t.f1, t.f2)]
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, path || ROW(t.f1, t.f2)
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY path;
```

> **Tip**
>
> Omit the `ROW()` syntax in the common case where only one field needs to be tracked. This allows a simple array rather than a composite-type array to be used, gaining efficiency.

To create a breadth-first order, you can add a column that tracks the depth of the search, for example:

```
WITH RECURSIVE search_tree(id, link, data, depth) AS (
    SELECT t.id, t.link, t.data, 0
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, depth + 1
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY depth;
```

To get a stable sort, add data columns as secondary sorting columns.

> **Tip**
>
> The recursive query evaluation algorithm produces its output in breadth-first search order. However, this is an implementation detail and it is perhaps unsound to rely on it. The order of the rows within each level is certainly undefined, so some explicit ordering might be desired in any case.

There is built-in syntax to compute a depth- or breadth-first sort column. For example:

```
WITH RECURSIVE search_tree(id, link, data) AS (
```

```
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
) SEARCH DEPTH FIRST BY id SET ordercol
SELECT * FROM search_tree ORDER BY ordercol;

WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
) SEARCH BREADTH FIRST BY id SET ordercol
SELECT * FROM search_tree ORDER BY ordercol;
```

This syntax is internally expanded to something similar to the above hand-written forms. The SEARCH clause specifies whether depth- or breadth first search is wanted, the list of columns to track for sorting, and a column name that will contain the result data that can be used for sorting. That column will implicitly be added to the output rows of the CTE.

7.8.2.2. Cycle Detection

When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. Sometimes, using UNION instead of UNION ALL can accomplish this by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are completely duplicate: it may be necessary to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the already-visited values. For example, consider again the following query that searches a table graph using a link field:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 0
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

This query will loop if the link relationships contain cycles. Because we require a “depth” output, just changing UNION ALL to UNION would not eliminate the looping. Instead we need to recognize whether we have reached the same row again while following a particular path of links. We add two columns is_cycle and path to the loop-prone query:

```
WITH RECURSIVE search_graph(id, link, data, depth, is_cycle, path)
 AS (
    SELECT g.id, g.link, g.data, 0,
      false,
      ARRAY[g.id]
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
```

```
      g.id = ANY(path),
      path || g.id
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT is_cycle
)
SELECT * FROM search_graph;
```

Aside from preventing cycles, the array value is often useful in its own right as representing the "path" taken to reach any particular row.

In the general case where more than one field needs to be checked to recognize a cycle, use an array of rows. For example, if we needed to compare fields `f1` and `f2`:

```
WITH RECURSIVE search_graph(id, link, data, depth, is_cycle, path)
 AS (
    SELECT g.id, g.link, g.data, 0,
      false,
      ARRAY[ROW(g.f1, g.f2)]
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      ROW(g.f1, g.f2) = ANY(path),
      path || ROW(g.f1, g.f2)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT is_cycle
)
SELECT * FROM search_graph;
```

> **Tip**
>
> Omit the `ROW()` syntax in the common case where only one field needs to be checked to recognize a cycle. This allows a simple array rather than a composite-type array to be used, gaining efficiency.

There is built-in syntax to simplify cycle detection. The above query can also be written like this:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
) CYCLE id SET is_cycle USING path
SELECT * FROM search_graph;
```

and it will be internally rewritten to the above form. The `CYCLE` clause specifies first the list of columns to track for cycle detection, then a column name that will show whether a cycle has been detected, and finally the name of another column that will track the path. The cycle and path columns will implicitly be added to the output rows of the CTE.

> **Tip**
>
> The cycle path column is computed in the same way as the depth-first ordering column show in the previous section. A query can have both a `SEARCH` and a `CYCLE` clause, but a depth-first

search specification and a cycle detection specification would create redundant computations, so it's more efficient to just use the CYCLE clause and order by the path column. If breadth-first ordering is wanted, then specifying both SEARCH and CYCLE can be useful.

A helpful trick for testing queries when you are not certain if they might loop is to place a LIMIT in the parent query. For example, this query would loop forever without the LIMIT:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

This works because PostgreSQL's implementation evaluates only as many rows of a WITH query as are actually fetched by the parent query. Using this trick in production is not recommended, because other systems might work differently. Also, it usually won't work if you make the outer query sort the recursive query's results or join them to some other table, because in such cases the outer query will usually try to fetch all of the WITH query's output anyway.

7.8.3. Common Table Expression Materialization

A useful property of WITH queries is that they are normally evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, expensive calculations that are needed in multiple places can be placed within a WITH query to avoid redundant work. Another possible application is to prevent unwanted multiple evaluations of functions with side-effects. However, the other side of this coin is that the optimizer is not able to push restrictions from the parent query down into a multiply-referenced WITH query, since that might affect all uses of the WITH query's output when it should affect only one. The multiply-referenced WITH query will be evaluated as written, without suppression of rows that the parent query might discard afterwards. (But, as mentioned above, evaluation might stop early if the reference(s) to the query demand only a limited number of rows.)

However, if a WITH query is non-recursive and side-effect-free (that is, it is a SELECT containing no volatile functions) then it can be folded into the parent query, allowing joint optimization of the two query levels. By default, this happens if the parent query references the WITH query just once, but not if it references the WITH query more than once. You can override that decision by specifying MATERIALIZED to force separate calculation of the WITH query, or by specifying NOT MATERIALIZED to force it to be merged into the parent query. The latter choice risks duplicate computation of the WITH query, but it can still give a net savings if each usage of the WITH query needs only a small part of the WITH query's full output.

A simple example of these rules is

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w WHERE key = 123;
```

This WITH query will be folded, producing the same execution plan as

```
SELECT * FROM big_table WHERE key = 123;
```

In particular, if there's an index on key, it will probably be used to fetch just the rows having key = 123. On the other hand, in

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

the `WITH` query will be materialized, producing a temporary copy of `big_table` that is then joined with itself — without benefit of any index. This query will be executed much more efficiently if written as

```
WITH w AS NOT MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

so that the parent query's restrictions can be applied directly to scans of `big_table`.

An example where `NOT MATERIALIZED` could be undesirable is

```
WITH w AS (
    SELECT key, very_expensive_function(val) as f FROM some_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.f = w2.f;
```

Here, materialization of the `WITH` query ensures that `very_expensive_function` is evaluated only once per table row, not twice.

The examples above only show `WITH` being used with `SELECT`, but it can be attached in the same way to `INSERT`, `UPDATE`, `DELETE`, or `MERGE`. In each case it effectively provides temporary table(s) that can be referred to in the main command.

7.8.4. Data-Modifying Statements in `WITH`

You can use data-modifying statements (`INSERT`, `UPDATE`, `DELETE`, or `MERGE`) in `WITH`. This allows you to perform several different operations in the same query. An example is:

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

This query effectively moves rows from `products` to `products_log`. The `DELETE` in `WITH` deletes the specified rows from `products`, returning their contents by means of its `RETURNING` clause; and then the primary query reads that output and inserts it into `products_log`.

A fine point of the above example is that the `WITH` clause is attached to the `INSERT`, not the sub-`SELECT` within the `INSERT`. This is necessary because data-modifying statements are only allowed in `WITH` clauses that are attached to the top-level statement. However, normal `WITH` visibility rules apply, so it is possible to refer to the `WITH` statement's output from the sub-`SELECT`.

Data-modifying statements in WITH usually have RETURNING clauses (see Section 6.4), as shown in the example above. It is the output of the RETURNING clause, *not* the target table of the data-modifying statement, that forms the temporary table that can be referred to by the rest of the query. If a data-modifying statement in WITH lacks a RETURNING clause, then it forms no temporary table and cannot be referred to in the rest of the query. Such a statement will be executed nonetheless. A not-particularly-useful example is:

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

This example would remove all rows from tables foo and bar. The number of affected rows reported to the client would only include rows removed from bar.

Recursive self-references in data-modifying statements are not allowed. In some cases it is possible to work around this limitation by referring to the output of a recursive WITH, for example:

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

This query would remove all direct and indirect subparts of a product.

Data-modifying statements in WITH are executed exactly once, and always to completion, independently of whether the primary query reads all (or indeed any) of their output. Notice that this is different from the rule for SELECT in WITH: as stated in the previous section, execution of a SELECT is carried only as far as the primary query demands its output.

The sub-statements in WITH are executed concurrently with each other and with the main query. Therefore, when using data-modifying statements in WITH, the order in which the specified updates actually happen is unpredictable. All the statements are executed with the same *snapshot* (see Chapter 13), so they cannot “see” one another's effects on the target tables. This alleviates the effects of the unpredictability of the actual order of row updates, and means that RETURNING data is the only way to communicate changes between different WITH sub-statements and the main query. An example of this is that in

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

the outer SELECT would return the original prices before the action of the UPDATE, while in

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

the outer SELECT would return the updated data.

Trying to update the same row twice in a single statement is not supported. Only one of the modifications takes place, but it is not easy (and sometimes not possible) to reliably predict which one. This also applies to deleting a row that was already updated in the same statement: only the update is performed. Therefore you should generally avoid trying to modify a single row twice in a single statement. In particular avoid writing WITH sub-statements that could affect the same rows changed by the main statement or a sibling sub-statement. The effects of such a statement will not be predictable.

At present, any table used as the target of a data-modifying statement in WITH must not have a conditional rule, nor an ALSO rule, nor an INSTEAD rule that expands to multiple statements.

Chapter 8. Data Types

PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the CREATE TYPE command.

Table 8.1 shows all the built-in general-purpose data types. Most of the alternative names listed in the “Aliases” column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but are not listed here.

Table 8.1. Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(*n*)]		fixed-length bit string
bit varying [(*n*)]	varbit [(*n*)]	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data (“byte array”)
character [(*n*)]	char [(*n*)]	fixed-length character string
character varying [(*n*)]	varchar [(*n*)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [*fields*] [(*p*)]		time span
json		textual JSON data
jsonb		binary JSON data, decomposed
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
macaddr8		MAC (Media Access Control) address (EUI-64 format)
money		currency amount
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	exact numeric of selectable precision
path		geometric path on a plane
pg_lsn		PostgreSQL Log Sequence Number
pg_snapshot		user-level transaction ID snapshot
point		geometric point on a plane

Name	Aliases	Description
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time zone]		time of day (no time zone)
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] [without time zone]		date and time (no time zone)
timestamp [(*p*)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot (deprecated; see pg_snapshot)
uuid		universally unique identifier
xml		XML data

Compatibility

The following types (or spellings thereof) are specified by SQL: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (with or without time zone), timestamp (with or without time zone), xml.

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as geometric paths, or have several possible formats, such as the date and time types. Some of the input and output functions are not invertible, i.e., the result of an output function might lose accuracy when compared to the original input.

8.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals. Table 8.2 lists the available types.

Table 8.2. Numeric Types

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807

Name	Storage Size	Description	Range
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

The syntax of constants for the numeric types is described in Section 4.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 9 for more information. The following sections describe the types in detail.

8.1.1. Integer Types

The types `smallint`, `integer`, and `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type `integer` is the common choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type is designed to be used when the range of the `integer` type is insufficient.

SQL only specifies the integer types `integer` (or `int`), `smallint`, and `bigint`. The type names `int2`, `int4`, and `int8` are extensions, which are also used by some other SQL database systems.

8.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required. Calculations with `numeric` values yield exact results where possible, e.g., addition, subtraction, multiplication. However, calculations on `numeric` values are very slow compared to the integer types, or to the floating-point types described in the next section.

We use the following terms below: The *precision* of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The *scale* of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the maximum precision and the maximum scale of a `numeric` column can be configured. To declare a column of type `numeric` use the syntax:

```
NUMERIC(precision, scale)
```

The precision must be positive, while the scale may be positive or negative (see below). Alternatively:

`NUMERIC(`*precision*`)`

selects a scale of 0. Specifying:

`NUMERIC`

without any precision or scale creates an “unconstrained numeric” column in which numeric values of any length can be stored, up to the implementation limits. A column of this kind will not coerce input values to any particular scale, whereas `numeric` columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

> **Note**
>
> The maximum precision that can be explicitly specified in a `numeric` type declaration is 1000. An unconstrained `numeric` column is subject to the limits described in Table 8.2.

If the scale of a value to be stored is greater than the declared scale of the column, the system will round the value to the specified number of fractional digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error is raised. For example, a column declared as

`NUMERIC(3, 1)`

will round values to 1 decimal place and can store values between -99.9 and 99.9, inclusive.

Beginning in PostgreSQL 15, it is allowed to declare a `numeric` column with a negative scale. Then values will be rounded to the left of the decimal point. The precision still represents the maximum number of non-rounded digits. Thus, a column declared as

`NUMERIC(2, -3)`

will round values to the nearest thousand and can store values between -99000 and 99000, inclusive. It is also allowed to declare a scale larger than the declared precision. Such a column can only hold fractional values, and it requires the number of zero digits just to the right of the decimal point to be at least the declared scale minus the declared precision. For example, a column declared as

`NUMERIC(3, 5)`

will round values to 5 decimal places and can store values between -0.00999 and 0.00999, inclusive.

> **Note**
>
> PostgreSQL permits the scale in a `numeric` type declaration to be any value in the range -1000 to 1000. However, the SQL standard requires the scale to be in the range 0 to *`precision`*. Using scales outside that range may not be portable to other database systems.

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the `numeric` type is more akin to `varchar(`*`n`*`)` than to `char(`*`n`*`)`.) The actual storage requirement is two bytes for each group of four decimal digits, plus three to eight bytes overhead.

In addition to ordinary numeric values, the `numeric` type has several special values:

```
Infinity
-Infinity
NaN
```

These are adapted from the IEEE 754 standard, and represent “infinity”, “negative infinity”, and “not-a-number”, respectively. When writing these values as constants in an SQL command, you must put quotes around them, for example `UPDATE table SET x = '-Infinity'`. On input, these strings are recognized in a case-insensitive manner. The infinity values can alternatively be spelled `inf` and `-inf`.

The infinity values behave as per mathematical expectations. For example, `Infinity` plus any finite value equals `Infinity`, as does `Infinity` plus `Infinity`; but `Infinity` minus `Infinity` yields `NaN` (not a number), because it has no well-defined interpretation. Note that an infinity can only be stored in an unconstrained `numeric` column, because it notionally exceeds any finite precision limit.

The `NaN` (not a number) value is used to represent undefined calculational results. In general, any operation with a `NaN` input yields another `NaN`. The only exception is when the operation's other inputs are such that the same output would be obtained if the `NaN` were to be replaced by any finite or infinite numeric value; then, that output value is used for `NaN` too. (An example of this principle is that `NaN` raised to the zero power yields one.)

> **Note**
>
> In most implementations of the “not-a-number” concept, `NaN` is not considered equal to any other numeric value (including `NaN`). In order to allow `numeric` values to be sorted and used in tree-based indexes, PostgreSQL treats `NaN` values as equal, and greater than all non-`NaN` values.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard.

When rounding values, the `numeric` type rounds ties away from zero, while (on most machines) the `real` and `double precision` types round ties to the nearest even number. For example:

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

8.1.3. Floating-Point Types

The data types `real` and `double precision` are inexact, variable-precision numeric types. On all currently supported platforms, these types are implementations of IEEE Standard 754 for Binary

Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the `numeric` type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality might not always work as expected.

On all currently supported platforms, the `real` type has a range of around 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The `double precision` type has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

By default, floating point values are output in text form in their shortest precise decimal representation; the decimal value produced is closer to the true stored binary value than to any other value representable in the same binary precision. (However, the output value is currently never *exactly* midway between two representable values, in order to avoid a widespread bug where input routines do not properly respect the round-to-nearest-even rule.) This value will use at most 17 significant decimal digits for `float8` values, and at most 9 digits for `float4` values.

> **Note**
>
> This shortest-precise output format is much faster to generate than the historical rounded format.

For compatibility with output generated by older versions of PostgreSQL, and to allow the output precision to be reduced, the extra_float_digits parameter can be used to select rounded decimal output instead. Setting a value of 0 restores the previous default of rounding the value to 6 (for `float4`) or 15 (for `float8`) significant decimal digits. Setting a negative value reduces the number of digits further; for example -2 would round output to 4 or 13 digits respectively.

Any value of extra_float_digits greater than 0 selects the shortest-precise format.

> **Note**
>
> Applications that wanted precise values have historically had to set extra_float_digits to 3 to obtain them. For maximum compatibility between versions, they should continue to do so.

In addition to ordinary numeric values, the floating-point types have several special values:

```
Infinity
-Infinity
NaN
```

These represent the IEEE 754 special values “infinity”, “negative infinity”, and “not-a-number”, respectively. When writing these values as constants in an SQL command, you must put quotes around

them, for example `UPDATE table SET x = '-Infinity'`. On input, these strings are recognized in a case-insensitive manner. The infinity values can alternatively be spelled `inf` and `-inf`.

> **Note**
>
> IEEE 754 specifies that `NaN` should not compare equal to any other floating-point value (including `NaN`). In order to allow floating-point values to be sorted and used in tree-based indexes, PostgreSQL treats `NaN` values as equal, and greater than all non-`NaN` values.

PostgreSQL also supports the SQL-standard notations `float` and `float(`*p*`)` for specifying inexact numeric types. Here, *p* specifies the minimum acceptable precision in *binary* digits. PostgreSQL accepts `float(1)` to `float(24)` as selecting the `real` type, while `float(25)` to `float(53)` select `double precision`. Values of *p* outside the allowed range draw an error. `float` with no precision specified is taken to mean `double precision`.

8.1.4. Serial Types

> **Note**
>
> This section describes a PostgreSQL-specific way to create an autoincrementing column. Another way is to use the SQL-standard identity column feature, described at Section 5.3.

The data types `smallserial`, `serial` and `bigserial` are not true types, but merely a notational convenience for creating unique identifier columns (similar to the `AUTO_INCREMENT` property supported by some other databases). In the current implementation, specifying:

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

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT
 nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. A `NOT NULL` constraint is applied to ensure that a null value cannot be inserted. (In most cases you would also want to attach a `UNIQUE` or `PRIMARY KEY` constraint to prevent duplicate values from being inserted by accident, but this is not automatic.) Lastly, the sequence is marked as “owned by” the column, so that it will be dropped if the column or table is dropped.

> **Note**
>
> Because `smallserial`, `serial` and `bigserial` are implemented using sequences, there may be "holes" or gaps in the sequence of values which appears in the column, even if no rows are ever deleted. A value allocated from the sequence is still "used up" even if a row containing that value is never successfully inserted into the table column. This may happen, for example, if the inserting transaction rolls back. See `nextval()` in Section 9.17 for details.

To insert the next value of the sequence into the `serial` column, specify that the `serial` column should be assigned its default value. This can be done either by excluding the column from the list of columns in the `INSERT` statement, or through the use of the `DEFAULT` key word.

The type names `serial` and `serial4` are equivalent: both create `integer` columns. The type names `bigserial` and `serial8` work the same way, except that they create a `bigint` column. `bigserial` should be used if you anticipate the use of more than 2^{31} identifiers over the lifetime of the table. The type names `smallserial` and `serial2` also work the same way, except that they create a `smallint` column.

The sequence created for a `serial` column is automatically dropped when the owning column is dropped. You can drop the sequence without dropping the column, but this will force removal of the column default expression.

8.2. Monetary Types

The `money` type stores a currency amount with a fixed fractional precision; see Table 8.3. The fractional precision is determined by the database's lc_monetary setting. The range shown in the table assumes there are two fractional digits. Input is accepted in a variety of formats, including integer and floating-point literals, as well as typical currency formatting, such as `'$1,000.00'`. Output is generally in the latter form but depends on the locale.

Table 8.3. Monetary Types

Name	Storage Size	Description	Range
`money`	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Since the output of this data type is locale-sensitive, it might not work to load `money` data into a database that has a different setting of `lc_monetary`. To avoid problems, before restoring a dump into a new database make sure `lc_monetary` has the same or equivalent value as in the database that was dumped.

Values of the `numeric`, `int`, and `bigint` data types can be cast to `money`. Conversion from the `real` and `double precision` data types can be done by casting to `numeric` first, for example:

```
SELECT '12.34'::float8::numeric::money;
```

However, this is not recommended. Floating point numbers should not be used to handle money due to the potential for rounding errors.

A `money` value can be cast to `numeric` without loss of precision. Conversion to other types could potentially lose precision, and must also be done in two stages:

```
SELECT '52093.89'::money::numeric::float8;
```

Division of a `money` value by an integer value is performed with truncation of the fractional part towards zero. To get a rounded result, divide by a floating-point value, or cast the `money` value to `numeric` before dividing and back to `money` afterwards. (The latter is preferable to avoid risking precision loss.) When a `money` value is divided by another `money` value, the result is `double precision` (i.e., a pure number, not money); the currency units cancel each other out in the division.

8.3. Character Types

Table 8.4. Character Types

Name	Description
`character varying(`*n*`)`, `varchar(`*n*`)`	variable-length with limit
`character(`*n*`)`, `char(`*n*`)`, `bpchar(`*n*`)`	fixed-length, blank-padded
`bpchar`	variable unlimited length, blank-trimmed
`text`	variable unlimited length

Table 8.4 shows the general-purpose character types available in PostgreSQL.

SQL defines two primary character types: `character varying(`*n*`)` and `character(`*n*`)`, where *n* is a positive integer. Both of these types can store strings up to *n* characters (not bytes) in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) However, if one explicitly casts a value to `character varying(`*n*`)` or `character(`*n*`)`, then an over-length value will be truncated to *n* characters without raising an error. (This too is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type `character` will be space-padded; values of type `character varying` will simply store the shorter string.

In addition, PostgreSQL provides the `text` type, which stores strings of any length. Although the `text` type is not in the SQL standard, several other SQL database management systems have it as well. `text` is PostgreSQL's native string data type, in that most built-in functions operating on strings are declared to take or return `text` not `character varying`. For many purposes, `character varying` acts as though it were a domain over `text`.

The type name `varchar` is an alias for `character varying`, while `bpchar` (with length specifier) and `char` are aliases for `character`. The `varchar` and `char` aliases are defined in the SQL standard; `bpchar` is a PostgreSQL extension.

If specified, the length *n* must be greater than zero and cannot exceed 10,485,760. If `character varying` (or `varchar`) is used without length specifier, the type accepts strings of any length. If `bpchar` lacks a length specifier, it also accepts strings of any length, but trailing spaces are semantically insignificant. If `character` (or `char`) lacks a specifier, it is equivalent to `character(1)`.

Values of type `character` are physically padded with spaces to the specified width *n*, and are stored and displayed that way. However, trailing spaces are treated as semantically insignificant and disregarded when comparing two values of type `character`. In collations where whitespace is significant, this behavior can produce unexpected results; for example `SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` returns true, even though `C` locale would consider a space to be greater than a newline. Trailing spaces are removed when converting a `character` value to one of the other string types. Note that trailing spaces *are* semantically significant in `character varying` and `text` values, and when using pattern matching, that is `LIKE` and regular expressions.

The characters that can be stored in any of these data types are determined by the database character set, which is selected when the database is created. Regardless of the specific character set, the character with code zero (sometimes called NUL) cannot be stored. For more information refer to Section 23.3.

The storage requirement for a short string (up to 126 bytes) is 1 byte plus the actual string, which includes the space padding in the case of `character`. Longer strings have 4 bytes of overhead instead of 1. Long strings are compressed by the system automatically, so the physical requirement on disk might be less. Very long values are also stored in background tables so that they do not interfere with rapid access to shorter column values. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for *n* in the data type declaration is less than that. It wouldn't be useful to change this because with multibyte character encodings the number of characters and bytes can be quite different. If you desire to store long strings with no specific upper limit, use `text` or `character varying` without a length specifier, rather than making up an arbitrary length limit.)

> **Tip**
>
> There is no performance difference among these three types, apart from increased storage space when using the blank-padded type, and a few extra CPU cycles to check the length when storing into a length-constrained column. While `character(n)` has performance advantages in some other database systems, there is no such advantage in PostgreSQL; in fact `character(n)` is usually the slowest of the three because of its additional storage costs. In most situations `text` or `character varying` should be used instead.

Refer to Section 4.1.2.1 for information about the syntax of string literals, and to Chapter 9 for information about available operators and functions.

Example 8.1. Using the Character Types

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit
 truncation
SELECT b, char_length(b) FROM test2;

   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

1 The `char_length` function is discussed in Section 9.4.

There are two other fixed-length character types in PostgreSQL, shown in Table 8.5. These are not intended for general-purpose use, only for use in the internal system catalogs. The `name` type is used to store identifiers. Its length is currently defined as 64 bytes (63 usable characters plus terminator) but should be referenced using the constant `NAMEDATALEN` in `C` source code. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length might change in a future release. The type `"char"` (note the quotes) is different from `char(1)` in that it only uses one byte of storage, and therefore can store only a single ASCII character. It is used in the system catalogs as a simplistic enumeration type.

Table 8.5. Special Character Types

Name	Storage Size	Description
`"char"`	1 byte	single-byte internal type

Name	Storage Size	Description
name	64 bytes	internal type for object names

8.4. Binary Data Types

The bytea data type allows storage of binary strings; see Table 8.6.

Table 8.6. Binary Data Types

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from character strings in two ways. First, binary strings specifically allow storing octets of value zero and other “non-printable” octets (usually, octets outside the decimal range 32 to 126). Character strings disallow zero octets, and also disallow any other octet values and sequences of octet values that are invalid according to the database's selected character set encoding. Second, operations on binary strings process the actual bytes, whereas the processing of character strings depends on locale settings. In short, binary strings are appropriate for storing data that the programmer thinks of as “raw bytes”, whereas character strings are appropriate for storing text.

The bytea type supports two formats for input and output: “hex” format and PostgreSQL's historical “escape” format. Both of these are always accepted on input. The output format depends on the configuration parameter bytea_output; the default is hex. (Note that the hex format was introduced in PostgreSQL 9.0; earlier versions and some tools don't understand it.)

The SQL standard defines a different binary string type, called BLOB or BINARY LARGE OBJECT. The input format is different from bytea, but the provided functions and operators are mostly the same.

8.4.1. bytea Hex Format

The “hex” format encodes binary data as 2 hexadecimal digits per byte, most significant nibble first. The entire string is preceded by the sequence \x (to distinguish it from the escape format). In some contexts, the initial backslash may need to be escaped by doubling it (see Section 4.1.2.1). For input, the hexadecimal digits can be either upper or lower case, and whitespace is permitted between digit pairs (but not within a digit pair nor in the starting \x sequence). The hex format is compatible with a wide range of external applications and protocols, and it tends to be faster to convert than the escape format, so its use is preferred.

Example:

```
SET bytea_output = 'hex';

SELECT '\xDEADBEEF'::bytea;
   bytea
------------
 \xdeadbeef
```

8.4.2. bytea Escape Format

The “escape” format is the traditional PostgreSQL format for the bytea type. It takes the approach of representing a binary string as a sequence of ASCII characters, while converting those bytes that cannot be represented as an ASCII character into special escape sequences. If, from the point of view of the application, representing bytes as characters makes sense, then this representation can be convenient.

But in practice it is usually confusing because it fuzzes up the distinction between binary strings and character strings, and also the particular escape mechanism that was chosen is somewhat unwieldy. Therefore, this format should probably be avoided for most new applications.

When entering `bytea` values in escape format, octets of certain values *must* be escaped, while all octet values *can* be escaped. In general, to escape an octet, convert it into its three-digit octal value and precede it by a backslash. Backslash itself (octet decimal value 92) can alternatively be represented by double backslashes. Table 8.7 shows the characters that must be escaped, and gives the alternative escape sequences where applicable.

Table 8.7. `bytea` Literal Escaped Octets

Decimal Octet Value	Description	Escaped Input Representation	Example	Hex Representation
0	zero octet	`'\000'`	`'\000'::bytea`	`\x00`
39	single quote	`''''` or `'\047'`	`''''::bytea`	`\x27`
92	backslash	`'\\'` or `'\134'`	`'\\'::bytea`	`\x5c`
0 to 31 and 127 to 255	“non-printable” octets	`'\`*xxx*`'` (octal value)	`'\001'::bytea`	`\x01`

The requirement to escape *non-printable* octets varies depending on locale settings. In some instances you can get away with leaving them unescaped.

The reason that single quotes must be doubled, as shown in Table 8.7, is that this is true for any string literal in an SQL command. The generic string-literal parser consumes the outermost single quotes and reduces any pair of single quotes to one data character. What the `bytea` input function sees is just one single quote, which it treats as a plain data character. However, the `bytea` input function treats backslashes as special, and the other behaviors shown in Table 8.7 are implemented by that function.

In some contexts, backslashes must be doubled compared to what is shown above, because the generic string-literal parser will also reduce pairs of backslashes to one data character; see Section 4.1.2.1.

`Bytea` octets are output in `hex` format by default. If you change bytea_output to `escape`, “non-printable” octets are converted to their equivalent three-digit octal value and preceded by one backslash. Most “printable” octets are output by their standard representation in the client character set, e.g.:

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

The octet with decimal value 92 (backslash) is doubled in the output. Details are in Table 8.8.

Table 8.8. `bytea` Output Escaped Octets

Decimal Octet Value	Description	Escaped Output Representation	Example	Output Result
92	backslash	`\\`	`'\134'::bytea`	`\\`
0 to 31 and 127 to 255	“non-printable” octets	`\`*xxx* (octal value)	`'\001'::bytea`	`\001`
32 to 126	“printable” octets	client character set representation	`'\176'::bytea`	~

Depending on the front end to PostgreSQL you use, you might have additional work to do in terms of escaping and unescaping `bytea` strings. For example, you might also have to escape line feeds and carriage returns if your interface automatically translates these.

8.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in Table 8.9. The operations available on these data types are described in Section 9.9. Dates are counted according to the Gregorian calendar, even in years before that calendar was introduced (see Section B.6 for more information).

Table 8.9. Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
`timestamp [ (p) ] [ without time zone ]`	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond
`timestamp [ (p) ] with time zone`	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond
`date`	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
`time [ (p) ] [ without time zone ]`	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond
`time [ (p) ] with time zone`	12 bytes	time of day (no date), with time zone	00:00:00+1559	24:00:00-1559	1 microsecond
`interval [ fields ] [ (p) ]`	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond

> **Note**
>
> The SQL standard requires that writing just `timestamp` be equivalent to `timestamp without time zone`, and PostgreSQL honors that behavior. `timestamptz` is accepted as an abbreviation for `timestamp with time zone`; this is a PostgreSQL extension.

`time`, `timestamp`, and `interval` accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of *p* is from 0 to 6.

The `interval` type has an additional option, which is to restrict the set of stored fields by writing one of these phrases:

```
YEAR
MONTH
DAY
```

```
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Note that if both *fields* and *p* are specified, the *fields* must include SECOND, since the precision applies only to the seconds.

The type time with time zone is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of date, time, timestamp without time zone, and timestamp with time zone should provide a complete range of date/time functionality required by any application.

8.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of day, month, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the DateStyle parameter to MDY to select month-day-year interpretation, DMY to select day-month-year interpretation, or YMD to select year-month-day interpretation.

PostgreSQL is more flexible in handling date/time input than the SQL standard requires. See Appendix B for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to Section 4.1.2.7 for more information. SQL requires the following syntax

type [(*p*)] '*value*'

where *p* is an optional precision specification giving the number of fractional digits in the seconds field. Precision can be specified for time, timestamp, and interval types, and can range from 0 to 6. If no precision is specified in a constant specification, it defaults to the precision of the literal value (but not more than 6 digits).

8.5.1.1. Dates

Table 8.10 shows some possible inputs for the date type.

Table 8.10. Date Input

Example	Description
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
January 8, 1999	unambiguous in any datestyle input mode
1/8/1999	January 8 in MDY mode; August 1 in DMY mode
1/18/1999	January 18 in MDY mode; rejected in other modes
01/02/03	January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode
1999-Jan-08	January 8 in any mode

Example	Description
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in `YMD` mode, else error
08-Jan-99	January 8, except error in `YMD` mode
Jan-08-99	January 8, except error in `YMD` mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	year and day of year
J2451187	Julian date
January 8, 99 BC	year 99 BC

8.5.1.2. Times

The time-of-day types are `time [ (`*p*`) ] without time zone` and `time [ (`*p*`) ] with time zone`. `time` alone is equivalent to `time without time zone`.

Valid input for these types consists of a time of day followed by an optional time zone. (See Table 8.11 and Table 8.12.) If a time zone is specified in the input for `time without time zone`, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as `America/New_York`. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the `time with time zone` value and is output as stored; it is not adjusted to the active time zone.

Table 8.11. Time Input

Example	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	same as 04:05; AM does not affect value
`04:05 PM`	same as 16:05; input hour must be <= 12
`04:05:06.789-8`	ISO 8601, with time zone as UTC offset
`04:05:06-08:00`	ISO 8601, with time zone as UTC offset
`04:05-08:00`	ISO 8601, with time zone as UTC offset
`040506-08`	ISO 8601, with time zone as UTC offset
`040506+0730`	ISO 8601, with fractional-hour time zone as UTC offset
`040506+07:30:00`	UTC offset specified to seconds (not allowed in ISO 8601)
`04:05:06 PST`	time zone specified by abbreviation

Example	Description
2003-04-12 04:05:06 America/New_York	time zone specified by full name

Table 8.12. Time Zone Input

Example	Description
PST	Abbreviation (for Pacific Standard Time)
America/New_York	Full time zone name
PST8PDT	POSIX-style time zone specification
-8:00:00	UTC offset for PST
-8:00	UTC offset for PST (ISO 8601 extended format)
-800	UTC offset for PST (ISO 8601 basic format)
-8	UTC offset for PST (ISO 8601 basic format)
zulu	Military abbreviation for UTC
z	Short form of zulu (also in ISO 8601)

Refer to Section 8.5.3 for more information on how to specify time zones.

8.5.1.3. Time Stamps

Valid input for the time stamp types consists of the concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC. (Alternatively, AD/BC can appear before the time zone, but this is not the preferred ordering.) Thus:

```
1999-01-08 04:05:06
```

and:

```
1999-01-08 04:05:06 -8:00
```

are valid values, which follow the ISO 8601 standard. In addition, the common format:

```
January 8 04:05:06 1999 PST
```

is supported.

The SQL standard differentiates `timestamp without time zone` and `timestamp with time zone` literals by the presence of a "+" or "-" symbol and time zone offset after the time. Hence, according to the standard,

```
TIMESTAMP '2004-10-19 10:23:54'
```

is a `timestamp without time zone`, while

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

is a `timestamp with time zone`. PostgreSQL never examines the content of a literal string before determining its type, and therefore will treat both of the above as `timestamp without time zone`. To ensure that a literal is treated as `timestamp with time zone`, give it the correct explicit type:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

In a value that has been determined to be `timestamp without time zone`, PostgreSQL will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input string, and is not adjusted for time zone.

For `timestamp with time zone` values, an input string that includes an explicit time zone will be converted to UTC (*Universal Coordinated Time*) using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's TimeZone parameter, and is converted to UTC using the offset for the `timezone` zone. In either case, the value is stored internally as UTC, and the originally stated or assumed time zone is not retained.

When a `timestamp with time zone` value is output, it is always converted from UTC to the current `timezone` zone, and displayed as local time in that zone. To see the time in another time zone, either change `timezone` or use the `AT TIME ZONE` construct (see Section 9.9.4).

Conversions between `timestamp without time zone` and `timestamp with time zone` normally assume that the `timestamp without time zone` value should be taken or given as `timezone` local time. A different time zone can be specified for the conversion using `AT TIME ZONE`.

8.5.1.4. Special Values

PostgreSQL supports several special date/time input values for convenience, as shown in Table 8.13. The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, `now` and related strings are converted to a specific time value as soon as they are read.) All of these values need to be enclosed in single quotes when used as constants in SQL commands.

Table 8.13. Special Date/Time Inputs

Input String	Valid Types	Description
`epoch`	`date`, `timestamp`	1970-01-01 00:00:00+00 (Unix system time zero)
`infinity`	`date`, `timestamp`, `interval`	later than all other time stamps
`-infinity`	`date`, `timestamp`, `interval`	earlier than all other time stamps
`now`	`date`, `time`, `timestamp`	current transaction's start time
`today`	`date`, `timestamp`	midnight (`00:00`) today
`tomorrow`	`date`, `timestamp`	midnight (`00:00`) tomorrow
`yesterday`	`date`, `timestamp`	midnight (`00:00`) yesterday
`allballs`	`time`	00:00:00.00 UTC

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`. (See Section 9.9.5.) Note that these are SQL functions and are *not* recognized in data input strings.

> **Caution**
>
> While the input strings `now`, `today`, `tomorrow`, and `yesterday` are fine to use in interactive SQL commands, they can have surprising behavior when the command is saved to be

executed later, for example in prepared statements, views, and function definitions. The string can be converted to a specific time value that continues to be used long after it becomes stale. Use one of the SQL functions instead in such contexts. For example, `CURRENT_DATE + 1` is safer than `'tomorrow'::date`.

8.5.2. Date/Time Output

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES (Unix date format), or German. The default is the ISO format. (The SQL standard requires the use of the ISO 8601 format. The name of the "SQL" output format is a historical accident.) Table 8.14 shows examples of each output style. The output of the `date` and `time` types is generally only the date or time part in accordance with the given examples. However, the POSTGRES style outputs date-only values in ISO format.

Table 8.14. Date/Time Output Styles

Style Specification	Description	Example
`ISO`	ISO 8601, SQL standard	`1997-12-17 07:37:16-08`
`SQL`	traditional style	`12/17/1997 07:37:16.00 PST`
`Postgres`	original style	`Wed Dec 17 07:37:16 1997 PST`
`German`	regional style	`17.12.1997 07:37:16.00 PST`

> **Note**
>
> ISO 8601 specifies the use of uppercase letter `T` to separate the date and time. PostgreSQL accepts that format on input, but on output it uses a space rather than `T`, as shown above. This is for readability and for consistency with RFC 3339[1] as well as some other database systems.

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See Section 8.5.1 for how this setting also affects interpretation of input values.) Table 8.15 shows examples.

Table 8.15. Date Order Conventions

`datestyle` Setting	Input Ordering	Example Output
`SQL, DMY`	*day*/*month*/*year*	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	*month*/*day*/*year*	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	*day*/*month*/*year*	`Wed 17 Dec 07:37:16 1997 PST`

In the ISO style, the time zone is always shown as a signed numeric offset from UTC, with positive sign used for zones east of Greenwich. The offset will be shown as *hh* (hours only) if it is an integral number of hours, else as *hh*:*mm* if it is an integral number of minutes, else as *hh*:*mm*:*ss*. (The third case is not possible with any modern time zone standard, but it can appear when working with timestamps that predate the adoption of standardized time zones.) In the other date styles, the time zone is shown as an alphabetic abbreviation if one is in common use in the current zone. Otherwise it appears as a signed numeric offset in ISO 8601 basic format (*hh* or *hhmm*).

The date/time style can be selected by the user using the `SET datestyle` command, the DateStyle parameter in the `postgresql.conf` configuration file, or the `PGDATESTYLE` environment variable on the server or client.

[1] https://datatracker.ietf.org/doc/html/rfc3339

The formatting function `to_char` (see Section 9.8) is also available as a more flexible way to format date/time output.

8.5.3. Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900s, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. PostgreSQL uses the widely-used IANA (Olson) time zone database for information about historical time zone rules. For times in the future, the assumption is that the latest known rules for a given time zone will continue to be observed indefinitely far into the future.

PostgreSQL endeavors to be compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type cannot have an associated time zone, the `time` type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore impossible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We do *not* recommend using the type `time with time zone` (though it is supported by PostgreSQL for legacy applications and for compliance with the SQL standard). PostgreSQL assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the TimeZone configuration parameter before being displayed to the client.

PostgreSQL allows you to specify time zones in three different forms:

- A full time zone name, for example `America/New_York`. The recognized time zone names are listed in the `pg_timezone_names` view (see Section 52.32). PostgreSQL uses the widely-used IANA time zone data for this purpose, so the same time zone names are also recognized by other software.
- A time zone abbreviation, for example `PST`. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which can imply a set of daylight savings transition rules as well. The recognized abbreviations are listed in the `pg_timezone_abbrevs` view (see Section 52.31). You cannot set the configuration parameters TimeZone or log_timezone to a time zone abbreviation, but you can use abbreviations in date/time input values and with the `AT TIME ZONE` operator.
- In addition to the timezone names and abbreviations, PostgreSQL will accept POSIX-style time zone specifications, as described in Section B.5. This option is not normally preferable to using a named time zone, but it may be necessary if no suitable IANA time zone entry is available.

In short, this is the difference between abbreviations and full names: abbreviations represent a specific offset from UTC, whereas many of the full names imply a local daylight-savings time rule, and so have two possible UTC offsets. As an example, `2014-06-04 12:00 America/New_York` represents noon local time in New York, which for this particular date was Eastern Daylight Time (UTC-4). So `2014-06-04 12:00 EDT` specifies that same time instant. But `2014-06-04 12:00 EST` specifies noon Eastern Standard Time (UTC-5), regardless of whether daylight savings was nominally in effect on that date.

To complicate matters, some jurisdictions have used the same timezone abbreviation to mean different UTC offsets at different times; for example, in Moscow `MSK` has meant UTC+3 in some years and UTC+4 in others. PostgreSQL interprets such abbreviations according to whatever they meant (or had most recently meant) on the specified date; but, as with the `EST` example above, this is not necessarily the same as local civil time on that date.

In all cases, timezone names and abbreviations are recognized case-insensitively. (This is a change from PostgreSQL versions prior to 8.2, which were case-sensitive in some contexts but not others.)

Neither timezone names nor abbreviations are hard-wired into the server; they are obtained from configuration files stored under `.../share/timezone/` and `.../share/timezonesets/` of the installation directory (see Section B.4).

The TimeZone configuration parameter can be set in the file `postgresql.conf`, or in any of the other standard ways described in Chapter 19. There are also some special ways to set it:

- The SQL command `SET TIME ZONE` sets the time zone for the session. This is an alternative spelling of `SET TIMEZONE TO` with a more SQL-spec-compatible syntax.
- The `PGTZ` environment variable is used by libpq clients to send a `SET TIME ZONE` command to the server upon connection.

8.5.4. Interval Input

`interval` values can be written using the following verbose syntax:

```
[@] quantity unit [quantity unit...] [direction]
```

where *`quantity`* is a number (possibly signed); *`unit`* is `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; *`direction`* can be `ago` or empty. The at sign (`@`) is optional noise. The amounts of the different units are implicitly added with appropriate sign accounting. `ago` negates all the fields. This syntax is also used for interval output, if IntervalStyle is set to `postgres_verbose`.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, `'1 12:59:10'` is read the same as `'1 day 12 hours 59 min 10 sec'`. Also, a combination of years and months can be specified with a dash; for example `'200-10'` is read the same as `'200 years 10 months'`. (These shorter forms are in fact the only ones allowed by the SQL standard, and are used for output when `IntervalStyle` is set to `sql_standard`.)

Interval values can also be written as ISO 8601 time intervals, using either the “format with designators” of the standard's section 4.4.3.2 or the “alternative format” of section 4.4.3.3. The format with designators looks like this:

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

The string must start with a `P`, and may include a `T` that introduces the time-of-day units. The available unit abbreviations are given in Table 8.16. Units may be omitted, and may be specified in any order, but units smaller than a day must appear after `T`. In particular, the meaning of `M` depends on whether it is before or after `T`.

Table 8.16. ISO 8601 Interval Unit Abbreviations

Abbreviation	Meaning
Y	Years
M	Months (in the date part)
W	Weeks
D	Days
H	Hours
M	Minutes (in the time part)
S	Seconds

In the alternative format:

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

the string must begin with P, and a T separates the date and time parts of the interval. The values are given as numbers similar to ISO 8601 dates.

When writing an interval constant with a *fields* specification, or when assigning a string to an interval column that was defined with a *fields* specification, the interpretation of unmarked quantities depends on the *fields*. For example INTERVAL '1' YEAR is read as 1 year, whereas INTERVAL '1' means 1 second. Also, field values "to the right" of the least significant field allowed by the *fields* specification are silently discarded. For example, writing INTERVAL '1 day 2:03:04' HOUR TO MINUTE results in dropping the seconds field, but not the day field.

According to the SQL standard all fields of an interval value must have the same sign, so a leading negative sign applies to all fields; for example the negative sign in the interval literal '-1 2:03:04' applies to both the days and hour/minute/second parts. PostgreSQL allows the fields to have different signs, and traditionally treats each field in the textual representation as independently signed, so that the hour/minute/second part is considered positive in this example. If IntervalStyle is set to sql_standard then a leading sign is considered to apply to all fields (but only if no additional signs appear). Otherwise the traditional PostgreSQL interpretation is used. To avoid ambiguity, it's recommended to attach an explicit sign to each field if any field is negative.

Internally, interval values are stored as three integral fields: months, days, and microseconds. These fields are kept separate because the number of days in a month varies, while a day can have 23 or 25 hours if a daylight savings time transition is involved. An interval input string that uses other units is normalized into this format, and then reconstructed in a standardized way for output, for example:

```
SELECT '2 years 15 months 100 weeks 99 hours 123456789
 milliseconds'::interval;
               interval
-------------------------------------
 3 years 3 mons 700 days 133:17:36.789
```

Here weeks, which are understood as "7 days", have been kept separate, while the smaller and larger time units were combined and normalized.

Input field values can have fractional parts, for example '1.5 weeks' or '01:02:03.45'. However, because interval internally stores only integral fields, fractional values must be converted into smaller units. Fractional parts of units greater than months are rounded to be an integer number of months, e.g. '1.5 years' becomes '1 year 6 mons'. Fractional parts of weeks and days are computed to be an integer number of days and microseconds, assuming 30 days per month and 24 hours per day, e.g., '1.75 months' becomes 1 mon 22 days 12:00:00. Only seconds will ever be shown as fractional on output.

Table 8.17 shows some examples of valid interval input.

Table 8.17. Interval Input

Example	Description
1-2	SQL standard format: 1 year 2 months
3 4:05:06	SQL standard format: 3 days 4 hours 5 minutes 6 seconds
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Traditional Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds
P1Y2M3DT4H5M6S	ISO 8601 "format with designators": same meaning as above

Example	Description
P0001-02-03T04:05:06	ISO 8601 “alternative format”: same meaning as above

8.5.5. Interval Output

As previously explained, PostgreSQL stores interval values as months, days, and microseconds. For output, the months field is converted to years and months by dividing by 12. The days field is shown as-is. The microseconds field is converted to hours, minutes, seconds, and fractional seconds. Thus months, minutes, and seconds will never be shown as exceeding the ranges 0–11, 0–59, and 0–59 respectively, while the displayed years, days, and hours fields can be quite large. (The justify_days and justify_hours functions can be used if it is desirable to transpose large days or hours values into the next higher field.)

The output format of the interval type can be set to one of the four styles sql_standard, postgres, postgres_verbose, or iso_8601, using the command SET intervalstyle. The default is the postgres format. Table 8.18 shows examples of each output style.

The sql_standard style produces output that conforms to the SQL standard's specification for interval literal strings, if the interval value meets the standard's restrictions (either year-month only or day-time only, with no mixing of positive and negative components). Otherwise the output looks like a standard year-month literal string followed by a day-time literal string, with explicit signs added to disambiguate mixed-sign intervals.

The output of the postgres style matches the output of PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to ISO.

The output of the postgres_verbose style matches the output of PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to non-ISO output.

The output of the iso_8601 style matches the “format with designators” described in section 4.4.3.2 of the ISO 8601 standard.

Table 8.18. Interval Output Style Examples

Style Specification	Year-Month Interval	Day-Time Interval	Mixed Interval
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3D T-4H-5M-6S

8.6. Boolean Type

PostgreSQL provides the standard SQL type boolean; see Table 8.19. The boolean type can have several states: “true”, “false”, and a third state, “unknown”, which is represented by the SQL null value.

Table 8.19. Boolean Data Type

Name	Storage Size	Description
boolean	1 byte	state of true or false

Boolean constants can be represented in SQL queries by the SQL key words TRUE, FALSE, and NULL.

The datatype input function for type boolean accepts these string representations for the “true” state:

```
true
yes
on
1
```

and these representations for the “false” state:

```
false
no
off
0
```

Unique prefixes of these strings are also accepted, for example t or n. Leading or trailing whitespace is ignored, and case does not matter.

The datatype output function for type boolean always emits either t or f, as shown in Example 8.2.

Example 8.2. Using the boolean Type

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

The key words TRUE and FALSE are the preferred (SQL-compliant) method for writing Boolean constants in SQL queries. But you can also use the string representations by following the generic string-literal constant syntax described in Section 4.1.2.7, for example 'yes'::boolean.

Note that the parser automatically understands that TRUE and FALSE are of type boolean, but this is not so for NULL because that can have any type. So in some contexts you might have to cast NULL to boolean explicitly, for example NULL::boolean. Conversely, the cast can be omitted from a string-literal Boolean value in contexts where the parser can deduce that the literal must be of type boolean.

8.7. Enumerated Types

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the enum types supported in a number of programming languages. An example of an enum type might be the days of the week, or a set of status values for a piece of data.

8.7.1. Declaration of Enumerated Types

Enum types are created using the CREATE TYPE command, for example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Once created, the enum type can be used in table and function definitions much like any other type:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. Ordering

The ordering of the values in an enum type is the order in which the values were listed when the type was created. All standard comparison operators and related aggregate functions are supported for enums. For example:

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY
 current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

8.7.3. Type Safety

Each enumerated data type is separate and cannot be compared with other enumerated types. See this example:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
```

```
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

If you really need to do something like that, you can either write a custom operator or add explicit casts to your query:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. Implementation Details

Enum labels are case sensitive, so `'happy'` is not the same as `'HAPPY'`. White space in the labels is significant too.

Although enum types are primarily intended for static sets of values, there is support for adding new values to an existing enum type, and for renaming values (see ALTER TYPE). Existing values cannot be removed from an enum type, nor can the sort ordering of such values be changed, short of dropping and re-creating the enum type.

An enum value occupies four bytes on disk. The length of an enum value's textual label is limited by the `NAMEDATALEN` setting compiled into PostgreSQL; in standard builds this means at most 63 bytes.

The translations from internal enum values to textual labels are kept in the system catalog `pg_enum`. Querying this catalog directly can be useful.

8.8. Geometric Types

Geometric data types represent two-dimensional spatial objects. Table 8.20 shows the geometric types available in PostgreSQL.

Table 8.20. Geometric Types

Name	Storage Size	Description	Representation
`point`	16 bytes	Point on a plane	(x,y)
`line`	24 bytes	Infinite line	{A,B,C}
`lseg`	32 bytes	Finite line segment	[(x1,y1),(x2,y2)]
`box`	32 bytes	Rectangular box	(x1,y1),(x2,y2)
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`circle`	24 bytes	Circle	<(x,y),r> (center point and radius)

In all these types, the individual coordinates are stored as `double precision` (`float8`) numbers.

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections. They are explained in Section 9.11.

8.8.1. Points

Points are the fundamental two-dimensional building block for geometric types. Values of type `point` are specified using either of the following syntaxes:

```
( x , y )
  x , y
```

where *x* and *y* are the respective coordinates, as floating-point numbers.

Points are output using the first syntax.

8.8.2. Lines

Lines are represented by the linear equation $Ax + By + C = 0$, where *A* and *B* are not both zero. Values of type `line` are input and output in the following form:

```
{ A, B, C }
```

Alternatively, any of the following forms can be used for input:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are two different points on the line.

8.8.3. Line Segments

Line segments are represented by pairs of points that are the endpoints of the segment. Values of type `lseg` are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are the end points of the line segment.

Line segments are output using the first syntax.

8.8.4. Boxes

Boxes are represented by pairs of points that are opposite corners of the box. Values of type `box` are specified using any of the following syntaxes:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where (*x1*,*y1*) and (*x2*,*y2*) are any two opposite corners of the box.

Boxes are output using the second syntax.

Any two opposite corners can be supplied on input, but the values will be reordered as needed to store the upper right and lower left corners, in that order.

8.8.5. Paths

Paths are represented by lists of connected points. Paths can be *open*, where the first and last points in the list are considered not connected, or *closed*, where the first and last points are considered connected.

Values of type `path` are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the points are the end points of the line segments comprising the path. Square brackets (`[]`) indicate an open path, while parentheses (`()`) indicate a closed path. When the outermost parentheses are omitted, as in the third through fifth syntaxes, a closed path is assumed.

Paths are output using the first or second syntax, as appropriate.

8.8.6. Polygons

Polygons are represented by lists of points (the vertices of the polygon). Polygons are very similar to closed paths; the essential semantic difference is that a polygon is considered to include the area within it, while a path is not.

An important implementation difference between polygons and paths is that the stored representation of a polygon includes its smallest bounding box. This speeds up certain search operations, although computing the bounding box adds overhead while constructing new polygons.

Values of type `polygon` are specified using any of the following syntaxes:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

8.8.7. Circles

Circles are represented by a center point and radius. Values of type `circle` are specified using any of the following syntaxes:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where `(x,y)` is the center point and `r` is the radius of the circle.

Circles are output using the first syntax.

8.9. Network Address Types

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses, as shown in Table 8.21. It is better to use these types instead of plain text types to store network addresses, because these types offer input error checking and specialized operators and functions (see Section 9.12).

Table 8.21. Network Address Types

Name	Storage Size	Description
`cidr`	7 or 19 bytes	IPv4 and IPv6 networks
`inet`	7 or 19 bytes	IPv4 and IPv6 hosts and networks
`macaddr`	6 bytes	MAC addresses
`macaddr8`	8 bytes	MAC addresses (EUI-64 format)

When sorting `inet` or `cidr` data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped to IPv6 addresses, such as ::10.2.3.4 or ::ffff:10.4.3.2.

8.9.1. `inet`

The `inet` type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. The subnet is represented by the number of network address bits present in the host address (the “netmask”). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address. Note that if you want to accept only networks, you should use the `cidr` type rather than `inet`.

The input format for this type is *`address/y`* where *`address`* is an IPv4 or IPv6 address and *`y`* is the number of bits in the netmask. If the */y* portion is omitted, the netmask is taken to be 32 for IPv4 or 128 for IPv6, so the value represents just a single host. On display, the */y* portion is suppressed if the netmask specifies a single host.

8.9.2. `cidr`

The `cidr` type holds an IPv4 or IPv6 network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying networks is *`address/y`* where *`address`* is the network's lowest address represented as an IPv4 or IPv6 address, and *`y`* is the number of bits in the netmask. If *`y`* is omitted, it is calculated using assumptions from the older classful network numbering system, except it will be at least large enough to include all of the octets written in the input. It is an error to specify a network address that has bits set to the right of the specified netmask.

Table 8.22 shows some examples.

Table 8.22. `cidr` Type Input Examples

`cidr` Input	`cidr` Output	`abbrev(cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24

`cidr` Input	`cidr` Output	`abbrev(cidr)`
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba:2e0:81f-f:fe22:d1f1/128	2001:4f8:3:ba:2e0:81f-f:fe22:d1f1/128	2001:4f8:3:ba:2e0:81f-f:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. `inet` vs. `cidr`

The essential difference between `inet` and `cidr` data types is that `inet` accepts values with nonzero bits to the right of the netmask, whereas `cidr` does not. For example, `192.168.0.1/24` is valid for `inet` but not for `cidr`.

> **Tip**
>
> If you do not like the output format for `inet` or `cidr` values, try the functions `host`, `text`, and `abbrev`.

8.9.4. `macaddr`

The `macaddr` type stores MAC addresses, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in the following formats:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

These examples all specify the same address. Upper and lower case is accepted for the digits `a` through `f`. Output is always in the first of the forms shown.

IEEE Standard 802-2001 specifies the second form shown (with hyphens) as the canonical form for MAC addresses, and specifies the first form (with colons) as used with bit-reversed, MSB-first notation, so that 08-00-2b-01-02-03 = 10:00:D4:80:40:C0. This convention is widely ignored nowadays, and it is relevant only for obsolete network protocols (such as Token Ring). PostgreSQL makes no provisions for bit reversal; all accepted formats use the canonical LSB order.

The remaining five input formats are not part of any standard.

8.9.5. `macaddr8`

The `macaddr8` type stores MAC addresses in EUI-64 format, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). This type can accept both 6 and 8 byte length MAC addresses and stores them in 8 byte length format. MAC addresses given in 6 byte format will be stored in 8 byte length format with the 4th and 5th bytes set to FF and FE, respectively. Note that IPv6 uses a modified EUI-64 format where the 7th bit should be set to one after the conversion from EUI-48. The function `macaddr8_set7bit` is provided to make this change. Generally speaking, any input which is comprised of pairs of hex digits (on byte boundaries), optionally separated consistently by one of `':'`, `'-'` or `'.'`, is accepted. The number of hex digits must be either 16 (8 bytes) or 12 (6 bytes). Leading and trailing whitespace is ignored. The following are examples of input formats that are accepted:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

These examples all specify the same address. Upper and lower case is accepted for the digits `a` through `f`. Output is always in the first of the forms shown.

The last six input formats shown above are not part of any standard.

To convert a traditional 48 bit MAC address in EUI-48 format to modified EUI-64 format to be included as the host portion of an IPv6 address, use `macaddr8_set7bit` as shown:

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: `bit(`*n*`)` and `bit varying(`*n*`)`, where *n* is a positive integer.

`bit` type data must match the length *n* exactly; it is an error to attempt to store shorter or longer bit strings. `bit varying` data is of variable length up to the maximum length *n*; longer strings will be rejected. Writing `bit` without a length is equivalent to `bit(1)`, while `bit varying` without a length specification means unlimited length.

> **Note**
>
> If one explicitly casts a bit-string value to `bit(`*n*`)`, it will be truncated or zero-padded on the right to be exactly *n* bits, without raising an error. Similarly, if one explicitly casts a bit-string value to `bit varying(`*n*`)`, it will be truncated on the right if it is more than *n* bits.

Refer to Section 4.1.2.5 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Section 9.6.

Example 8.3. Using the Bit String Types

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

A bit string value requires 1 byte for each group of 8 bits, plus 5 or 8 bytes overhead depending on the length of the string (but long values may be compressed or moved out-of-line, as explained in Section 8.3 for character strings).

8.11. Text Search Types

PostgreSQL provides two data types that are designed to support full text search, which is the activity of searching through a collection of natural-language *documents* to locate those that best match a *query*. The `tsvector` type represents a document in a form optimized for text search; the `tsquery` type similarly represents a text query. Chapter 12 provides a detailed explanation of this facility, and Section 9.13 summarizes the related functions and operators.

8.11.1. `tsvector`

A `tsvector` value is a sorted list of distinct *lexemes*, which are words that have been *normalized* to merge different variants of the same word (see Chapter 12 for details). Sorting and duplicate-elimination are done automatically during input, as shown in this example:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

To represent lexemes containing whitespace or punctuation, surround them with quotes:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                  tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(We use dollar-quoted string literals in this example and the next one to avoid the confusion of having to double quote marks within the literals.) Embedded quotes and backslashes must be doubled:

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
-----------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Optionally, integer *positions* can be attached to lexemes:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10
 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------
------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5
 'rat':12 'sat':4
```

A position normally indicates the source word's location in the document. Positional information can be used for *proximity ranking*. Position values can range from 1 to 16383; larger numbers are silently set to 16383. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a *weight*, which can be `A`, `B`, `C`, or `D`. `D` is the default and hence is not shown on output:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Weights are typically used to reflect document structure, for example by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

It is important to understand that the `tsvector` type itself does not perform any word normalization; it assumes the words it is given are normalized appropriately for the application. For example,

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

For most English-text-searching applications the above words would be considered non-normalized, but `tsvector` doesn't care. Raw document text should usually be passed through `to_tsvector` to normalize the words appropriately for searching:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

Again, see Chapter 12 for more detail.

8.11.2. `tsquery`

A `tsquery` value stores lexemes that are to be searched for, and can combine them using the Boolean operators `&` (AND), `|` (OR), and `!` (NOT), as well as the phrase search operator `<->` (FOLLOWED BY). There is also a variant `<N>` of the FOLLOWED BY operator, where *N* is an integer constant that specifies the distance between the two lexemes being searched for. `<->` is equivalent to `<1>`.

Parentheses can be used to enforce grouping of these operators. In the absence of parentheses, `!` (NOT) binds most tightly, `<->` (FOLLOWED BY) next most tightly, then `&` (AND), with `|` (OR) binding the least tightly.

Here are some examples:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

Optionally, lexemes in a `tsquery` can be labeled with one or more weight letters, which restricts them to match only `tsvector` lexemes with one of those weights:

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Also, lexemes in a `tsquery` can be labeled with `*` to specify prefix matching:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

This query will match any word in a `tsvector` that begins with “super”.

Quoting rules for lexemes are the same as described previously for lexemes in `tsvector`; and, as with `tsvector`, any required normalization of words must be done before converting to the `tsquery` type. The `to_tsquery` function is convenient for performing such normalization:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

Note that `to_tsquery` will process prefixes in the same way as other words, which means this comparison returns true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

because `postgres` gets stemmed to `postgr`:

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
```

```
---------------+------------
 'postgradu':1 | 'postgr':*
```

which will match the stemmed form of `postgraduate`.

8.12. UUID Type

The data type `uuid` stores Universally Unique Identifiers (UUID) as defined by RFC 4122[2], ISO/IEC 9834-8:2005, and related standards. (Some systems refer to this data type as a globally unique identifier, or GUID, instead.) This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm. Therefore, for distributed systems, these identifiers provide a better uniqueness guarantee than sequence generators, which are only unique within a single database.

A UUID is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of 8 digits followed by three groups of 4 digits followed by a group of 12 digits, for a total of 32 digits representing the 128 bits. An example of a UUID in this standard form is:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL also accepts the following alternative forms for input: use of upper-case digits, the standard format surrounded by braces, omitting some or all hyphens, adding a hyphen after any group of four digits. Examples are:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Output is always in the standard form.

See Section 9.14 for how to generate a UUID in PostgreSQL.

8.13. XML Type

The `xml` data type can be used to store XML data. Its advantage over storing XML data in a `text` field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it; see Section 9.15. Use of this data type requires the installation to have been built with `configure --with-libxml`.

The `xml` type can store well-formed "documents", as defined by the XML standard, as well as "content" fragments, which are defined by reference to the more permissive "document node"[3] of the XQuery and XPath data model. Roughly, this means that content fragments can have more than one top-level element or character node. The expression *`xmlvalue`* `IS DOCUMENT` can be used to evaluate whether a particular `xml` value is a full document or only a content fragment.

Limits and compatibility notes for the `xml` data type can be found in Section D.3.

8.13.1. Creating XML Values

To produce a value of type `xml` from character data, use the function `xmlparse`:

[2] https://datatracker.ietf.org/doc/html/rfc4122
[3] https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Examples:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</
title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

While this is the only way to convert character strings into XML values according to the SQL standard, the PostgreSQL-specific syntaxes:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

can also be used.

The `xml` type does not validate input values against a document type declaration (DTD), even when the input value specifies a DTD. There is also currently no built-in support for validating against other XML schema languages such as XML Schema.

The inverse operation, producing a character string value from `xml`, uses the function `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type [ [ NO ]
 INDENT ] )
```

type can be `character`, `character varying`, or `text` (or an alias for one of those). Again, according to the SQL standard, this is the only way to convert between type `xml` and character types, but PostgreSQL also allows you to simply cast the value.

The `INDENT` option causes the result to be pretty-printed, while `NO INDENT` (which is the default) just emits the original input string. Casting to a character type likewise produces the original string.

When a character string value is cast to or from type `xml` without going through `XMLPARSE` or `XMLSERIALIZE`, respectively, the choice of `DOCUMENT` versus `CONTENT` is determined by the "XML option" session configuration parameter, which can be set using the standard command:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

or the more PostgreSQL-like syntax

```
SET xmloption TO { DOCUMENT | CONTENT };
```

The default is `CONTENT`, so all forms of XML data are allowed.

8.13.2. Encoding Handling

Care must be taken when dealing with multiple character encodings on the client, server, and in the XML data passed through them. When using the text mode to pass queries to the server and query results to the client (which is the normal mode), PostgreSQL converts all character data passed between the client and the server and vice versa to the character encoding of the respective end; see Section 23.3. This includes string representations of XML values, such as in the above examples. This would ordinarily mean that encoding declarations contained in XML data can become invalid as the character data is converted to other encodings while traveling between client and server, because the embedded encoding declaration is not changed. To cope with this behavior, encoding declarations contained in character strings presented for input to the `xml` type are *ignored*, and content is assumed

to be in the current server encoding. Consequently, for correct processing, character strings of XML data must be sent from the client in the current client encoding. It is the responsibility of the client to either convert documents to the current client encoding before sending them to the server, or to adjust the client encoding appropriately. On output, values of type `xml` will not have an encoding declaration, and clients should assume all data is in the current client encoding.

When using binary mode to pass query parameters to the server and query results back to the client, no encoding conversion is performed, so the situation is different. In this case, an encoding declaration in the XML data will be observed, and if it is absent, the data will be assumed to be in UTF-8 (as required by the XML standard; note that PostgreSQL does not support UTF-16). On output, data will have an encoding declaration specifying the client encoding, unless the client encoding is UTF-8, in which case it will be omitted.

Needless to say, processing XML data with PostgreSQL will be less error-prone and more efficient if the XML data encoding, client encoding, and server encoding are the same. Since XML data is internally processed in UTF-8, computations will be most efficient if the server encoding is also UTF-8.

> **Caution**
>
> Some XML-related functions may not work at all on non-ASCII data when the server encoding is not UTF-8. This is known to be an issue for `xmltable()` and `xpath()` in particular.

8.13.3. Accessing XML Values

The `xml` data type is unusual in that it does not provide any comparison operators. This is because there is no well-defined and universally useful comparison algorithm for XML data. One consequence of this is that you cannot retrieve rows by comparing an `xml` column against a search value. XML values should therefore typically be accompanied by a separate key field such as an ID. An alternative solution for comparing XML values is to convert them to character strings first, but note that character string comparison has little to do with a useful XML comparison method.

Since there are no comparison operators for the `xml` data type, it is not possible to create an index directly on a column of this type. If speedy searches in XML data are desired, possible workarounds include casting the expression to a character string type and indexing that, or indexing an XPath expression. Of course, the actual query would have to be adjusted to search by the indexed expression.

The text-search functionality in PostgreSQL can also be used to speed up full-document searches of XML data. The necessary preprocessing support is, however, not yet available in the PostgreSQL distribution.

8.14. JSON Types

JSON data types are for storing JSON (JavaScript Object Notation) data, as specified in RFC 7159[4]. Such data can also be stored as `text`, but the JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules. There are also assorted JSON-specific functions and operators available for data stored in these data types; see Section 9.16.

PostgreSQL offers two types for storing JSON data: `json` and `jsonb`. To implement efficient query mechanisms for these data types, PostgreSQL also provides the `jsonpath` data type described in Section 8.14.7.

The `json` and `jsonb` data types accept *almost* identical sets of values as input. The major practical difference is one of efficiency. The `json` data type stores an exact copy of the input text, which processing functions must reparse on each execution; while `jsonb` data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster

[4] https://datatracker.ietf.org/doc/html/rfc7159

to process, since no reparsing is needed. `jsonb` also supports indexing, which can be a significant advantage.

Because the `json` type stores an exact copy of the input text, it will preserve semantically-insignificant white space between tokens, as well as the order of keys within JSON objects. Also, if a JSON object within the value contains the same key more than once, all the key/value pairs are kept. (The processing functions consider the last value as the operative one.) By contrast, `jsonb` does not preserve white space, does not preserve the order of object keys, and does not keep duplicate object keys. If duplicate keys are specified in the input, only the last value is kept.

In general, most applications should prefer to store JSON data as `jsonb`, unless there are quite specialized needs, such as legacy assumptions about ordering of object keys.

RFC 7159 specifies that JSON strings should be encoded in UTF8. It is therefore not possible for the JSON types to conform rigidly to the JSON specification unless the database encoding is UTF8. Attempts to directly include characters that cannot be represented in the database encoding will fail; conversely, characters that can be represented in the database encoding but not in UTF8 will be allowed.

RFC 7159 permits JSON strings to contain Unicode escape sequences denoted by `\u`*`XXXX`*. In the input function for the `json` type, Unicode escapes are allowed regardless of the database encoding, and are checked only for syntactic correctness (that is, that four hex digits follow `\u`). However, the input function for `jsonb` is stricter: it disallows Unicode escapes for characters that cannot be represented in the database encoding. The `jsonb` type also rejects `\u0000` (because that cannot be represented in PostgreSQL's `text` type), and it insists that any use of Unicode surrogate pairs to designate characters outside the Unicode Basic Multilingual Plane be correct. Valid Unicode escapes are converted to the equivalent single character for storage; this includes folding surrogate pairs into a single character.

> **Note**
>
> Many of the JSON processing functions described in Section 9.16 will convert Unicode escapes to regular characters, and will therefore throw the same types of errors just described even if their input is of type `json` not `jsonb`. The fact that the `json` input function does not make these checks may be considered a historical artifact, although it does allow for simple storage (without processing) of JSON Unicode escapes in a database encoding that does not support the represented characters.

When converting textual JSON input into `jsonb`, the primitive types described by RFC 7159 are effectively mapped onto native PostgreSQL types, as shown in Table 8.23. Therefore, there are some minor additional constraints on what constitutes valid `jsonb` data that do not apply to the `json` type, nor to JSON in the abstract, corresponding to limits on what can be represented by the underlying data type. Notably, `jsonb` will reject numbers that are outside the range of the PostgreSQL `numeric` data type, while `json` will not. Such implementation-defined restrictions are permitted by RFC 7159. However, in practice such problems are far more likely to occur in other implementations, as it is common to represent JSON's `number` primitive type as IEEE 754 double precision floating point (which RFC 7159 explicitly anticipates and allows for). When using JSON as an interchange format with such systems, the danger of losing numeric precision compared to data originally stored by PostgreSQL should be considered.

Conversely, as noted in the table there are some minor restrictions on the input format of JSON primitive types that do not apply to the corresponding PostgreSQL types.

Table 8.23. JSON Primitive Types and Corresponding PostgreSQL Types

JSON primitive type	PostgreSQL type	Notes
`string`	`text`	`\u0000` is disallowed, as are Unicode escapes representing characters not available in the database encoding

JSON primitive type	PostgreSQL type	Notes
number	numeric	NaN and infinity values are disallowed
boolean	boolean	Only lowercase true and false spellings are accepted
null	(none)	SQL NULL is a different concept

8.14.1. JSON Input and Output Syntax

The input/output syntax for the JSON data types is as specified in RFC 7159.

The following are all valid json (or jsonb) expressions:

```
-- Simple scalar/primitive value
-- Primitive values can be numbers, quoted strings, true, false, or
 null
SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same
 type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

As previously stated, when a JSON value is input and then printed without any additional processing, json outputs the same text that was input, while jsonb does not preserve semantically-insignificant details such as whitespace. For example, note the differences here:

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                       json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                       jsonb
-------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

One semantically-insignificant detail worth noting is that in jsonb, numbers will be printed according to the behavior of the underlying numeric type. In practice this means that numbers entered with E notation will be printed without it, for example:

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading":
 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

However, `jsonb` will preserve trailing fractional zeroes, as seen in this example, even though those are semantically insignificant for purposes such as equality checks.

For the list of built-in functions and operators available for constructing and processing JSON values, see Section 9.16.

8.14.2. Designing JSON Documents

Representing data as JSON can be considerably more flexible than the traditional relational data model, which is compelling in environments where requirements are fluid. It is quite possible for both approaches to co-exist and complement each other within the same application. However, even for applications where maximal flexibility is desired, it is still recommended that JSON documents have a somewhat fixed structure. The structure is typically unenforced (though enforcing some business rules declaratively is possible), but having a predictable structure makes it easier to write queries that usefully summarize a set of "documents" (datums) in a table.

JSON data is subject to the same concurrency-control considerations as any other data type when stored in a table. Although storing large documents is practicable, keep in mind that any update acquires a row-level lock on the whole row. Consider limiting JSON documents to a manageable size in order to decrease lock contention among updating transactions. Ideally, JSON documents should each represent an atomic datum that business rules dictate cannot reasonably be further subdivided into smaller datums that could be modified independently.

8.14.3. `jsonb` Containment and Existence

Testing *containment* is an important capability of `jsonb`. There is no parallel set of facilities for the `json` type. Containment tests whether one `jsonb` document has contained within it another one. These examples return true except as noted:

```
-- Simple scalar/primitive values contain only the identical value:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- The array on the right side is contained within the one on the
 left:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Order of array elements is not significant, so this is also
 true:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- Duplicate array elements don't matter either:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- The object with a single pair on the right side is contained
-- within the object on the left side:
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb":
 true}'::jsonb @> '{"version": 9.4}'::jsonb;

-- The array on the right side is not considered contained within
 the
-- array on the left, even though a similar array is nested within
 it:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- yields false

-- But with a layer of nesting, it is contained:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;
```

```
-- Similarly, containment is not reported here:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;
  -- yields false

-- A top-level key and an empty object is contained:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

The general principle is that the contained object must match the containing object as to structure and data contents, possibly after discarding some non-matching array elements or object key/value pairs from the containing object. But remember that the order of array elements is not significant when doing a containment match, and duplicate array elements are effectively considered only once.

As a special exception to the general principle that the structures must match, an array may contain a primitive value:

```
-- This array contains the primitive string value:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- This exception is not reciprocal -- non-containment is reported
 here:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- yields false
```

`jsonb` also has an *existence* operator, which is a variation on the theme of containment: it tests whether a string (given as a `text` value) appears as an object key or array element at the top level of the `jsonb` value. These examples return true except as noted:

```
-- String exists as array element:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- String exists as object key:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Object values are not considered:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- yields false

-- As with containment, existence must match at the top level:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- yields false

-- A string is considered to exist if it matches a primitive JSON
 string:
SELECT '"foo"'::jsonb ? 'foo';
```

JSON objects are better suited than arrays for testing containment or existence when there are many keys or elements involved, because unlike arrays they are internally optimized for searching, and do not need to be searched linearly.

> **Tip**
>
> Because JSON containment is nested, an appropriate query can skip explicit selection of sub-objects. As an example, suppose that we have a `doc` column containing objects at the top level, with most objects containing `tags` fields that contain arrays of sub-objects. This query finds entries in which sub-objects containing both `"term":"paris"` and `"term":"food"` appear, while ignoring any such keys outside the `tags` array:
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
> ```

One could accomplish the same thing with, say,

```
SELECT doc->'site_name' FROM websites
  WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

but that approach is less flexible, and often less efficient as well.

On the other hand, the JSON existence operator is not nested: it will only look for the specified key or array element at top level of the JSON value.

The various containment and existence operators, along with all other JSON operators and functions are documented in Section 9.16.

8.14.4. `jsonb` Indexing

GIN indexes can be used to efficiently search for keys or key/value pairs occurring within a large number of `jsonb` documents (datums). Two GIN "operator classes" are provided, offering different performance and flexibility trade-offs.

The default GIN operator class for `jsonb` supports queries with the key-exists operators `?`, `?|` and `?&`, the containment operator `@>`, and the `jsonpath` match operators `@?` and `@@`. (For details of the semantics that these operators implement, see Table 9.46.) An example of creating an index with this operator class is:

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

The non-default GIN operator class `jsonb_path_ops` does not support the key-exists operators, but it does support `@>`, `@?` and `@@`. An example of creating an index with this operator class is:

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

Consider the example of a table that stores JSON documents retrieved from a third-party web service, with a documented schema definition. A typical document is:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

We store these documents in a table named `api`, in a `jsonb` column named `jdoc`. If a GIN index is created on this column, queries like the following can make use of the index:

```
-- Find documents in which the key "company" has value "Magnafone"
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @>
 '{"company": "Magnafone"}';
```

However, the index could not be used for queries like the following, because though the operator ? is indexable, it is not applied directly to the indexed column `jdoc`:

```
-- Find documents in which the key "tags" contains key or array
 element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ?
 'qui';
```

Still, with appropriate use of expression indexes, the above query can use an index. If querying for particular items within the `"tags"` key is common, defining an index like this may be worthwhile:

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

Now, the `WHERE` clause `jdoc -> 'tags' ? 'qui'` will be recognized as an application of the indexable operator ? to the indexed expression `jdoc -> 'tags'`. (More information on expression indexes can be found in Section 11.7.)

Another approach to querying is to exploit containment, for example:

```
-- Find documents in which the key "tags" contains array element
 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags":
 ["qui"]}';
```

A simple GIN index on the `jdoc` column can support this query. But note that such an index will store copies of every key and value in the `jdoc` column, whereas the expression index of the previous example stores only data found under the `tags` key. While the simple-index approach is far more flexible (since it supports queries about any key), targeted expression indexes are likely to be smaller and faster to search than a simple index.

GIN indexes also support the @? and @@ operators, which perform `jsonpath` matching. Examples are

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @?
 '$.tags[*] ? (@ == "qui")';
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*]
 == "qui"';
```

For these operators, a GIN index extracts clauses of the form *`accessors_chain == constant`* out of the `jsonpath` pattern, and does the index search based on the keys and values mentioned in these clauses. The accessors chain may include `.`*`key`*, `[*]`, and `[`*`index`*`]` accessors. The `jsonb_ops` operator class also supports `.*` and `.**` accessors, but the `jsonb_path_ops` operator class does not.

Although the `jsonb_path_ops` operator class supports only queries with the @>, @? and @@ operators, it has notable performance advantages over the default operator class `jsonb_ops`. A `jsonb_path_ops` index is usually much smaller than a `jsonb_ops` index over the same data, and the specificity of searches is better, particularly when queries contain keys that appear frequently in the data. Therefore search operations typically perform better than with the default operator class.

The technical difference between a `jsonb_ops` and a `jsonb_path_ops` GIN index is that the former creates independent index items for each key and value in the data, while the latter creates

index items only for each value in the data. [5] Basically, each `jsonb_path_ops` index item is a hash of the value and the key(s) leading to it; for example to index `{"foo": {"bar": "baz"}}`, a single index item would be created incorporating all three of `foo`, `bar`, and `baz` into the hash value. Thus a containment query looking for this structure would result in an extremely specific index search; but there is no way at all to find out whether `foo` appears as a key. On the other hand, a `jsonb_ops` index would create three index items representing `foo`, `bar`, and `baz` separately; then to do the containment query, it would look for rows containing all three of these items. While GIN indexes can perform such an AND search fairly efficiently, it will still be less specific and slower than the equivalent `jsonb_path_ops` search, especially if there are a very large number of rows containing any single one of the three index items.

A disadvantage of the `jsonb_path_ops` approach is that it produces no index entries for JSON structures not containing any values, such as `{"a": {}}`. If a search for documents containing such a structure is requested, it will require a full-index scan, which is quite slow. `jsonb_path_ops` is therefore ill-suited for applications that often perform such searches.

`jsonb` also supports `btree` and `hash` indexes. These are usually useful only if it's important to check equality of complete JSON documents. The `btree` ordering for `jsonb` datums is seldom of great interest, but for completeness it is:

```
Object > Array > Boolean > Number > String > null

Object with n pairs > object with n - 1 pairs

Array with n elements > array with n - 1 elements
```

with the exception that (for historical reasons) an empty top level array sorts less than `null`. Objects with equal numbers of pairs are compared in the order:

```
key-1, value-1, key-2 ...
```

Note that object keys are compared in their storage order; in particular, since shorter keys are stored before longer keys, this can lead to results that might be unintuitive, such as:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

Similarly, arrays with equal numbers of elements are compared in the order:

```
element-1, element-2 ...
```

Primitive JSON values are compared using the same comparison rules as for the underlying PostgreSQL data type. Strings are compared using the default database collation.

8.14.5. `jsonb` Subscripting

The `jsonb` data type supports array-style subscripting expressions to extract and modify elements. Nested values can be indicated by chaining subscripting expressions, following the same rules as the `path` argument in the `jsonb_set` function. If a `jsonb` value is an array, numeric subscripts start at zero, and negative integers count backwards from the last element of the array. Slice expressions are not supported. The result of a subscripting expression is always of the jsonb data type.

`UPDATE` statements may use subscripting in the `SET` clause to modify `jsonb` values. Subscript paths must be traversable for all affected values insofar as they exist. For instance, the path `val['a']['b']['c']` can be traversed all the way to `c` if every `val`, `val['a']`, and `val['a']['b']`

[5] For this purpose, the term “value” includes array elements, though JSON terminology sometimes considers array elements distinct from values within objects.

is an object. If any `val['a']` or `val['a']['b']` is not defined, it will be created as an empty object and filled as necessary. However, if any `val` itself or one of the intermediary values is defined as a non-object such as a string, number, or `jsonb null`, traversal cannot proceed so an error is raised and the transaction aborted.

An example of subscripting syntax:

```
-- Extract object value by key
SELECT ('{"a": 1}'::jsonb)['a'];

-- Extract nested object value by key path
SELECT ('{"a": {"b": {"c": 1}}}'::jsonb)['a']['b']['c'];

-- Extract array element by index
SELECT ('[1, "2", null]'::jsonb)[1];

-- Update object value by key. Note the quotes around '1': the
 assigned
-- value must be of the jsonb type as well
UPDATE table_name SET jsonb_field['key'] = '1';

-- This will raise an error if any record's jsonb_field['a']['b']
 is something
-- other than an object. For example, the value {"a": 1} has a
 numeric value
-- of the key 'a'.
UPDATE table_name SET jsonb_field['a']['b']['c'] = '1';

-- Filter records using a WHERE clause with subscripting. Since the
 result of
-- subscripting is jsonb, the value we compare it against must also
 be jsonb.
-- The double quotes make "value" also a valid jsonb string.
SELECT * FROM table_name WHERE jsonb_field['key'] = '"value"';
```

`jsonb` assignment via subscripting handles a few edge cases differently from `jsonb_set`. When a source `jsonb` value is `NULL`, assignment via subscripting will proceed as if it was an empty JSON value of the type (object or array) implied by the subscript key:

```
-- Where jsonb_field was NULL, it is now {"a": 1}
UPDATE table_name SET jsonb_field['a'] = '1';

-- Where jsonb_field was NULL, it is now [1]
UPDATE table_name SET jsonb_field[0] = '1';
```

If an index is specified for an array containing too few elements, `NULL` elements will be appended until the index is reachable and the value can be set.

```
-- Where jsonb_field was [], it is now [null, null, 2];
-- where jsonb_field was [0], it is now [0, null, 2]
UPDATE table_name SET jsonb_field[2] = '2';
```

A `jsonb` value will accept assignments to nonexistent subscript paths as long as the last existing element to be traversed is an object or array, as implied by the corresponding subscript (the element indicated by the last subscript in the path is not traversed and may be anything). Nested array and

object structures will be created, and in the former case `null`-padded, as specified by the subscript path until the assigned value can be placed.

```
-- Where jsonb_field was {}, it is now {"a": [{"b": 1}]}
UPDATE table_name SET jsonb_field['a'][0]['b'] = '1';

-- Where jsonb_field was [], it is now [null, {"a": 1}]
UPDATE table_name SET jsonb_field[1]['a'] = '1';
```

8.14.6. Transforms

Additional extensions are available that implement transforms for the `jsonb` type for different procedural languages.

The extensions for PL/Perl are called `jsonb_plperl` and `jsonb_plperlu`. If you use them, `jsonb` values are mapped to Perl arrays, hashes, and scalars, as appropriate.

The extension for PL/Python is called `jsonb_plpython3u`. If you use it, `jsonb` values are mapped to Python dictionaries, lists, and scalars, as appropriate.

Of these extensions, `jsonb_plperl` is considered "trusted", that is, it can be installed by non-superusers who have `CREATE` privilege on the current database. The rest require superuser privilege to install.

8.14.7. jsonpath Type

The `jsonpath` type implements support for the SQL/JSON path language in PostgreSQL to efficiently query JSON data. It provides a binary representation of the parsed SQL/JSON path expression that specifies the items to be retrieved by the path engine from the JSON data for further processing with the SQL/JSON query functions.

The semantics of SQL/JSON path predicates and operators generally follow SQL. At the same time, to provide a natural way of working with JSON data, SQL/JSON path syntax uses some JavaScript conventions:

- Dot (`.`) is used for member access.
- Square brackets (`[]`) are used for array access.
- SQL/JSON arrays are 0-relative, unlike regular SQL arrays that start from 1.

Numeric literals in SQL/JSON path expressions follow JavaScript rules, which are different from both SQL and JSON in some minor details. For example, SQL/JSON path allows `.1` and `1.`, which are invalid in JSON. Non-decimal integer literals and underscore separators are supported, for example, `1_000_000`, `0x1EEE_FFFF`, `0o273`, `0b100101`. In SQL/JSON path (and in JavaScript, but not in SQL proper), there must not be an underscore separator directly after the radix prefix.

An SQL/JSON path expression is typically written in an SQL query as an SQL character string literal, so it must be enclosed in single quotes, and any single quotes desired within the value must be doubled (see Section 4.1.2.1). Some forms of path expressions require string literals within them. These embedded string literals follow JavaScript/ECMAScript conventions: they must be surrounded by double quotes, and backslash escapes may be used within them to represent otherwise-hard-to-type characters. In particular, the way to write a double quote within an embedded string literal is `\"`, and to write a backslash itself, you must write `\\`. Other special backslash sequences include those recognized in JavaScript strings: `\b`, `\f`, `\n`, `\r`, `\t`, `\v` for various ASCII control characters, `\x`*`NN`* for a character code written with only two hex digits, `\u`*`NNNN`* for a Unicode character identified by its 4-hex-digit code point, and `\u{`*`N...`*`}` for a Unicode character code point written with 1 to 6 hex digits.

A path expression consists of a sequence of path elements, which can be any of the following:

- Path literals of JSON primitive types: Unicode text, numeric, true, false, or null.
- Path variables listed in Table 8.24.
- Accessor operators listed in Table 8.25.
- `jsonpath` operators and methods listed in Section 9.16.2.3.
- Parentheses, which can be used to provide filter expressions or define the order of path evaluation.

For details on using `jsonpath` expressions with SQL/JSON query functions, see Section 9.16.2.

Table 8.24. `jsonpath` Variables

Variable	Description
`$`	A variable representing the JSON value being queried (the *context item*).
`$varname`	A named variable. Its value can be set by the parameter *`vars`* of several JSON processing functions; see Table 9.49 for details.
`@`	A variable representing the result of path evaluation in filter expressions.

Table 8.25. `jsonpath` Accessors

Accessor Operator	Description
`.`*`key`* `."$`*`varname`*`"`	Member accessor that returns an object member with the specified key. If the key name matches some named variable starting with `$` or does not meet the JavaScript rules for an identifier, it must be enclosed in double quotes to make it a string literal.
`.*`	Wildcard member accessor that returns the values of all members located at the top level of the current object.
`.**`	Recursive wildcard member accessor that processes all levels of the JSON hierarchy of the current object and returns all the member values, regardless of their nesting level. This is a PostgreSQL extension of the SQL/JSON standard.

`.**{`*`level`*`}`
 	
`.**{`*`start_level`* `to` *`end_level`*`}`
	Like `.**`, but selects only the specified levels of the JSON hierarchy. Nesting levels are specified as integers. Level zero corresponds to the current object. To access the lowest nesting level, you can use the `last` keyword. This is a PostgreSQL extension of the SQL/JSON standard.

`[`*`subscript`*`, ...]`	Array element accessor. *`subscript`* can be given in two forms: *`index`* or *`start_index`* `to` *`end_index`*. The first form returns a single array element by its index. The second form returns an array slice by the range of indexes, including the elements that correspond to the provided *`start_index`* and *`end_index`*. The specified *`index`* can be an integer, as well as an expression returning a single numeric value, which is automatically cast to integer. Index zero corresponds to the first array element. You can also use the `last` keyword to denote the last array element, which is useful for handling arrays of unknown length.
`[*]`	Wildcard array element accessor that returns all array elements.

8.15. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in or user-defined base type, enum type, composite type, range type, or domain can be created.

8.15.1. Declaration of Array Types

To illustrate the use of array types, we create this table:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

As shown, an array data type is named by appending square brackets (`[]`) to the data type name of the array elements. The above command will create a table named `sal_emp` with a column of type `text` (`name`), a one-dimensional array of type `integer` (`pay_by_quarter`), which represents the employee's salary by quarter, and a two-dimensional array of `text` (`schedule`), which represents the employee's weekly schedule.

The syntax for `CREATE TABLE` allows the exact size of arrays to be specified, for example:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

However, the current implementation ignores any supplied array size limits, i.e., the behavior is the same as for arrays of unspecified length.

The current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring the array size or number of dimensions in `CREATE TABLE` is simply documentation; it does not affect run-time behavior.

An alternative syntax, which conforms to the SQL standard by using the keyword `ARRAY`, can be used for one-dimensional arrays. `pay_by_quarter` could have been defined as:

```
    pay_by_quarter  integer ARRAY[4],
```

Or, if no array size is to be specified:

```
    pay_by_quarter  integer ARRAY,
```

As before, however, PostgreSQL does not enforce the size restriction in any case.

8.15.2. Array Value Input

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. (If you know C, this is not unlike the C syntax for initializing structures.) You can put double quotes around any element value, and must do so if it contains commas or curly braces. (More details appear below.) Thus, the general format of an array constant is the following:

```
'{ val1 delim val2 delim ... }'
```

where *`delim`* is the delimiter character for the type, as recorded in its `pg_type` entry. Among the standard data types provided in the PostgreSQL distribution, all use a comma (`,`), except for type `box`

which uses a semicolon (;). Each *val* is either a constant of the array element type, or a subarray. An example of an array constant is:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

To set an element of an array constant to NULL, write NULL for the element value. (Any upper- or lower-case variant of NULL will do.) If you want an actual string value “NULL”, you must put double quotes around it.

(These kinds of array constants are actually only a special case of the generic type constants discussed in Section 4.1.2.7. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

Now we can show some INSERT statements:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

The result of the previous two inserts looks like this:

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                 schedule
-------+---------------------------
+-----------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},
{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},
{meeting,lunch}}
(2 rows)
```

Multidimensional arrays must have matching extents for each dimension. A mismatch causes an error, for example:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  malformed array literal: "{{"meeting", "lunch"},
 {"meeting"}}"
DETAIL:  Multidimensional arrays must have sub-arrays with matching
 dimensions.
```

The ARRAY constructor syntax can also be used:

```
INSERT INTO sal_emp
    VALUES ('Bill',
```

```
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Notice that the array elements are ordinary SQL constants or expressions; for instance, string literals are single quoted, instead of double quoted as they would be in an array literal. The ARRAY constructor syntax is discussed in more detail in Section 4.2.12.

8.15.3. Accessing Arrays

Now, we can run some queries on the table. First, we show how to access a single element of an array. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <>
 pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

The array subscript numbers are written within square brackets. By default PostgreSQL uses a one-based numbering convention for arrays, that is, an array of *n* elements starts with `array[1]` and ends with `array[`*n*`]`.

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing *`lower-bound`*`:`*`upper-bound`* for one or more array dimensions. For example, this query retrieves the first item on Bill's schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

If any dimension is written as a slice, i.e., contains a colon, then all dimensions are treated as slices. Any dimension that has only a single number (no colon) is treated as being from 1 to the number specified. For example, `[2]` is treated as `[1:2]`, as in this example:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';
```

```
                schedule
-------------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

To avoid confusion with the non-slice case, it's best to use slice syntax for all dimensions, e.g., `[1:2][1:1]`, not `[2][1:1]`.

It is possible to omit the *`lower-bound`* and/or *`upper-bound`* of a slice specifier; the missing bound is replaced by the lower or upper limit of the array's subscripts. For example:

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

An array subscript expression will return null if either the array itself or any of the subscript expressions are null. Also, null is returned if a subscript is outside the array bounds (this case does not raise an error). For example, if `schedule` currently has the dimensions `[1:3][1:2]` then referencing `schedule[3][3]` yields NULL. Similarly, an array reference with the wrong number of subscripts yields a null rather than an error.

An array slice expression likewise yields null if the array itself or any of the subscript expressions are null. However, in other cases such as selecting an array slice that is completely outside the current array bounds, a slice expression yields an empty (zero-dimensional) array instead of null. (This does not match non-slice behavior and is done for historical reasons.) If the requested slice partially overlaps the array bounds, then it is silently reduced to just the overlapping region instead of returning null.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps inconvenient for programs. Dimensions can also be retrieved with `array_upper` and `array_lower`, which return the upper and lower bound of a specified array dimension, respectively:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` will return the length of a specified array dimension:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

`cardinality` returns the total number of elements in an array across all dimensions. It is effectively the number of rows a call to `unnest` would yield:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

8.15.4. Modifying Arrays

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or using the ARRAY expression syntax:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

An array can also be updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

The slice syntaxes with omitted *`lower-bound`* and/or *`upper-bound`* can be used too, but only when updating an array value that is not NULL or zero-dimensional (otherwise, there is no existing subscript limit to substitute).

A stored array value can be enlarged by assigning to elements not already present. Any positions between those previously present and the newly assigned elements will be filled with nulls. For example, if array `myarray` currently has 4 elements, it will have six elements after an update that assigns to `myarray[6]`; `myarray[5]` will contain null. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Subscripted assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to `myarray[-2:7]` to create an array with subscript values from -2 to 7.

New array values can also be constructed using the concatenation operator, `||`:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

The concatenation operator allows a single element to be pushed onto the beginning or end of a one-dimensional array. It also accepts two *N*-dimensional arrays, or an *N*-dimensional and an *N+1*-dimensional array.

When a single element is pushed onto either the beginning or end of a one-dimensional array, the result is an array with the same lower bound subscript as the array operand. For example:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

When two arrays with an equal number of dimensions are concatenated, the result retains the lower bound subscript of the left-hand operand's outer dimension. The result is an array comprising every element of the left-hand operand followed by every element of the right-hand operand. For example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

When an *N*-dimensional array is pushed onto the beginning or end of an *N+1*-dimensional array, the result is analogous to the element-array case above. Each *N*-dimensional sub-array is essentially an element of the *N+1*-dimensional array's outer dimension. For example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
```

```
(1 row)
```

An array can also be constructed by using the functions `array_prepend`, `array_append`, or `array_cat`. The first two only support one-dimensional arrays, but `array_cat` supports multidimensional arrays. Some examples:

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

In simple cases, the concatenation operator discussed above is preferred over direct use of these functions. However, because the concatenation operator is overloaded to serve all three cases, there are situations where use of one of the functions is helpful to avoid ambiguity. For example consider:

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- the untyped literal is taken as
 an array
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                -- so is this one
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;               -- so is an undecorated
 NULL
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);    -- this might have been
 meant
```

```
 array_append
--------------
 {1,2,NULL}
```

In the examples above, the parser sees an integer array on one side of the concatenation operator, and a constant of undetermined type on the other. The heuristic it uses to resolve the constant's type is to assume it's of the same type as the operator's other input — in this case, integer array. So the concatenation operator is presumed to represent `array_cat`, not `array_append`. When that's the wrong choice, it could be fixed by casting the constant to the array's element type; but explicit use of `array_append` might be a preferable solution.

8.15.5. Searching in Arrays

To search for a value in an array, each value must be checked. This can be done manually, if you know the size of the array. For example:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. An alternative method is described in Section 9.25. The above query could be replaced by:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

In addition, you can find rows where the array has all values equal to 10000 with:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Alternatively, the `generate_subscripts` function can be used. For example:

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

This function is described in Table 9.68.

You can also search an array using the && operator, which checks whether the left operand overlaps with the right operand. For instance:

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

This and other array operators are further described in Section 9.19. It can be accelerated by an appropriate index, as described in Section 11.2.

You can also search for specific values in an array using the `array_position` and `array_positions` functions. The former returns the subscript of the first occurrence of a value in an array; the latter returns an array with the subscripts of all occurrences of the value in the array. For example:

```
SELECT
 array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'],
 'mon');
```

```
 array_position
----------------
              2
(1 row)

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
(1 row)
```

> **Tip**
>
> Arrays are not sets; searching for specific array elements can be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale better for a large number of elements.

8.15.6. Array Input and Output Syntax

The external text representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array's element type, plus decoration that indicates the array structure. The decoration consists of curly braces (`{` and `}`) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (`,`) but can be something else: it is determined by the `typdelim` setting for the array's element type. Among the standard data types provided in the PostgreSQL distribution, all use a comma, except for type `box`, which uses a semicolon (`;`). In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level.

The array output routine will put double quotes around element values if they are empty strings, contain curly braces, delimiter characters, double quotes, backslashes, or white space, or match the word `NULL`. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric data types it is safe to assume that double quotes will never appear, but for textual data types one should be prepared to cope with either the presence or absence of quotes.

By default, the lower bound index value of an array's dimensions is set to one. To represent arrays with other lower bounds, the array subscript ranges can be specified explicitly before writing the array contents. This decoration consists of square brackets (`[]`) around each array dimension's lower and upper bounds, with a colon (`:`) delimiter character in between. The array dimension decoration is followed by an equal sign (`=`). For example:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1)
 AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

The array output routine will include explicit dimensions in its result only when there are one or more lower bounds different from one.

If the value written for an element is `NULL` (in any case variant), the element is taken to be NULL. The presence of any quotes or backslashes disables this and allows the literal string value “NULL” to be entered. Also, for backward compatibility with pre-8.2 versions of PostgreSQL, the array_nulls configuration parameter can be turned `off` to suppress recognition of `NULL` as a NULL.

As shown previously, when writing an array value you can use double quotes around any individual array element. You *must* do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or the data type's delimiter character), double quotes, backslashes, or leading or trailing whitespace must be double-quoted. Empty strings and strings matching the word NULL must be quoted, too. To put a double quote or backslash in a quoted array element value, precede it with a backslash. Alternatively, you can avoid quotes and use backslash-escaping to protect all data characters that would otherwise be taken as array syntax.

You can add whitespace before a left brace or after a right brace. You can also add whitespace before or after any individual item string. In all of these cases the whitespace will be ignored. However, whitespace within double-quoted elements, or surrounded on both sides by non-whitespace characters of an element, is not ignored.

> **Tip**
>
> The ARRAY constructor syntax (see Section 4.2.12) is often easier to work with than the array-literal syntax when writing array values in SQL commands. In ARRAY, individual element values are written the same way they would be written when not members of an array.

8.16. Composite Types

A *composite type* represents the structure of a row or record; it is essentially just a list of field names and their data types. PostgreSQL allows composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

8.16.1. Declaration of Composite Types

Here are two simple examples of defining composite types:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

The syntax is comparable to CREATE TABLE, except that only field names and types can be specified; no constraints (such as NOT NULL) can presently be included. Note that the AS keyword is essential; without it, the system will think a different kind of CREATE TYPE command is meant, and you will get odd syntax errors.

Having defined the types, we can use them to create tables:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

or functions:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS
 numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Whenever you create a table, a composite type is also automatically created, with the same name as the table, to represent the table's row type. For example, had we said:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

then the same `inventory_item` composite type shown above would come into being as a byproduct, and could be used just as above. Note however an important restriction of the current implementation: since no constraints are associated with a composite type, the constraints shown in the table definition *do not apply* to values of the composite type outside the table. (To work around this, create a *domain* over the composite type, and apply the desired constraints as `CHECK` constraints of the domain.)

8.16.2. Constructing Composite Values

To write a composite value as a literal constant, enclose the field values within parentheses and separate them by commas. You can put double quotes around any field value, and must do so if it contains commas or parentheses. (More details appear below.) Thus, the general format of a composite constant is the following:

```
'( val1 , val2 , ... )'
```

An example is:

```
'("fuzzy dice",42,1.99)'
```

which would be a valid value of the `inventory_item` type defined above. To make a field be NULL, write no characters at all in its position in the list. For example, this constant specifies a NULL third field:

```
'("fuzzy dice",42,)'
```

If you want an empty string rather than NULL, write double quotes:

```
'("",42,)'
```

Here the first field is a non-NULL empty string, the third is NULL.

(These constants are actually only a special case of the generic type constants discussed in Section 4.1.2.7. The constant is initially treated as a string and passed to the composite-type input conversion routine. An explicit type specification might be necessary to tell which type to convert the constant to.)

The ROW expression syntax can also be used to construct composite values. In most cases this is considerably simpler to use than the string-literal syntax since you don't have to worry about multiple layers of quoting. We already used this method above:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

The ROW keyword is actually optional as long as you have more than one field in the expression, so these can be simplified to:

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

The ROW expression syntax is discussed in more detail in Section 4.2.13.

8.16.3. Accessing Composite Types

To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name. In fact, it's so much like selecting from a table name that you often have to use parentheses to keep from confusing the parser. For example, you might try to select some subfields from our on_hand example table with something like:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

This will not work since the name item is taken to be a table name, not a column name of on_hand, per SQL syntax rules. You must write it like this:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

or if you need to use the table name as well (for instance in a multitable query), like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price
 > 9.99;
```

Now the parenthesized object is correctly interpreted as a reference to the item column, and then the subfield can be selected from it.

Similar syntactic issues apply whenever you select a field from a composite value. For instance, to select just one field from the result of a function that returns a composite value, you'd need to write something like:

```
SELECT (my_func(...)).field FROM ...
```

Without the extra parentheses, this will generate a syntax error.

The special field name * means “all fields”, as further explained in Section 8.16.5.

8.16.4. Modifying Composite Types

Here are some examples of the proper syntax for inserting and updating composite columns. First, inserting or updating a whole column:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));
```

```
UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

The first example omits `ROW`, the second uses it; we could have done it either way.

We can update an individual subfield of a composite column:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Notice here that we don't need to (and indeed cannot) put parentheses around the column name appearing just after `SET`, but we do need parentheses when referencing the same column in the expression to the right of the equal sign.

And we can specify subfields as targets for `INSERT`, too:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Had we not supplied values for all the subfields of the column, the remaining subfields would have been filled with null values.

8.16.5. Using Composite Types in Queries

There are various special syntax rules and behaviors associated with composite types in queries. These rules provide useful shortcuts, but can be confusing if you don't know the logic behind them.

In PostgreSQL, a reference to a table name (or alias) in a query is effectively a reference to the composite value of the table's current row. For example, if we had a table `inventory_item` as shown above, we could write:

```
SELECT c FROM inventory_item c;
```

This query produces a single composite-valued column, so we might get output like:

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

Note however that simple names are matched to column names before table names, so this example works only because there is no column named `c` in the query's tables.

The ordinary qualified-column-name syntax *`table_name.column_name`* can be understood as applying field selection to the composite value of the table's current row. (For efficiency reasons, it's not actually implemented that way.)

When we write

```
SELECT c.* FROM inventory_item c;
```

then, according to the SQL standard, we should get the contents of the table expanded into separate columns:

```
    name    | supplier_id | price
------------+-------------+-------
```

```
 fuzzy dice |          42 |  1.99
(1 row)
```

as if the query were

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

PostgreSQL will apply this expansion behavior to any composite-valued expression, although as shown above, you need to write parentheses around the value that `.*` is applied to whenever it's not a simple table name. For example, if `myfunc()` is a function returning a composite type with columns `a`, `b`, and `c`, then these two queries have the same result:

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

> **Tip**
>
> PostgreSQL handles column expansion by actually transforming the first form into the second. So, in this example, `myfunc()` would get invoked three times per row with either syntax. If it's an expensive function you may wish to avoid that, which you can do with a query like:
>
> ```
> SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
> ```
>
> Placing the function in a `LATERAL FROM` item keeps it from being invoked more than once per row. `m.*` is still expanded into `m.a, m.b, m.c`, but now those variables are just references to the output of the `FROM` item. (The `LATERAL` keyword is optional here, but we show it to clarify that the function is getting `x` from `some_table`.)

The *`composite_value`*`.*` syntax results in column expansion of this kind when it appears at the top level of a `SELECT` output list, a `RETURNING` list in `INSERT`/`UPDATE`/`DELETE`/`MERGE`, a `VALUES` clause, or a row constructor. In all other contexts (including when nested inside one of those constructs), attaching `.*` to a composite value does not change the value, since it means "all columns" and so the same composite value is produced again. For example, if `somefunc()` accepts a composite-valued argument, these queries are the same:

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

In both cases, the current row of `inventory_item` is passed to the function as a single composite-valued argument. Even though `.*` does nothing in such cases, using it is good style, since it makes clear that a composite value is intended. In particular, the parser will consider `c` in `c.*` to refer to a table name or alias, not to a column name, so that there is no ambiguity; whereas without `.*`, it is not clear whether `c` means a table name or a column name, and in fact the column-name interpretation will be preferred if there is a column named `c`.

Another example demonstrating these concepts is that all these queries mean the same thing:

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

All of these `ORDER BY` clauses specify the row's composite value, resulting in sorting the rows according to the rules described in Section 9.25.6. However, if `inventory_item` contained a column

named `c`, the first case would be different from the others, as it would mean to sort by that column only. Given the column names previously shown, these queries are also equivalent to those above:

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id,
 c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id,
 c.price);
```

(The last case uses a row constructor with the key word `ROW` omitted.)

Another special syntactical behavior associated with composite values is that we can use *functional notation* for extracting a field of a composite value. The simple way to explain this is that the notations *`field`*`(`*`table`*`)` and *`table`*`.`*`field`* are interchangeable. For example, these queries are equivalent:

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

Moreover, if we have a function that accepts a single argument of a composite type, we can call it with either notation. These queries are all equivalent:

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

This equivalence between functional notation and field notation makes it possible to use functions on composite types to implement “computed fields”. An application using the last query above wouldn't need to be directly aware that `somefunc` isn't a real column of the table.

> **Tip**
>
> Because of this behavior, it's unwise to give a function that takes a single composite-type argument the same name as any of the fields of that composite type. If there is ambiguity, the field-name interpretation will be chosen if field-name syntax is used, while the function will be chosen if function-call syntax is used. However, PostgreSQL versions before 11 always chose the field-name interpretation, unless the syntax of the call required it to be a function call. One way to force the function interpretation in older versions is to schema-qualify the function name, that is, write *`schema`*`.`*`func`*`(`*`compositevalue`*`)`.

8.16.6. Composite Type Input and Output Syntax

The external text representation of a composite value consists of items that are interpreted according to the I/O conversion rules for the individual field types, plus decoration that indicates the composite structure. The decoration consists of parentheses (`(` and `)`) around the whole value, plus commas (`,`) between adjacent items. Whitespace outside the parentheses is ignored, but within the parentheses it is considered part of the field value, and might or might not be significant depending on the input conversion rules for the field data type. For example, in:

```
'(  42)'
```

the whitespace will be ignored if the field type is integer, but not if it is text.

As shown previously, when writing a composite value you can write double quotes around any individual field value. You *must* do so if the field value would otherwise confuse the composite-value

parser. In particular, fields containing parentheses, commas, double quotes, or backslashes must be double-quoted. To put a double quote or backslash in a quoted composite field value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted field value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as composite syntax.

A completely empty field value (no characters at all between the commas or parentheses) represents a NULL. To write a value that is an empty string rather than NULL, write `""`.

The composite output routine will put double quotes around field values if they are empty strings or contain parentheses, commas, double quotes, backslashes, or white space. (Doing so for white space is not essential, but aids legibility.) Double quotes and backslashes embedded in field values will be doubled.

> **Note**
>
> Remember that what you write in an SQL command will first be interpreted as a string literal, and then as a composite. This doubles the number of backslashes you need (assuming escape string syntax is used). For example, to insert a `text` field containing a double quote and a backslash in a composite value, you'd need to write:
>
> ```
> INSERT ... VALUES ('("\"\\")');
> ```
>
> The string-literal processor removes one level of backslashes, so that what arrives at the composite-value parser looks like `("\"\\")`. In turn, the string fed to the `text` data type's input routine becomes `"\`. (If we were working with a data type whose input routine also treated backslashes specially, `bytea` for example, we might need as many as eight backslashes in the command to get one backslash into the stored composite field.) Dollar quoting (see Section 4.1.2.4) can be used to avoid the need to double backslashes.

> **Tip**
>
> The `ROW` constructor syntax is usually easier to work with than the composite-literal syntax when writing composite values in SQL commands. In `ROW`, individual field values are written the same way they would be written when not members of a composite.

8.17. Range Types

Range types are data types representing a range of values of some element type (called the range's *subtype*). For instance, ranges of `timestamp` might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is `tsrange` (short for “timestamp range”), and `timestamp` is the subtype. The subtype must have a total order so that it is well-defined whether element values are within, before, or after a range of values.

Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly. The use of time and date ranges for scheduling purposes is the clearest example; but price ranges, measurement ranges from an instrument, and so forth can also be useful.

Every range type has a corresponding multirange type. A multirange is an ordered list of non-contiguous, non-empty, non-null ranges. Most range operators also work on multiranges, and they have a few functions of their own.

8.17.1. Built-in Range and Multirange Types

PostgreSQL comes with the following built-in range types:

- `int4range` — Range of `integer`, `int4multirange` — corresponding Multirange
- `int8range` — Range of `bigint`, `int8multirange` — corresponding Multirange
- `numrange` — Range of `numeric`, `nummultirange` — corresponding Multirange
- `tsrange` — Range of `timestamp without time zone`, `tsmultirange` — corresponding Multirange
- `tstzrange` — Range of `timestamp with time zone`, `tstzmultirange` — corresponding Multirange
- `daterange` — Range of `date`, `datemultirange` — corresponding Multirange

In addition, you can define your own range types; see CREATE TYPE for more information.

8.17.2. Examples

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
```

See Table 9.56 and Table 9.58 for complete lists of operators and functions on range types.

8.17.3. Inclusive and Exclusive Bounds

Every non-empty range has two bounds, the lower bound and the upper bound. All points between these values are included in the range. An inclusive bound means that the boundary point itself is included in the range as well, while an exclusive bound means that the boundary point is not included in the range.

In the text form of a range, an inclusive lower bound is represented by "`[`" while an exclusive lower bound is represented by "`(`". Likewise, an inclusive upper bound is represented by "`]`", while an exclusive upper bound is represented by "`)`". (See Section 8.17.5 for more details.)

The functions `lower_inc` and `upper_inc` test the inclusivity of the lower and upper bounds of a range value, respectively.

8.17.4. Infinite (Unbounded) Ranges

The lower bound of a range can be omitted, meaning that all values less than the upper bound are included in the range, e.g., `(,3]`. Likewise, if the upper bound of the range is omitted, then all values greater than the lower bound are included in the range. If both lower and upper bounds are omitted, all values of the element type are considered to be in the range. Specifying a missing bound as inclusive is automatically converted to exclusive, e.g., `[,]` is converted to `(,)`. You can think of these missing values as +/-infinity, but they are special range type values and are considered to be beyond any range element type's +/-infinity values.

Element types that have the notion of "infinity" can use them as explicit bound values. For example, with timestamp ranges, `[today,infinity)` excludes the special `timestamp` value `infinity`, while `[today,infinity]` include it, as does `[today,)` and `[today,]`.

The functions `lower_inf` and `upper_inf` test for infinite lower and upper bounds of a range, respectively.

8.17.5. Range Input/Output

The input for a range value must follow one of the following patterns:

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

The parentheses or brackets indicate whether the lower and upper bounds are exclusive or inclusive, as described previously. Notice that the final pattern is `empty`, which represents an empty range (a range that contains no points).

The *`lower-bound`* may be either a string that is valid input for the subtype, or empty to indicate no lower bound. Likewise, *`upper-bound`* may be either a string that is valid input for the subtype, or empty to indicate no upper bound.

Each bound value can be quoted using `"` (double quote) characters. This is necessary if the bound value contains parentheses, brackets, commas, double quotes, or backslashes, since these characters would otherwise be taken as part of the range syntax. To put a double quote or backslash in a quoted bound value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted bound value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as range syntax. Also, to write a bound value that is an empty string, write `""`, since writing nothing means an infinite bound.

Whitespace is allowed before and after the range value, but any whitespace between the parentheses or brackets is taken as part of the lower or upper bound value. (Depending on the element type, it might or might not be significant.)

> **Note**
>
> These rules are very similar to those for writing field values in composite-type literals. See Section 8.16.6 for additional commentary.

Examples:

```
-- includes 3, does not include 7, and does include all points in
 between
SELECT '[3,7)'::int4range;

-- does not include either 3 or 7, but includes all points in
 between
SELECT '(3,7)'::int4range;

-- includes only the single point 4
SELECT '[4,4]'::int4range;

-- includes no points (and will be normalized to 'empty')
SELECT '[4,4)'::int4range;
```

The input for a multirange is curly brackets (`{` and `}`) containing zero or more valid ranges, separated by commas. Whitespace is permitted around the brackets and commas. This is intended to be reminiscent of array syntax, although multiranges are much simpler: they have just one dimension and there is no need to quote their contents. (The bounds of their ranges may be quoted as above however.)

Examples:

```
SELECT '{}'::int4multirange;
SELECT '{[3,7)}'::int4multirange;
SELECT '{[3,7), [8,9)}'::int4multirange;
```

8.17.6. Constructing Ranges and Multiranges

Each range type has a constructor function with the same name as the range type. Using the constructor function is frequently more convenient than writing a range literal constant, since it avoids the need for extra quoting of the bound values. The constructor function accepts two or three arguments. The two-argument form constructs a range in standard form (lower bound inclusive, upper bound exclusive), while the three-argument form constructs a range with bounds of the form specified by the third argument. The third argument must be one of the strings "`()`", "`(]`", "`[)`", or "`[]`". For example:

```
-- The full form is: lower bound, upper bound, and text argument
 indicating
-- inclusivity/exclusivity of bounds.
SELECT numrange(1.0, 14.0, '(]');

-- If the third argument is omitted, '[)' is assumed.
SELECT numrange(1.0, 14.0);

-- Although '(]' is specified here, on display the value will be
 converted to
-- canonical form, since int8range is a discrete range type (see
 below).
SELECT int8range(1, 14, '(]');

-- Using NULL for either bound causes the range to be unbounded on
 that side.
SELECT numrange(NULL, 2.2);
```

Each range type also has a multirange constructor with the same name as the multirange type. The constructor function takes zero or more arguments which are all ranges of the appropriate type. For example:

```
SELECT nummultirange();
SELECT nummultirange(numrange(1.0, 14.0));
SELECT nummultirange(numrange(1.0, 14.0), numrange(20.0, 25.0));
```

8.17.7. Discrete Range Types

A discrete range is one whose element type has a well-defined “step”, such as `integer` or `date`. In these types two elements can be said to be adjacent, when there are no valid values between them. This contrasts with continuous ranges, where it's always (or almost always) possible to identify other element values between two given values. For example, a range over the `numeric` type is continuous, as is a range over `timestamp`. (Even though `timestamp` has limited precision, and so could theoretically be treated as discrete, it's better to consider it continuous since the step size is normally not of interest.)

Another way to think about a discrete range type is that there is a clear idea of a “next” or “previous” value for each element value. Knowing that, it is possible to convert between inclusive and exclusive representations of a range's bounds, by choosing the next or previous element value instead of the one originally given. For example, in an integer range type `[4,8]` and `(3,9)` denote the same set of values; but this would not be so for a range over numeric.

A discrete range type should have a *canonicalization* function that is aware of the desired step size for the element type. The canonicalization function is charged with converting equivalent values of the range type to have identical representations, in particular consistently inclusive or exclusive bounds. If a canonicalization function is not specified, then ranges with different formatting will always be treated as unequal, even though they might represent the same set of values in reality.

The built-in range types `int4range`, `int8range`, and `daterange` all use a canonical form that includes the lower bound and excludes the upper bound; that is, `[)`. User-defined range types can use other conventions, however.

8.17.8. Defining New Range Types

Users can define their own range types. The most common reason to do this is to use ranges over subtypes not provided among the built-in range types. For example, to define a new range type of subtype `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Because `float8` has no meaningful “step”, we do not define a canonicalization function in this example.

When you define your own range you automatically get a corresponding multirange type.

Defining your own range type also allows you to specify a different subtype B-tree operator class or collation to use, so as to change the sort ordering that determines which values fall into a given range.

If the subtype is considered to have discrete rather than continuous values, the `CREATE TYPE` command should specify a `canonical` function. The canonicalization function takes an input range value, and must return an equivalent range value that may have different bounds and formatting. The canonical output for two ranges that represent the same set of values, for example the integer ranges `[1, 7]` and `[1, 8)`, must be identical. It doesn't matter which representation you choose to be the canonical one, so long as two equivalent values with different formattings are always mapped to the same value with the same formatting. In addition to adjusting the inclusive/exclusive bounds format, a

canonicalization function might round off boundary values, in case the desired step size is larger than what the subtype is capable of storing. For instance, a range type over `timestamp` could be defined to have a step size of an hour, in which case the canonicalization function would need to round off bounds that weren't a multiple of an hour, or perhaps throw an error instead.

In addition, any range type that is meant to be used with GiST or SP-GiST indexes should define a subtype difference, or `subtype_diff`, function. (The index will still work without `subtype_diff`, but it is likely to be considerably less efficient than if a difference function is provided.) The subtype difference function takes two input values of the subtype, and returns their difference (i.e., *X* minus *Y*) represented as a `float8` value. In our example above, the function `float8mi` that underlies the regular `float8` minus operator can be used; but for any other subtype, some type conversion would be necessary. Some creative thought about how to represent differences as numbers might be needed, too. To the greatest extent possible, the `subtype_diff` function should agree with the sort ordering implied by the selected operator class and collation; that is, its result should be positive whenever its first argument is greater than its second according to the sort ordering.

A less-oversimplified example of a `subtype_diff` function is:

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

See CREATE TYPE for more information about creating range types.

8.17.9. Indexing

GiST and SP-GiST indexes can be created for table columns of range types. GiST indexes can be also created for table columns of multirange types. For instance, to create a GiST index:

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

A GiST or SP-GiST index on ranges can accelerate queries involving these range operators: =, &&, <@, @>, <<, >>, -|-, &<, and &>. A GiST index on multiranges can accelerate queries involving the same set of multirange operators. A GiST index on ranges and GiST index on multiranges can also accelerate queries involving these cross-type range to multirange and multirange to range operators correspondingly: &&, <@, @>, <<, >>, -|-, &<, and &>. See Table 9.56 for more information.

In addition, B-tree and hash indexes can be created for table columns of range types. For these index types, basically the only useful range operation is equality. There is a B-tree sort ordering defined for range values, with corresponding < and > operators, but the ordering is rather arbitrary and not usually useful in the real world. Range types' B-tree and hash support is primarily meant to allow sorting and hashing internally in queries, rather than creation of actual indexes.

8.17.10. Constraints on Ranges

While `UNIQUE` is a natural constraint for scalar values, it is usually unsuitable for range types. Instead, an exclusion constraint is often more appropriate (see CREATE TABLE ... CONSTRAINT ... EXCLUDE). Exclusion constraints allow the specification of constraints such as “non-overlapping” on a range type. For example:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

That constraint will prevent any overlapping values from existing in the table at the same time:

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint
 "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01
 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01
 15:00:00")).
```

You can use the `btree_gist` extension to define exclusion constraints on plain scalar data types, which can then be combined with range exclusions for maximum flexibility. For example, after `btree_gist` is installed, the following constraint will reject overlapping ranges only if the meeting room numbers are equal:

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint
 "room_reservation_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01
 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01
 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. Domain Types

A *domain* is a user-defined data type that is based on another *underlying type*. Optionally, it can have constraints that restrict its valid values to a subset of what the underlying type would allow. Otherwise it behaves like the underlying type — for example, any operator or function that can be applied to the underlying type will work on the domain type. The underlying type can be any built-in or user-defined base type, enum type, array type, composite type, range type, or another domain.

For example, we could create a domain over integers that accepts only positive integers:

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- works
INSERT INTO mytable VALUES(-1);  -- fails
```

When an operator or function of the underlying type is applied to a domain value, the domain is automatically down-cast to the underlying type. Thus, for example, the result of `mytable.id - 1` is considered to be of type `integer` not `posint`. We could write `(mytable.id - 1)::posint` to cast the result back to `posint`, causing the domain's constraints to be rechecked. In this case, that would result in an error if the expression had been applied to an `id` value of 1. Assigning a value of the underlying type to a field or variable of the domain type is allowed without writing an explicit cast, but the domain's constraints will be checked.

For additional information see CREATE DOMAIN.

8.19. Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. Type `oid` represents an object identifier. There are also several alias types for `oid`, each named `reg`*`something`*. Table 8.26 shows an overview.

The `oid` type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables.

The `oid` type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators. (Beware of possible signed-versus-unsigned confusion if you do this.)

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type `oid` would use. The alias types allow simplified lookup of OID values for objects. For example, to examine the `pg_attribute` rows related to a table `mytable`, one could write:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

rather than:

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname =
 'mytable');
```

While that doesn't look all that bad by itself, it's still oversimplified. A far more complicated sub-select would be needed to select the right OID if there are multiple tables named `mytable` in different schemas. The `regclass` input converter handles the table lookup according to the schema path setting, and so it does the “right thing” automatically. Similarly, casting a table's OID to `regclass` is handy for symbolic display of a numeric OID.

Table 8.26. Object Identifier Types

Name	References	Description	Value Example
`oid`	any	numeric object identifier	`564182`

Name	References	Description	Value Example
regclass	pg_class	relation name	pg_type
regcollation	pg_collation	collation name	"POSIX"
regconfig	pg_ts_config	text search configuration	english
regdictionary	pg_ts_dict	text search dictionary	simple
regnamespace	pg_namespace	namespace name	pg_catalog
regoper	pg_operator	operator name	+
regoperator	pg_operator	operator with argument types	*(integer,integer) or -(NONE,integer)
regproc	pg_proc	function name	sum
regprocedure	pg_proc	function with argument types	sum(int4)
regrole	pg_authid	role name	smithee
regtype	pg_type	data type name	integer

All of the OID alias types for objects that are grouped by namespace accept schema-qualified names, and will display schema-qualified names on output if the object would not be found in the current search path without being qualified. For example, `myschema.mytable` is acceptable input for `regclass` (if there is such a table). That value might be output as `myschema.mytable`, or just `mytable`, depending on the current search path. The `regproc` and `regoper` alias types will only accept input names that are unique (not overloaded), so they are of limited use; for most uses `regprocedure` or `regoperator` are more appropriate. For `regoperator`, unary operators are identified by writing `NONE` for the unused operand.

The input functions for these types allow whitespace between tokens, and will fold upper-case letters to lower case, except within double quotes; this is done to make the syntax rules similar to the way object names are written in SQL. Conversely, the output functions will use double quotes if needed to make the output be a valid SQL identifier. For example, the OID of a function named `Foo` (with upper case `F`) taking two integer arguments could be entered as `' "Foo" ( int, integer ) '::regprocedure`. The output would look like `"Foo"(integer,integer)`. Both the function name and the argument type names could be schema-qualified, too.

Many built-in PostgreSQL functions accept the OID of a table, or another kind of database object, and for convenience are declared as taking `regclass` (or the appropriate OID alias type). This means you do not have to look up the object's OID by hand, but can just enter its name as a string literal. For example, the `nextval(regclass)` function takes a sequence relation's OID, so you could call it like this:

```
nextval('foo')              operates on sequence foo
nextval('FOO')              same as above
nextval('"Foo"')            operates on sequence Foo
nextval('myschema.foo')     operates on myschema.foo
nextval('"myschema".foo')   same as above
nextval('foo')              searches search path for foo
```

> **Note**
>
> When you write the argument of such a function as an unadorned literal string, it becomes a constant of type `regclass` (or the appropriate type). Since this is really just an OID, it will track the originally identified object despite later renaming, schema reassignment, etc. This “early binding” behavior is usually desirable for object references in column defaults and

views. But sometimes you might want “late binding” where the object reference is resolved at run time. To get late-binding behavior, force the constant to be stored as a `text` constant instead of `regclass`:

```
nextval('foo'::text)      foo is looked up at runtime
```

The `to_regclass()` function and its siblings can also be used to perform run-time lookups. See Table 9.74.

Another practical example of use of `regclass` is to look up the OID of a table listed in the `information_schema` views, which don't supply such OIDs directly. One might for example wish to call the `pg_relation_size()` function, which requires the table OID. Taking the above rules into account, the correct way to do that is

```
SELECT table_schema, table_name,
       pg_relation_size((quote_ident(table_schema) || '.' ||
                         quote_ident(table_name))::regclass)
FROM information_schema.tables
WHERE ...
```

The `quote_ident()` function will take care of double-quoting the identifiers where needed. The seemingly easier

```
SELECT pg_relation_size(table_name)
FROM information_schema.tables
WHERE ...
```

is *not recommended*, because it will fail for tables that are outside your search path or have names that require quoting.

An additional property of most of the OID alias types is the creation of dependencies. If a constant of one of these types appears in a stored expression (such as a column default expression or view), it creates a dependency on the referenced object. For example, if a column has a default expression `nextval('my_seq'::regclass)`, PostgreSQL understands that the default expression depends on the sequence `my_seq`, so the system will not let the sequence be dropped without first removing the default expression. The alternative of `nextval('my_seq'::text)` does not create a dependency. (`regrole` is an exception to this property. Constants of this type are not allowed in stored expressions.)

Another identifier type used by the system is `xid`, or transaction (abbreviated xact) identifier. This is the data type of the system columns `xmin` and `xmax`. Transaction identifiers are 32-bit quantities. In some contexts, a 64-bit variant `xid8` is used. Unlike `xid` values, `xid8` values increase strictly monotonically and cannot be reused in the lifetime of a database cluster. See Section 66.1 for more details.

A third identifier type used by the system is `cid`, or command identifier. This is the data type of the system columns `cmin` and `cmax`. Command identifiers are also 32-bit quantities.

A final identifier type used by the system is `tid`, or tuple identifier (row identifier). This is the data type of the system column `ctid`. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.

(The system columns are further explained in Section 5.6.)

8.20. pg_lsn Type

The pg_lsn data type can be used to store LSN (Log Sequence Number) data which is a pointer to a location in the WAL. This type is a representation of XLogRecPtr and an internal system type of PostgreSQL.

Internally, an LSN is a 64-bit integer, representing a byte position in the write-ahead log stream. It is printed as two hexadecimal numbers of up to 8 digits each, separated by a slash; for example, 16/B374D848. The pg_lsn type supports the standard comparison operators, like = and >. Two LSNs can be subtracted using the - operator; the result is the number of bytes separating those write-ahead log locations. Also the number of bytes can be added into and subtracted from LSN using the +(pg_lsn,numeric) and -(pg_lsn,numeric) operators, respectively. Note that the calculated LSN should be in the range of pg_lsn type, i.e., between 0/0 and FFFFFFFF/FFFFFFFF.

8.21. Pseudo-Types

The PostgreSQL type system contains a number of special-purpose entries that are collectively called *pseudo-types*. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type. Each of the available pseudo-types is useful in situations where a function's behavior does not correspond to simply taking or returning a value of a specific SQL data type. Table 8.27 lists the existing pseudo-types.

Table 8.27. Pseudo-Types

Name	Description
any	Indicates that a function accepts any input data type.
anyelement	Indicates that a function accepts any data type (see Section 36.2.5).
anyarray	Indicates that a function accepts any array data type (see Section 36.2.5).
anynonarray	Indicates that a function accepts any non-array data type (see Section 36.2.5).
anyenum	Indicates that a function accepts any enum data type (see Section 36.2.5 and Section 8.7).
anyrange	Indicates that a function accepts any range data type (see Section 36.2.5 and Section 8.17).
anymultirange	Indicates that a function accepts any multirange data type (see Section 36.2.5 and Section 8.17).
anycompatible	Indicates that a function accepts any data type, with automatic promotion of multiple arguments to a common data type (see Section 36.2.5).
anycompatiblearray	Indicates that a function accepts any array data type, with automatic promotion of multiple arguments to a common data type (see Section 36.2.5).
anycompatiblenonarray	Indicates that a function accepts any non-array data type, with automatic promotion of multiple arguments to a common data type (see Section 36.2.5).
anycompatiblerange	Indicates that a function accepts any range data type, with automatic promotion of multiple arguments to a common data type (see Section 36.2.5 and Section 8.17).
anycompatiblemultirange	Indicates that a function accepts any multirange data type, with automatic promotion of multiple arguments to a common data type (see Section 36.2.5 and Section 8.17).
cstring	Indicates that a function accepts or returns a null-terminated C string.

Name	Description
internal	Indicates that a function accepts or returns a server-internal data type.
language_handler	A procedural language call handler is declared to return language_handler.
fdw_handler	A foreign-data wrapper handler is declared to return fdw_handler.
table_am_handler	A table access method handler is declared to return table_am_handler.
index_am_handler	An index access method handler is declared to return index_am_handler.
tsm_handler	A tablesample method handler is declared to return tsm_handler.
record	Identifies a function taking or returning an unspecified row type.
trigger	A trigger function is declared to return trigger.
event_trigger	An event trigger function is declared to return event_trigger.
pg_ddl_command	Identifies a representation of DDL commands that is available to event triggers.
void	Indicates that a function returns no value.
unknown	Identifies a not-yet-resolved type, e.g., of an undecorated string literal.

Functions coded in C (whether built-in or dynamically loaded) can be declared to accept or return any of these pseudo-types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. At present most procedural languages forbid use of a pseudo-type as an argument type, and allow only void and record as a result type (plus trigger or event_trigger when the function is used as a trigger or event trigger). Some also support polymorphic functions using the polymorphic pseudo-types, which are shown above and discussed in detail in Section 36.2.5.

The internal pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct invocation in an SQL query. If a function has at least one internal-type argument then it cannot be called from SQL. To preserve the type safety of this restriction it is important to follow this coding rule: do not create any function that is declared to return internal unless it has at least one internal argument.

Chapter 9. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. This chapter describes most of them, although additional special-purpose functions appear in relevant sections of the manual. Users can also define their own functions and operators, as described in Part V. The psql commands `\df` and `\do` can be used to list all available functions and operators, respectively.

The notation used throughout this chapter to describe the argument and result data types of a function or operator is like this:

```
repeat ( text, integer ) → text
```

which says that the function `repeat` takes one text and one integer argument and returns a result of type text. The right arrow is also used to indicate the result of an example, thus:

```
repeat('Pg', 4) → PgPgPgPg
```

If you are concerned about portability then note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of this extended functionality is present in other SQL database management systems, and in many cases this functionality is compatible and consistent between the various implementations.

9.1. Logical Operators

The usual logical operators are available:

```
boolean AND boolean → boolean
boolean OR boolean → boolean
NOT boolean → boolean
```

SQL uses a three-valued logic system with true, false, and `null`, which represents "unknown". Observe the following truth tables:

a	***b***	***a* AND *b***	***a* OR *b***
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT *a***
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The operators `AND` and `OR` are commutative, that is, you can switch the left and right operands without affecting the result. (However, it is not guaranteed that the left operand is evaluated before the right operand. See Section 4.2.14 for more information about the order of evaluation of subexpressions.)

9.2. Comparison Functions and Operators

The usual comparison operators are available, as shown in Table 9.1.

Table 9.1. Comparison Operators

Operator	Description
datatype < *datatype* → boolean	Less than
datatype > *datatype* → boolean	Greater than
datatype <= *datatype* → boolean	Less than or equal to
datatype >= *datatype* → boolean	Greater than or equal to
datatype = *datatype* → boolean	Equal
datatype <> *datatype* → boolean	Not equal
datatype != *datatype* → boolean	Not equal

> **Note**
>
> <> is the standard SQL notation for "not equal". != is an alias, which is converted to <> at a very early stage of parsing. Hence, it is not possible to implement != and <> operators that do different things.

These comparison operators are available for all built-in data types that have a natural ordering, including numeric, string, and date/time types. In addition, arrays, composite types, and ranges can be compared if their component data types are comparable.

It is usually possible to compare values of related data types as well; for example integer > bigint will work. Some cases of this sort are implemented directly by "cross-type" comparison operators, but if no such operator is available, the parser will coerce the less-general type to the more-general type and apply the latter's comparison operator.

As shown above, all comparison operators are binary operators that return values of type boolean. Thus, expressions like 1 < 2 < 3 are not valid (because there is no < operator to compare a Boolean value with 3). Use the BETWEEN predicates shown below to perform range tests.

There are also some comparison predicates, as shown in Table 9.2. These behave much like operators, but have special syntax mandated by the SQL standard.

Table 9.2. Comparison Predicates

Predicate Description Example(s)
datatype BETWEEN *datatype* AND *datatype* → boolean Between (inclusive of the range endpoints). 2 BETWEEN 1 AND 3 → t 2 BETWEEN 3 AND 1 → f
datatype NOT BETWEEN *datatype* AND *datatype* → boolean Not between (the negation of BETWEEN). 2 NOT BETWEEN 1 AND 3 → f

Predicate **Description** **Example(s)**
datatype `BETWEEN SYMMETRIC` *datatype* `AND` *datatype* → `boolean` Between, after sorting the two endpoint values. `2 BETWEEN SYMMETRIC 3 AND 1` → `t`
datatype `NOT BETWEEN SYMMETRIC` *datatype* `AND` *datatype* → `boolean` Not between, after sorting the two endpoint values. `2 NOT BETWEEN SYMMETRIC 3 AND 1` → `f`
datatype `IS DISTINCT FROM` *datatype* → `boolean` Not equal, treating null as a comparable value. `1 IS DISTINCT FROM NULL` → `t` (rather than `NULL`) `NULL IS DISTINCT FROM NULL` → `f` (rather than `NULL`)
datatype `IS NOT DISTINCT FROM` *datatype* → `boolean` Equal, treating null as a comparable value. `1 IS NOT DISTINCT FROM NULL` → `f` (rather than `NULL`) `NULL IS NOT DISTINCT FROM NULL` → `t` (rather than `NULL`)
datatype `IS NULL` → `boolean` Test whether value is null. `1.5 IS NULL` → `f`
datatype `IS NOT NULL` → `boolean` Test whether value is not null. `'null' IS NOT NULL` → `t`
datatype `ISNULL` → `boolean` Test whether value is null (nonstandard syntax).
datatype `NOTNULL` → `boolean` Test whether value is not null (nonstandard syntax).

`boolean IS TRUE` → `boolean`
Test whether boolean expression yields true.
`true IS TRUE` → `t`
`NULL::boolean IS TRUE` → `f` (rather than `NULL`)

`boolean IS NOT TRUE` → `boolean`
Test whether boolean expression yields false or unknown.
`true IS NOT TRUE` → `f`
`NULL::boolean IS NOT TRUE` → `t` (rather than `NULL`)

`boolean IS FALSE` → `boolean`
Test whether boolean expression yields false.
`true IS FALSE` → `f`
`NULL::boolean IS FALSE` → `f` (rather than `NULL`)

`boolean IS NOT FALSE` → `boolean`
Test whether boolean expression yields true or unknown.
`true IS NOT FALSE` → `t`
`NULL::boolean IS NOT FALSE` → `t` (rather than `NULL`)

Predicate **Description** **Example(s)**

`boolean IS UNKNOWN → boolean`
Test whether boolean expression yields unknown.
`true IS UNKNOWN → f`
`NULL::boolean IS UNKNOWN → t` (rather than `NULL`)

`boolean IS NOT UNKNOWN → boolean`
Test whether boolean expression yields true or false.
`true IS NOT UNKNOWN → t`
`NULL::boolean IS NOT UNKNOWN → f` (rather than `NULL`)

The `BETWEEN` predicate simplifies range tests:

```
a BETWEEN x AND y
```

is equivalent to

```
a >= x AND a <= y
```

Notice that `BETWEEN` treats the endpoint values as included in the range. `BETWEEN SYMMETRIC` is like `BETWEEN` except there is no requirement that the argument to the left of `AND` be less than or equal to the argument on the right. If it is not, those two arguments are automatically swapped, so that a nonempty range is always implied.

The various variants of `BETWEEN` are implemented in terms of the ordinary comparison operators, and therefore will work for any data type(s) that can be compared.

> **Note**
>
> The use of `AND` in the `BETWEEN` syntax creates an ambiguity with the use of `AND` as a logical operator. To resolve this, only a limited set of expression types are allowed as the second argument of a `BETWEEN` clause. If you need to write a more complex sub-expression in `BETWEEN`, write parentheses around the sub-expression.

Ordinary comparison operators yield null (signifying "unknown"), not true or false, when either input is null. For example, `7 = NULL` yields null, as does `7 <> NULL`. When this behavior is not suitable, use the `IS [ NOT ] DISTINCT FROM` predicates:

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

For non-null inputs, `IS DISTINCT FROM` is the same as the `<>` operator. However, if both inputs are null it returns false, and if only one input is null it returns true. Similarly, `IS NOT DISTINCT FROM` is identical to `=` for non-null inputs, but it returns true when both inputs are null, and false when only one input is null. Thus, these predicates effectively act as though null were a normal data value, rather than "unknown".

To check whether a value is or is not null, use the predicates:

```
expression IS NULL
```

```
expression IS NOT NULL
```

or the equivalent, but nonstandard, predicates:

```
expression ISNULL
expression NOTNULL
```

Do *not* write `expression = NULL` because `NULL` is not "equal to" `NULL`. (The null value represents an unknown value, and it is not known whether two unknown values are equal.)

> **Tip**
>
> Some applications might expect that `expression = NULL` returns true if `expression` evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard. However, if that cannot be done the transform_null_equals configuration variable is available. If it is enabled, PostgreSQL will convert `x = NULL` clauses to `x IS NULL`.

If the `expression` is row-valued, then `IS NULL` is true when the row expression itself is null or when all the row's fields are null, while `IS NOT NULL` is true when the row expression itself is non-null and all the row's fields are non-null. Because of this behavior, `IS NULL` and `IS NOT NULL` do not always return inverse results for row-valued expressions; in particular, a row-valued expression that contains both null and non-null fields will return false for both tests. For example:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows

SELECT ROW(table.*) IS NOT NULL FROM table;  -- detect all-non-null
 rows

SELECT NOT(ROW(table.*) IS NOT NULL) FROM TABLE; -- detect at least
 one null in rows
```

In some cases, it may be preferable to write `row IS DISTINCT FROM NULL` or `row IS NOT DISTINCT FROM NULL`, which will simply check whether the overall row value is null without any additional tests on the row fields.

Boolean values can also be tested using the predicates

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

These will always return true or false, never a null value, even when the operand is null. A null input is treated as the logical value "unknown". Notice that `IS UNKNOWN` and `IS NOT UNKNOWN` are effectively the same as `IS NULL` and `IS NOT NULL`, respectively, except that the input expression must be of Boolean type.

Some comparison-related functions are also available, as shown in Table 9.3.

Table 9.3. Comparison Functions

Function Description Example(s)

`num_nonnulls ( VARIADIC "any" ) → integer`
Returns the number of non-null arguments.
`num_nonnulls(1, NULL, 2) → 2`

`num_nulls ( VARIADIC "any" ) → integer`
Returns the number of null arguments.
`num_nulls(1, NULL, 2) → 1`

9.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without standard mathematical conventions (e.g., date/time types) we describe the actual behavior in subsequent sections.

Table 9.4 shows the mathematical operators that are available for the standard numeric types. Unless otherwise noted, operators shown as accepting *`numeric_type`* are available for all the types `smallint`, `integer`, `bigint`, `numeric`, `real`, and `double precision`. Operators shown as accepting *`integral_type`* are available for the types `smallint`, `integer`, and `bigint`. Except where noted, each form of an operator returns the same data type as its argument(s). Calls involving multiple argument data types, such as `integer + numeric`, are resolved by using the type appearing later in these lists.

Table 9.4. Mathematical Operators

Operator Description Example(s)
`numeric_type` `+` *`numeric_type`* `→` *`numeric_type`* Addition `2 + 3 → 5`
`+` *`numeric_type`* `→` *`numeric_type`* Unary plus (no operation) `+ 3.5 → 3.5`
`numeric_type` `-` *`numeric_type`* `→` *`numeric_type`* Subtraction `2 - 3 → -1`
`-` *`numeric_type`* `→` *`numeric_type`* Negation `- (-4) → 4`
`numeric_type` `*` *`numeric_type`* `→` *`numeric_type`* Multiplication `2 * 3 → 6`

`numeric_type` `/` *`numeric_type`* `→` *`numeric_type`*
Division (for integral types, division truncates the result towards zero)
`5.0 / 2 → 2.5000000000000000`
`5 / 2 → 2`

Operator **Description** **Example(s)**
`(-5) / 2 → -2`
numeric_type `%` *numeric_type* → *numeric_type* Modulo (remainder); available for `smallint`, `integer`, `bigint`, and `numeric` `5 % 4 → 1`

`numeric ^ numeric → numeric`
`double precision ^ double precision → double precision`
Exponentiation
`2 ^ 3 → 8`
Unlike typical mathematical practice, multiple uses of `^` will associate left to right by default:
`2 ^ 3 ^ 3 → 512`
`2 ^ (3 ^ 3) → 134217728`

`|/ double precision → double precision`
Square root
`|/ 25.0 → 5`

`||/ double precision → double precision`
Cube root
`||/ 64.0 → 4`

`@` *numeric_type* → *numeric_type* Absolute value `@ -5.0 → 5.0`
integral_type `&` *integral_type* → *integral_type* Bitwise AND `91 & 15 → 11`
integral_type `|` *integral_type* → *integral_type* Bitwise OR `32 | 3 → 35`
integral_type `#` *integral_type* → *integral_type* Bitwise exclusive OR `17 # 5 → 20`
`~` *integral_type* → *integral_type* Bitwise NOT `~1 → -2`
integral_type `<<` `integer` → *integral_type* Bitwise shift left `1 << 4 → 16`
integral_type `>>` `integer` → *integral_type* Bitwise shift right `8 >> 2 → 2`

Table 9.5 shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the

same data type as its argument(s); cross-type cases are resolved in the same way as explained above for operators. The functions working with `double precision` data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases can therefore vary depending on the host system.

Table 9.5. Mathematical Functions

Function **Description** **Example(s)**

`abs ( numeric_type ) → numeric_type`
Absolute value
`abs(-17.4) → 17.4`

`cbrt ( double precision ) → double precision`
Cube root
`cbrt(64.0) → 4`

`ceil ( numeric ) → numeric`
`ceil ( double precision ) → double precision`
Nearest integer greater than or equal to argument
`ceil(42.2) → 43`
`ceil(-42.8) → -42`

`ceiling ( numeric ) → numeric`
`ceiling ( double precision ) → double precision`
Nearest integer greater than or equal to argument (same as `ceil`)
`ceiling(95.3) → 96`

`degrees ( double precision ) → double precision`
Converts radians to degrees
`degrees(0.5) → 28.64788975654116`

`div ( y numeric, x numeric ) → numeric`
Integer quotient of *y*/*x* (truncates towards zero)
`div(9, 4) → 2`

`erf ( double precision ) → double precision`
Error function
`erf(1.0) → 0.8427007929497149`

`erfc ( double precision ) → double precision`
Complementary error function (`1 - erf(x)`, without loss of precision for large inputs)
`erfc(1.0) → 0.15729920705028513`

`exp ( numeric ) → numeric`
`exp ( double precision ) → double precision`
Exponential (`e` raised to the given power)
`exp(1.0) → 2.7182818284590452`

`factorial ( bigint ) → numeric`
Factorial
`factorial(5) → 120`

`floor ( numeric ) → numeric`

Function **Description** **Example(s)**

`floor ( double precision ) → double precision`
Nearest integer less than or equal to argument
`floor(42.8) → 42`
`floor(-42.8) → -43`

`gcd ( numeric_type, numeric_type ) → numeric_type`
Greatest common divisor (the largest positive number that divides both inputs with no remainder); returns `0` if both inputs are zero; available for `integer`, `bigint`, and `numeric`
`gcd(1071, 462) → 21`

`lcm ( numeric_type, numeric_type ) → numeric_type`
Least common multiple (the smallest strictly positive number that is an integral multiple of both inputs); returns `0` if either input is zero; available for `integer`, `bigint`, and `numeric`
`lcm(1071, 462) → 23562`

`ln ( numeric ) → numeric`
`ln ( double precision ) → double precision`
Natural logarithm
`ln(2.0) → 0.6931471805599453`

`log ( numeric ) → numeric`
`log ( double precision ) → double precision`
Base 10 logarithm
`log(100) → 2`

`log10 ( numeric ) → numeric`
`log10 ( double precision ) → double precision`
Base 10 logarithm (same as `log`)
`log10(1000) → 3`

`log ( b numeric, x numeric ) → numeric`
Logarithm of *x* to base *b*
`log(2.0, 64.0) → 6.0000000000000000`

`min_scale ( numeric ) → integer`
Minimum scale (number of fractional decimal digits) needed to represent the supplied value precisely
`min_scale(8.4100) → 2`

`mod ( y numeric_type, x numeric_type ) → numeric_type`
Remainder of *y*/*x*; available for `smallint`, `integer`, `bigint`, and `numeric`
`mod(9, 4) → 1`

`pi ( ) → double precision`
Approximate value of π
`pi() → 3.141592653589793`

`power ( a numeric, b numeric ) → numeric` `power ( a double precision, b double precision ) → double precision`

Function **Description** **Example(s)**
a raised to the power of *b* `power(9, 3)` → `729`

`radians ( double precision ) → double precision`
Converts degrees to radians
`radians(45.0)` → `0.7853981633974483`

`round ( numeric ) → numeric`
`round ( double precision ) → double precision`
Rounds to nearest integer. For `numeric`, ties are broken by rounding away from zero. For `double precision`, the tie-breaking behavior is platform dependent, but “round to nearest even” is the most common rule.
`round(42.4)` → `42`

`round ( v numeric, s integer ) → numeric`
Rounds *v* to *s* decimal places. Ties are broken by rounding away from zero.
`round(42.4382, 2)` → `42.44`
`round(1234.56, -1)` → `1230`

`scale ( numeric ) → integer`
Scale of the argument (the number of decimal digits in the fractional part)
`scale(8.4100)` → `4`

`sign ( numeric ) → numeric`
`sign ( double precision ) → double precision`
Sign of the argument (-1, 0, or +1)
`sign(-8.4)` → `-1`

`sqrt ( numeric ) → numeric`
`sqrt ( double precision ) → double precision`
Square root
`sqrt(2)` → `1.4142135623730951`

`trim_scale ( numeric ) → numeric`
Reduces the value's scale (number of fractional decimal digits) by removing trailing zeroes
`trim_scale(8.4100)` → `8.41`

`trunc ( numeric ) → numeric`
`trunc ( double precision ) → double precision`
Truncates to integer (towards zero)
`trunc(42.8)` → `42`
`trunc(-42.8)` → `-42`

`trunc ( v numeric, s integer ) → numeric`
Truncates *v* to *s* decimal places
`trunc(42.4382, 2)` → `42.43`

`width_bucket ( operand numeric, low numeric, high numeric, count integer ) → integer`

Function Description Example(s)

`width_bucket` (*operand* `double precision`, *low* `double precision`, *high* `double precision`, *count* `integer`) → `integer`
Returns the number of the bucket in which *operand* falls in a histogram having *count* equal-width buckets spanning the range *low* to *high*. Returns `0` or *count*`+1` for an input outside that range.
`width_bucket(5.35, 0.024, 10.06, 5)` → `3`

`width_bucket` (*operand* `anycompatible`, *thresholds* `anycompatiblearray`) → `integer`
Returns the number of the bucket in which *operand* falls given an array listing the lower bounds of the buckets. Returns `0` for an input less than the first lower bound. *operand* and the array elements can be of any type having standard comparison operators. The *thresholds* array *must be sorted*, smallest first, or unexpected results will be obtained.
`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])` → `2`

Table 9.6 shows functions for generating random numbers.

Table 9.6. Random Functions

Function Description Example(s)

`random` ( ) → `double precision`
Returns a random value in the range 0.0 <= x < 1.0
`random()` → `0.897124072839091`

`random` (*min* `integer`, *max* `integer`) → `integer`
`random` (*min* `bigint`, *max* `bigint`) → `bigint`
`random` (*min* `numeric`, *max* `numeric`) → `numeric`
Returns a random value in the range *min* <= x <= *max*. For type `numeric`, the result will have the same number of fractional decimal digits as *min* or *max*, whichever has more.
`random(1, 10)` → `7`
`random(-0.499, 0.499)` → `0.347`

`random_normal` ([*mean* `double precision` [, *stddev* `double precision`]]) → `double precision`
Returns a random value from the normal distribution with the given parameters; *mean* defaults to 0.0 and *stddev* defaults to 1.0
`random_normal(0.0, 1.0)` → `0.051285419`

`setseed` (`double precision`) → `void`
Sets the seed for subsequent `random()` and `random_normal()` calls; argument must be between -1.0 and 1.0, inclusive
`setseed(0.12345)`

The `random()` and `random_normal()` functions listed in Table 9.6 use a deterministic pseudo-random number generator. It is fast but not suitable for cryptographic applications; see the pgcrypto module for a more secure alternative. If `setseed()` is called, the series of results of subsequent calls to these functions in the current session can be repeated by re-issuing `setseed()` with the

same argument. Without any prior `setseed()` call in the same session, the first call to any of these functions obtains a seed from a platform-dependent source of random bits.

Table 9.7 shows the available trigonometric functions. Each of these functions comes in two variants, one that measures angles in radians and one that measures angles in degrees.

Table 9.7. Trigonometric Functions

Function Description Example(s)

`acos(double precision)` → `double precision`
Inverse cosine, result in radians
`acos(1)` → `0`

`acosd(double precision)` → `double precision`
Inverse cosine, result in degrees
`acosd(0.5)` → `60`

`asin(double precision)` → `double precision`
Inverse sine, result in radians
`asin(1)` → `1.5707963267948966`

`asind(double precision)` → `double precision`
Inverse sine, result in degrees
`asind(0.5)` → `30`

`atan(double precision)` → `double precision`
Inverse tangent, result in radians
`atan(1)` → `0.7853981633974483`

`atand(double precision)` → `double precision`
Inverse tangent, result in degrees
`atand(1)` → `45`

`atan2(`*y* `double precision`, *x* `double precision`) → `double precision`
Inverse tangent of *y*/*x*, result in radians
`atan2(1, 0)` → `1.5707963267948966`

`atan2d(`*y* `double precision`, *x* `double precision`) → `double precision`
Inverse tangent of *y*/*x*, result in degrees
`atan2d(1, 0)` → `90`

`cos(double precision)` → `double precision`
Cosine, argument in radians
`cos(0)` → `1`

`cosd(double precision)` → `double precision`
Cosine, argument in degrees
`cosd(60)` → `0.5`

`cot(double precision)` → `double precision`
Cotangent, argument in radians
`cot(0.5)` → `1.830487721712452`

`cotd(double precision)` → `double precision` Cotangent, argument in degrees

Function Description Example(s)
`cotd(45)` → `1`

`sin(double precision)` → `double precision`
Sine, argument in radians
`sin(1)` → `0.8414709848078965`

`sind(double precision)` → `double precision`
Sine, argument in degrees
`sind(30)` → `0.5`

`tan(double precision)` → `double precision`
Tangent, argument in radians
`tan(1)` → `1.5574077246549023`

`tand(double precision)` → `double precision`
Tangent, argument in degrees
`tand(45)` → `1`

> **Note**
>
> Another way to work with angles measured in degrees is to use the unit transformation functions `radians()` and `degrees()` shown earlier. However, using the degree-based trigonometric functions is preferred, as that way avoids round-off error for special cases such as `sind(30)`.

Table 9.8 shows the available hyperbolic functions.

Table 9.8. Hyperbolic Functions

Function Description Example(s)

`sinh(double precision)` → `double precision`
Hyperbolic sine
`sinh(1)` → `1.1752011936438014`

`cosh(double precision)` → `double precision`
Hyperbolic cosine
`cosh(0)` → `1`

`tanh(double precision)` → `double precision`
Hyperbolic tangent
`tanh(1)` → `0.7615941559557649`

`asinh(double precision)` → `double precision`
Inverse hyperbolic sine
`asinh(1)` → `0.881373587019543`

`acosh(double precision)` → `double precision`
Inverse hyperbolic cosine
`acosh(1)` → `0`

Function Description Example(s)

`atanh ( double precision ) → double precision`
Inverse hyperbolic tangent
`atanh(0.5) → 0.5493061443340548`

9.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types `character`, `character varying`, and `text`. Except where noted, these functions and operators are declared to accept and return type `text`. They will interchangeably accept `character varying` arguments. Values of type `character` will be converted to `text` before the function or operator is applied, resulting in stripping any trailing spaces in the `character` value.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9.9. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9.10).

> **Note**
>
> The string concatenation operator (`||`) will accept non-string input, so long as at least one input is of string type, as shown in Table 9.9. For other cases, inserting an explicit coercion to `text` can be used to have non-string input accepted.

Table 9.9. SQL String Functions and Operators

Function/Operator Description Example(s)

`text || text → text`
Concatenates the two strings.
`'Post' || 'greSQL' → PostgreSQL`

`text || anynonarray → text`
`anynonarray || text → text`
Converts the non-string input to text, then concatenates the two strings. (The non-string input cannot be of an array type, because that would create ambiguity with the array `||` operators. If you want to concatenate an array's text equivalent, cast it to `text` explicitly.)
`'Value: ' || 42 → Value: 42`

`btrim ( string text [, characters text ] ) → text`
Removes the longest string containing only characters in *`characters`* (a space by default) from the start and end of *`string`*.
`btrim('xyxtrimyyx', 'xyz') → trim`

`text IS [NOT] [form] NORMALIZED → boolean` Checks whether the string is in the specified Unicode normalization form. The optional *`form`* key word specifies the form: `NFC` (the default), `NFD`, `NFKC`, or `NFKD`. This expression can only be used when the server encoding is `UTF8`. Note that checking for normalization using this expression is often faster than normalizing possibly already normalized strings.

Function/Operator **Description** **Example(s)**
`U&'\0061\0308bc' IS NFD NORMALIZED` → `t`

`bit_length` (`text`) → `integer`
Returns number of bits in the string (8 times the `octet_length`).
`bit_length('jose')` → `32`

`char_length` (`text`) → `integer`
`character_length` (`text`) → `integer`
Returns number of characters in the string.
`char_length('josé')` → `4`

`lower` (`text`) → `text`
Converts the string to all lower case, according to the rules of the database's locale.
`lower('TOM')` → `tom`

`lpad` (*`string`* `text`, *`length`* `integer` [, *`fill`* `text`]) → `text`
Extends the *`string`* to length *`length`* by prepending the characters *`fill`* (a space by default). If the *`string`* is already longer than *`length`* then it is truncated (on the right).
`lpad('hi', 5, 'xy')` → `xyxhi`

`ltrim` (*`string`* `text` [, *`characters`* `text`]) → `text`
Removes the longest string containing only characters in *`characters`* (a space by default) from the start of *`string`*.
`ltrim('zzzytest', 'xyz')` → `test`

`normalize` (`text` [, *`form`*]) → `text`
Converts the string to the specified Unicode normalization form. The optional *`form`* key word specifies the form: `NFC` (the default), `NFD`, `NFKC`, or `NFKD`. This function can only be used when the server encoding is `UTF8`.
`normalize(U&'\0061\0308bc', NFC)` → `U&'\00E4bc'`

`octet_length` (`text`) → `integer` Returns number of bytes in the string. `octet_length('josé')` → `5` (if server encoding is UTF8)

`octet_length` (`character`) → `integer`
Returns number of bytes in the string. Since this version of the function accepts type `character` directly, it will not strip trailing spaces.
`octet_length('abc '::character(4))` → `4`

`overlay` (*`string`* `text` `PLACING` *`newsubstring`* `text` `FROM` *`start`* `integer` [`FOR` *`count`* `integer`]) → `text`
Replaces the substring of *`string`* that starts at the *`start`*'th character and extends for *`count`* characters with *`newsubstring`*. If *`count`* is omitted, it defaults to the length of *`newsubstring`*.
`overlay('Txxxxas' placing 'hom' from 2 for 4)` → `Thomas`

`position` (*`substring`* `text` `IN` *`string`* `text`) → `integer`
Returns first starting index of the specified *`substring`* within *`string`*, or zero if it's not present.
`position('om' in 'Thomas')` → `3`

`rpad` (*`string`* `text`, *`length`* `integer` [, *`fill`* `text`]) → `text`

Function/Operator **Description** **Example(s)**
Extends the *string* to length *length* by appending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated. `rpad('hi', 5, 'xy')` → `hixyx`

`rtrim(` *string* `text` [, *characters* `text`]`)` → `text`
Removes the longest string containing only characters in *characters* (a space by default) from the end of *string*.
`rtrim('testxxzx', 'xyz')` → `test`

`substring(` *string* `text` [`FROM` *start* `integer`] [`FOR` *count* `integer`]`)` → `text`
Extracts the substring of *string* starting at the *start*'th character if that is specified, and stopping after *count* characters if that is specified. Provide at least one of *start* and *count*.
`substring('Thomas' from 2 for 3)` → `hom`
`substring('Thomas' from 3)` → `omas`
`substring('Thomas' for 2)` → `Th`

`substring(` *string* `text` `FROM` *pattern* `text` `)` → `text`
Extracts the first substring matching POSIX regular expression; see Section 9.7.3.
`substring('Thomas' from '...$')` → `mas`

`substring(` *string* `text` `SIMILAR` *pattern* `text` `ESCAPE` *escape* `text` `)` → `text`
`substring(` *string* `text` `FROM` *pattern* `text` `FOR` *escape* `text` `)` → `text`
Extracts the first substring matching SQL regular expression; see Section 9.7.2. The first form has been specified since SQL:2003; the second form was only in SQL:1999 and should be considered obsolete.
`substring('Thomas' similar '%#"o_a#"_' escape '#')` → `oma`

`trim(` [`LEADING` | `TRAILING` | `BOTH`] [*characters* `text`] `FROM` *string* `text` `)` → `text`
Removes the longest string containing only characters in *characters* (a space by default) from the start, end, or both ends (`BOTH` is the default) of *string*.
`trim(both 'xyz' from 'yxTomxx')` → `Tom`

`trim(` [`LEADING` | `TRAILING` | `BOTH`] [`FROM`] *string* `text` [, *characters* `text`]`)` → `text`
This is a non-standard syntax for `trim()`.
`trim(both from 'yxTomxx', 'xyz')` → `Tom`

`unicode_assigned(` `text` `)` → `boolean` Returns `true` if all characters in the string are assigned Unicode codepoints; `false` otherwise. This function can only be used when the server encoding is `UTF8`.

`upper(` `text` `)` → `text`
Converts the string to all upper case, according to the rules of the database's locale.
`upper('tom')` → `TOM`

Additional string manipulation functions and operators are available and are listed in Table 9.10. (Some of these are used internally to implement the SQL-standard string functions listed in Table 9.9.) There are also pattern-matching operators, which are described in Section 9.7, and operators for full-text search, which are described in Chapter 12.

Table 9.10. Other String Functions and Operators

Function/Operator Description Example(s)

`text ^@ text → boolean`
Returns true if the first string starts with the second string (equivalent to the `starts_with()` function).
`'alphabet' ^@ 'alph' → t`

`ascii ( text ) → integer`
Returns the numeric code of the first character of the argument. In UTF8 encoding, returns the Unicode code point of the character. In other multibyte encodings, the argument must be an ASCII character.
`ascii('x') → 120`

`chr ( integer ) → text`
Returns the character with the given code. In UTF8 encoding the argument is treated as a Unicode code point. In other multibyte encodings the argument must designate an ASCII character. `chr(0)` is disallowed because text data types cannot store that character.
`chr(65) → A`

`concat ( val1 "any" [, val2 "any" [, ...] ] ) → text`
Concatenates the text representations of all the arguments. NULL arguments are ignored.
`concat('abcde', 2, NULL, 22) → abcde222`

`concat_ws ( sep text, val1 "any" [, val2 "any" [, ...] ] ) → text`
Concatenates all but the first argument, with separators. The first argument is used as the separator string, and should not be NULL. Other NULL arguments are ignored.
`concat_ws(',', 'abcde', 2, NULL, 22) → abcde,2,22`

`format ( formatstr text [, formatarg "any" [, ...] ] ) → text`
Formats arguments according to a format string; see Section 9.4.1. This function is similar to the C function `sprintf`.
`format('Hello %s, %1$s', 'World') → Hello World, World`

`initcap ( text ) → text`
Converts the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.
`initcap('hi THOMAS') → Hi Thomas`

`left ( string text, n integer ) → text`
Returns first *n* characters in the string, or when *n* is negative, returns all but last |*n*| characters.
`left('abcde', 2) → ab`

`length ( text ) → integer`
Returns the number of characters in the string.
`length('jose') → 4`

`md5 ( text ) → text`
Computes the MD5 hash of the argument, with the result written in hexadecimal.
`md5('abc') → 900150983cd24fb0d6963f7d28e17f72`

`parse_ident ( qualified_identifier text [, strict_mode boolean DEFAULT true ] ) → text[]`

Function/Operator **Description** **Example(s)**
Splits *qualified_identifier* into an array of identifiers, removing any quoting of individual identifiers. By default, extra characters after the last identifier are considered an error; but if the second parameter is false, then such extra characters are ignored. (This behavior is useful for parsing names for objects like functions.) Note that this function does not truncate over-length identifiers. If you want truncation you can cast the result to name[]. parse_ident('"SomeSchema".someTable') → {SomeSchema,sometable}
pg_client_encoding ( ) → name Returns current client encoding name. pg_client_encoding() → UTF8
quote_ident (text) → text Returns the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled. See also Example 41.1. quote_ident('Foo bar') → "Foo bar"
quote_literal (text) → text Returns the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that quote_literal returns null on null input; if the argument might be null, quote_nullable is often more suitable. See also Example 41.1. quote_literal(E'O\'Reilly') → 'O''Reilly'
quote_literal (anyelement) → text Converts the given value to text and then quotes it as a literal. Embedded single-quotes and backslashes are properly doubled. quote_literal(42.5) → '42.5'
quote_nullable (text) → text Returns the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. See also Example 41.1. quote_nullable(NULL) → NULL
quote_nullable (anyelement) → text Converts the given value to text and then quotes it as a literal; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. quote_nullable(42.5) → '42.5'
regexp_count (*string* text, *pattern* text [, *start* integer [, *flags* text]]) → integer Returns the number of times the POSIX regular expression *pattern* matches in the *string*; see Section 9.7.3. regexp_count('123456789012', '\d\d\d', 2) → 3
regexp_instr (*string* text, *pattern* text [, *start* integer [, *N* integer [, *endoption* integer [, *flags* text [, *subexpr* integer]]]]]) → integer Returns the position within *string* where the *N*'th match of the POSIX regular expression *pattern* occurs, or zero if there is no such match; see Section 9.7.3. regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i') → 3

Function/Operator **Description** **Example(s)**
`regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i', 2)` → `5`

`regexp_like` (*`string`* `text`, *`pattern`* `text` [, *`flags`* `text`]) → `boolean`
Checks whether a match of the POSIX regular expression *`pattern`* occurs within *`string`*; see Section 9.7.3.
`regexp_like('Hello World', 'world$', 'i')` → `t`

`regexp_match` (*`string`* `text`, *`pattern`* `text` [, *`flags`* `text`]) → `text[]`
Returns substrings within the first match of the POSIX regular expression *`pattern`* to the *`string`*; see Section 9.7.3.
`regexp_match('foobarbequebaz', '(bar)(beque)')` → `{bar,beque}`

`regexp_matches` (*`string`* `text`, *`pattern`* `text` [, *`flags`* `text`]) → `setof text[]`
Returns substrings within the first match of the POSIX regular expression *`pattern`* to the *`string`*, or substrings within all such matches if the `g` flag is used; see Section 9.7.3.
`regexp_matches('foobarbequebaz', 'ba.', 'g')` →
`{bar}`
`{baz}`

`regexp_replace` (*`string`* `text`, *`pattern`* `text`, *`replacement`* `text` [, *`start`* `integer`] [, *`flags`* `text`]) → `text`
Replaces the substring that is the first match to the POSIX regular expression *`pattern`*, or all such matches if the `g` flag is used; see Section 9.7.3.
`regexp_replace('Thomas', '.[mN]a.', 'M')` → `ThM`

`regexp_replace` (*`string`* `text`, *`pattern`* `text`, *`replacement`* `text`, *`start`* `integer`, *`N`* `integer` [, *`flags`* `text`]) → `text`
Replaces the substring that is the *`N`*'th match to the POSIX regular expression *`pattern`*, or all such matches if *`N`* is zero; see Section 9.7.3.
`regexp_replace('Thomas', '.', 'X', 3, 2)` → `ThoXas`

`regexp_split_to_array` (*`string`* `text`, *`pattern`* `text` [, *`flags`* `text`]) → `text[]`
Splits *`string`* using a POSIX regular expression as the delimiter, producing an array of results; see Section 9.7.3.
`regexp_split_to_array('hello world', '\s+')` → `{hello,world}`

`regexp_split_to_table` (*`string`* `text`, *`pattern`* `text` [, *`flags`* `text`]) → `setof text`
Splits *`string`* using a POSIX regular expression as the delimiter, producing a set of results; see Section 9.7.3.
`regexp_split_to_table('hello world', '\s+')` →
`hello`
`world`

`regexp_substr` (*`string`* `text`, *`pattern`* `text` [, *`start`* `integer` [, *`N`* `integer` [, *`flags`* `text` [, *`subexpr`* `integer`]]]]) → `text`

Function/Operator **Description** **Example(s)**

Returns the substring within *string* that matches the *N*'th occurrence of the POSIX regular expression *pattern*, or NULL if there is no such match; see Section 9.7.3.
`regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i')` → `CDEF`
`regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i', 2)` → `EF`

`repeat` (*string* `text`, *number* `integer`) → `text`
Repeats *string* the specified *number* of times.
`repeat('Pg', 4)` → `PgPgPgPg`

`replace` (*string* `text`, *from* `text`, *to* `text`) → `text`
Replaces all occurrences in *string* of substring *from* with substring *to*.
`replace('abcdefabcdef', 'cd', 'XX')` → `abXXefabXXef`

`reverse` (`text`) → `text`
Reverses the order of the characters in the string.
`reverse('abcde')` → `edcba`

`right` (*string* `text`, *n* `integer`) → `text`
Returns last *n* characters in the string, or when *n* is negative, returns all but first |*n*| characters.
`right('abcde', 2)` → `de`

`split_part` (*string* `text`, *delimiter* `text`, *n* `integer`) → `text`
Splits *string* at occurrences of *delimiter* and returns the *n*'th field (counting from one), or when *n* is negative, returns the |*n*|'th-from-last field.
`split_part('abc~@~def~@~ghi', '~@~', 2)` → `def`
`split_part('abc,def,ghi,jkl', ',', -2)` → `ghi`

`starts_with` (*string* `text`, *prefix* `text`) → `boolean`
Returns true if *string* starts with *prefix*.
`starts_with('alphabet', 'alph')` → `t`

`string_to_array` (*string* `text`, *delimiter* `text` [, *null_string* `text`]) → `text[]`
Splits the *string* at occurrences of *delimiter* and forms the resulting fields into a `text` array. If *delimiter* is NULL, each character in the *string* will become a separate element in the array. If *delimiter* is an empty string, then the *string* is treated as a single field. If *null_string* is supplied and is not NULL, fields matching that string are replaced by NULL. See also `array_to_string`.
`string_to_array('xx~~yy~~zz', '~~', 'yy')` → `{xx,NULL,zz}`

`string_to_table` (*string* `text`, *delimiter* `text` [, *null_string* `text`]) → `setof text`
Splits the *string* at occurrences of *delimiter* and returns the resulting fields as a set of `text` rows. If *delimiter* is NULL, each character in the *string* will become a separate row of the result. If *delimiter* is an empty string, then the *string* is treated as a single field. If *null_string* is supplied and is not NULL, fields matching that string are replaced by NULL.
`string_to_table('xx~^~yy~^~zz', '~^~', 'yy')` →
`xx`
`NULL`

Function/Operator **Description** **Example(s)**
zz

`strpos` (*`string`* `text`, *`substring`* `text`) → `integer`
Returns first starting index of the specified *`substring`* within *`string`*, or zero if it's not present. (Same as `position(`*`substring`*` in `*`string`*`)`, but note the reversed argument order.)
`strpos('high', 'ig')` → `2`

`substr` (*`string`* `text`, *`start`* `integer` [, *`count`* `integer`]) → `text`
Extracts the substring of *`string`* starting at the *`start`*'th character, and extending for *`count`* characters if that is specified. (Same as `substring(`*`string`*` from `*`start`*` for `*`count`*`)`.)
`substr('alphabet', 3)` → `phabet`
`substr('alphabet', 3, 2)` → `ph`

`to_ascii` (*`string`* `text`) → `text`
`to_ascii` (*`string`* `text`, *`encoding`* `name`) → `text`
`to_ascii` (*`string`* `text`, *`encoding`* `integer`) → `text`
Converts *`string`* to ASCII from another encoding, which may be identified by name or number. If *`encoding`* is omitted the database encoding is assumed (which in practice is the only useful case). The conversion consists primarily of dropping accents. Conversion is only supported from `LATIN1`, `LATIN2`, `LATIN9`, and `WIN1250` encodings. (See the unaccent module for another, more flexible solution.)
`to_ascii('Karél')` → `Karel`

`to_bin` (`integer`) → `text`
`to_bin` (`bigint`) → `text`
Converts the number to its equivalent two's complement binary representation.
`to_bin(2147483647)` → `11111111111111111111111111111111`
`to_bin(-1234)` → `11111111111111111111101100101110`

`to_hex` (`integer`) → `text`
`to_hex` (`bigint`) → `text`
Converts the number to its equivalent two's complement hexadecimal representation.
`to_hex(2147483647)` → `7fffffff`
`to_hex(-1234)` → `fffffb2e`

`to_oct` (`integer`) → `text`
`to_oct` (`bigint`) → `text`
Converts the number to its equivalent two's complement octal representation.
`to_oct(2147483647)` → `17777777777`
`to_oct(-1234)` → `37777775456`

`translate` (*`string`* `text`, *`from`* `text`, *`to`* `text`) → `text`
Replaces each character in *`string`* that matches a character in the *`from`* set with the corresponding character in the *`to`* set. If *`from`* is longer than *`to`*, occurrences of the extra characters in *`from`* are deleted.
`translate('12345', '143', 'ax')` → `a2x5`

`unistr` (`text`) → `text`

Function/Operator Description Example(s)	

	Evaluate escaped Unicode characters in the argument. Unicode characters can be specified as `\`*`XXXX`* (4 hexadecimal digits), `\+`*`XXXXXX`* (6 hexadecimal digits), `\u`*`XXXX`* (4 hexadecimal digits), or `\U`*`XXXXXXXX`* (8 hexadecimal digits). To specify a backslash, write two backslashes. All other characters are taken literally.
	If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that's not possible.
	This function provides a (non-standard) alternative to string constants with Unicode escapes (see Section 4.1.2.3).
	`unistr('d\0061t\+000061')` → `data`
	`unistr('d\u0061t\U00000061')` → `data`

The `concat`, `concat_ws` and `format` functions are variadic, so it is possible to pass the values to be concatenated or formatted as an array marked with the `VARIADIC` keyword (see Section 36.5.6). The array's elements are treated as if they were separate ordinary arguments to the function. If the variadic array argument is NULL, `concat` and `concat_ws` return NULL, but `format` treats a NULL as a zero-element array.

See also the aggregate function `string_agg` in Section 9.21, and the functions for converting between strings and the `bytea` type in Table 9.13.

9.4.1. `format`

The function `format` produces output formatted according to a format string, in a style similar to the C function `sprintf`.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

`formatstr` is a format string that specifies how the result should be formatted. Text in the format string is copied directly to the result, except where *format specifiers* are used. Format specifiers act as placeholders in the string, defining how subsequent function arguments should be formatted and inserted into the result. Each *`formatarg`* argument is converted to text according to the usual output rules for its data type, and then formatted and inserted into the result string according to the format specifier(s).

Format specifiers are introduced by a `%` character and have the form

```
%[position][flags][width]type
```

where the component fields are:

`position` (optional)

A string of the form *`n`*`$` where *`n`* is the index of the argument to print. Index 1 means the first argument after *`formatstr`*. If the *`position`* is omitted, the default is to use the next argument in sequence.

`flags` (optional)

Additional options controlling how the format specifier's output is formatted. Currently the only supported flag is a minus sign (`-`) which will cause the format specifier's output to be left-justified. This has no effect unless the *`width`* field is also specified.

width (optional)

Specifies the *minimum* number of characters to use to display the format specifier's output. The output is padded on the left or right (depending on the - flag) with spaces as needed to fill the width. A too-small width does not cause truncation of the output, but is simply ignored. The width may be specified using any of the following: a positive integer; an asterisk (*) to use the next function argument as the width; or a string of the form *n$ to use the *n*th function argument as the width.

If the width comes from a function argument, that argument is consumed before the argument that is used for the format specifier's value. If the width argument is negative, the result is left aligned (as if the - flag had been specified) within a field of length abs(*width*).

type (required)

The type of format conversion to use to produce the format specifier's output. The following types are supported:

- s formats the argument value as a simple string. A null value is treated as an empty string.
- I treats the argument value as an SQL identifier, double-quoting it if necessary. It is an error for the value to be null (equivalent to quote_ident).
- L quotes the argument value as an SQL literal. A null value is displayed as the string NULL, without quotes (equivalent to quote_nullable).

In addition to the format specifiers described above, the special sequence %% may be used to output a literal % character.

Here are some examples of the basic format conversions:

```
SELECT format('Hello %s', 'World');
Result: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Result: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O
\'Reilly');
Result: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program
 Files');
Result: INSERT INTO locations VALUES('C:\Program Files')
```

Here are examples using *width* fields and the - flag:

```
SELECT format('|%10s|', 'foo');
Result: |       foo|

SELECT format('|%-10s|', 'foo');
Result: |foo       |

SELECT format('|%*s|', 10, 'foo');
Result: |       foo|

SELECT format('|%*s|', -10, 'foo');
```

```
Result: |foo       |

SELECT format('|%-*s|', 10, 'foo');
Result: |foo       |

SELECT format('|%-*s|', -10, 'foo');
Result: |foo       |
```

These examples show use of *position* fields:

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Result: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Result: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
Result: |       foo|
```

Unlike the standard C function `sprintf`, PostgreSQL's `format` function allows format specifiers with and without *position* fields to be mixed in the same format string. A format specifier without a *position* field always uses the next argument after the last argument consumed. In addition, the `format` function does not require all function arguments to be used in the format string. For example:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Result: Testing three, two, three
```

The `%I` and `%L` format specifiers are particularly useful for safely constructing dynamic SQL statements. See Example 41.1.

9.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating binary strings, that is values of type `bytea`. Many of these are equivalent, in purpose and syntax, to the text-string functions described in the previous section.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9.11. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9.12).

Table 9.11. SQL Binary String Functions and Operators

Function/Operator **Description** **Example(s)**

`bytea || bytea → bytea`
Concatenates the two binary strings.
`'\x123456'::bytea || '\x789a00bcde'::bytea → \x123456789a00bcde`

`bit_length ( bytea ) → integer`
Returns number of bits in the binary string (8 times the `octet_length`).
`bit_length('\x123456'::bytea) → 24`

`btrim ( bytes bytea, bytesremoved bytea ) → bytea`

Function/Operator Description Example(s)
Removes the longest string containing only bytes appearing in *bytesremoved* from the start and end of *bytes*. `btrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678`

`ltrim` (*bytes* `bytea`, *bytesremoved* `bytea`) → `bytea`
Removes the longest string containing only bytes appearing in *bytesremoved* from the start of *bytes*.
`ltrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x34567890`

`octet_length` (`bytea`) → `integer`
Returns number of bytes in the binary string.
`octet_length('\x123456'::bytea)` → `3`

`overlay` (*bytes* `bytea` `PLACING` *newsubstring* `bytea` `FROM` *start* `integer` [`FOR` *count* `integer`]) → `bytea`
Replaces the substring of *bytes* that starts at the *start*'th byte and extends for *count* bytes with *newsubstring*. If *count* is omitted, it defaults to the length of *newsubstring*.
`overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3)` → `\x12020390`

`position` (*substring* `bytea` `IN` *bytes* `bytea`) → `integer`
Returns first starting index of the specified *substring* within *bytes*, or zero if it's not present.
`position('\x5678'::bytea in '\x1234567890'::bytea)` → `3`

`rtrim` (*bytes* `bytea`, *bytesremoved* `bytea`) → `bytea`
Removes the longest string containing only bytes appearing in *bytesremoved* from the end of *bytes*.
`rtrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x12345678`

`substring` (*bytes* `bytea` [`FROM` *start* `integer`] [`FOR` *count* `integer`]) → `bytea`
Extracts the substring of *bytes* starting at the *start*'th byte if that is specified, and stopping after *count* bytes if that is specified. Provide at least one of *start* and *count*.
`substring('\x1234567890'::bytea from 3 for 2)` → `\x5678`

`trim` ([`LEADING` | `TRAILING` | `BOTH`] *bytesremoved* `bytea` `FROM` *bytes* `bytea`) → `bytea`
Removes the longest string containing only bytes appearing in *bytesremoved* from the start, end, or both ends (`BOTH` is the default) of *bytes*.
`trim('\x9012'::bytea from '\x1234567890'::bytea)` → `\x345678`

`trim` ([`LEADING` | `TRAILING` | `BOTH`] [`FROM`] *bytes* `bytea`, *bytesremoved* `bytea`) → `bytea`
This is a non-standard syntax for `trim()`.
`trim(both from '\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678`

Additional binary string manipulation functions are available and are listed in Table 9.12. Some of them are used internally to implement the SQL-standard string functions listed in Table 9.11.

Table 9.12. Other Binary String Functions

Function Description Example(s)

`bit_count ( bytes bytea ) → bigint`
Returns the number of bits set in the binary string (also known as “popcount”).
`bit_count('\x1234567890'::bytea) → 15`

`get_bit ( bytes bytea, n bigint ) → integer`
Extracts n'th bit from binary string.
`get_bit('\x1234567890'::bytea, 30) → 1`

`get_byte ( bytes bytea, n integer ) → integer`
Extracts n'th byte from binary string.
`get_byte('\x1234567890'::bytea, 4) → 144`

`length ( bytea ) → integer`
Returns the number of bytes in the binary string.
`length('\x1234567890'::bytea) → 5`

`length ( bytes bytea, encoding name ) → integer`
Returns the number of characters in the binary string, assuming that it is text in the given *encoding*.
`length('jose'::bytea, 'UTF8') → 4`

`md5 ( bytea ) → text`
Computes the MD5 hash of the binary string, with the result written in hexadecimal.
`md5('Th\000omas'::bytea) → 8ab2d3c9689aaf18b4958c334c82d8b1`

`set_bit ( bytes bytea, n bigint, newvalue integer ) → bytea`
Sets n'th bit in binary string to *newvalue*.
`set_bit('\x1234567890'::bytea, 30, 0) → \x1234563890`

`set_byte ( bytes bytea, n integer, newvalue integer ) → bytea`
Sets n'th byte in binary string to *newvalue*.
`set_byte('\x1234567890'::bytea, 4, 64) → \x1234567840`

`sha224 ( bytea ) → bytea`
Computes the SHA-224 hash of the binary string.
`sha224('abc'::bytea) → \x23097d223405d8228642a477bda2 55b32aadbce4bda0b3f7e36c9da7`

`sha256 ( bytea ) → bytea`
Computes the SHA-256 hash of the binary string.
`sha256('abc'::bytea) → \xba7816bf8f01cfea414140de5dae2223 b00361a396177a9cb410ff61f20015ad`

`sha384 ( bytea ) → bytea`
Computes the SHA-384 hash of the binary string.
`sha384('abc'::bytea) → \xcb00753f45a35e8bb5a03d699ac65007 272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358bae- ca134c825a7`

`sha512 ( bytea ) → bytea` Computes the SHA-512 hash of the binary string.

Function Description Example(s)
sha512('abc'::bytea) → \xddaf35a193617abac- c417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a 2192992a274fc1a836ba3c23a3feebbd 454d4423643ce80e2a9ac94fa54ca49f
substr (*bytes* bytea, *start* integer [, *count* integer]) → bytea Extracts the substring of *bytes* starting at the *start*'th byte, and extending for *count* bytes if that is specified. (Same as substring(*bytes* from *start* for *count*).) substr('\x1234567890'::bytea, 3, 2) → \x5678

Functions get_byte and set_byte number the first byte of a binary string as byte 0. Functions get_bit and set_bit number bits from the right within each byte; for example bit 0 is the least significant bit of the first byte, and bit 15 is the most significant bit of the second byte.

For historical reasons, the function md5 returns a hex-encoded value of type text whereas the SHA-2 functions return type bytea. Use the functions encode and decode to convert between the two. For example write encode(sha256('abc'), 'hex') to get a hex-encoded text representation, or decode(md5('abc'), 'hex') to get a bytea value.

Functions for converting strings between different character sets (encodings), and for representing arbitrary binary data in textual form, are shown in Table 9.13. For these functions, an argument or result of type text is expressed in the database's default encoding, while arguments or results of type bytea are in an encoding named by another argument.

Table 9.13. Text/Binary String Conversion Functions

Function Description Example(s)
convert (*bytes* bytea, *src_encoding* name, *dest_encoding* name) → bytea Converts a binary string representing text in encoding *src_encoding* to a binary string in encoding *dest_encoding* (see Section 23.3.4 for available conversions). convert('text_in_utf8', 'UTF8', 'LATIN1') → \x746578745f696e5f75746638
convert_from (*bytes* bytea, *src_encoding* name) → text Converts a binary string representing text in encoding *src_encoding* to text in the database encoding (see Section 23.3.4 for available conversions). convert_from('text_in_utf8', 'UTF8') → text_in_utf8
convert_to (*string* text, *dest_encoding* name) → bytea Converts a text string (in the database encoding) to a binary string encoded in encoding *dest_encoding* (see Section 23.3.4 for available conversions). convert_to('some_text', 'UTF8') → \x736f6d655f74657874
encode (*bytes* bytea, *format* text) → text Encodes binary data into a textual representation; supported *format* values are: base64, escape, hex. encode('123\000\001', 'base64') → MTIzAAE=
decode (*string* text, *format* text) → bytea Decodes binary data from a textual representation; supported *format* values are the same as for encode.

Function Description Example(s)
`decode('MTIzAAE=', 'base64')` → `\x3132330001`

The `encode` and `decode` functions support the following textual formats:

base64

The `base64` format is that of RFC 2045 Section 6.8[1]. As per the RFC, encoded lines are broken at 76 characters. However instead of the MIME CRLF end-of-line marker, only a newline is used for end-of-line. The `decode` function ignores carriage-return, newline, space, and tab characters. Otherwise, an error is raised when `decode` is supplied invalid base64 data — including when trailing padding is incorrect.

escape

The `escape` format converts zero bytes and bytes with the high bit set into octal escape sequences (`\`*nnn*), and it doubles backslashes. Other byte values are represented literally. The `decode` function will raise an error if a backslash is not followed by either a second backslash or three octal digits; it accepts other byte values unchanged.

hex

The `hex` format represents each 4 bits of data as one hexadecimal digit, `0` through `f`, writing the higher-order digit of each byte first. The `encode` function outputs the `a-f` hex digits in lower case. Because the smallest unit of data is 8 bits, there are always an even number of characters returned by `encode`. The `decode` function accepts the `a-f` characters in either upper or lower case. An error is raised when `decode` is given invalid hex data — including when given an odd number of characters.

See also the aggregate function `string_agg` in Section 9.21 and the large object functions in Section 33.4.

9.6. Bit String Functions and Operators

This section describes functions and operators for examining and manipulating bit strings, that is values of the types `bit` and `bit varying`. (While only type `bit` is mentioned in these tables, values of type `bit varying` can be used interchangeably.) Bit strings support the usual comparison operators shown in Table 9.1, as well as the operators shown in Table 9.14.

Table 9.14. Bit String Operators

Operator Description Example(s)

`bit || bit` → `bit`
Concatenation
`B'10001' || B'011'` → `10001011`

`bit & bit` → `bit`
Bitwise AND (inputs must be of equal length)
`B'10001' & B'01101'` → `00001`

`bit | bit` → `bit` Bitwise OR (inputs must be of equal length)

[1] https://datatracker.ietf.org/doc/html/rfc2045#section-6.8

Operator Description Example(s)
`B'10001' | B'01101'` → `11101`

`bit # bit` → `bit`
Bitwise exclusive OR (inputs must be of equal length)
`B'10001' # B'01101'` → `11100`

`~ bit` → `bit`
Bitwise NOT
`~ B'10001'` → `01110`

`bit << integer` → `bit`
Bitwise shift left (string length is preserved)
`B'10001' << 3` → `01000`

`bit >> integer` → `bit`
Bitwise shift right (string length is preserved)
`B'10001' >> 2` → `00100`

Some of the functions available for binary strings are also available for bit strings, as shown in Table 9.15.

Table 9.15. Bit String Functions

Function Description Example(s)

`bit_count ( bit )` → `bigint`
Returns the number of bits set in the bit string (also known as “popcount”).
`bit_count(B'10111')` → `4`

`bit_length ( bit )` → `integer`
Returns number of bits in the bit string.
`bit_length(B'10111')` → `5`

`length ( bit )` → `integer`
Returns number of bits in the bit string.
`length(B'10111')` → `5`

`octet_length ( bit )` → `integer`
Returns number of bytes in the bit string.
`octet_length(B'1011111011')` → `2`

`overlay ( bits bit PLACING newsubstring bit FROM start integer [ FOR count integer ] )` → `bit`
Replaces the substring of *`bits`* that starts at the *`start`*'th bit and extends for *`count`* bits with *`newsubstring`*. If *`count`* is omitted, it defaults to the length of *`newsubstring`*.
`overlay(B'01010101010101010' placing B'11111' from 2 for 3)` → `0111110101010101010`

`position ( substring bit IN bits bit )` → `integer` Returns first starting index of the specified *`substring`* within *`bits`*, or zero if it's not present.

Function Description Example(s)
`position(B'010' in B'000001101011')` → `8`

`substring(`*`bits`* `bit [ FROM`*`start`* `integer ] [ FOR`*`count`* `integer ])` → `bit`
Extracts the substring of *`bits`* starting at the *`start`*'th bit if that is specified, and stopping after *`count`* bits if that is specified. Provide at least one of *`start`* and *`count`*.
`substring(B'110010111111' from 3 for 2)` → `00`

`get_bit(`*`bits`* `bit,`*`n`* `integer )` → `integer`
Extracts *n*'th bit from bit string; the first (leftmost) bit is bit 0.
`get_bit(B'101010101010101010', 6)` → `1`

`set_bit(`*`bits`* `bit,`*`n`* `integer,`*`newvalue`* `integer )` → `bit`
Sets *n*'th bit in bit string to *`newvalue`*; the first (leftmost) bit is bit 0.
`set_bit(B'101010101010101010', 6, 0)` → `101010001010101010`

In addition, it is possible to cast integral values to and from type `bit`. Casting an integer to `bit(n)` copies the rightmost `n` bits. Casting an integer to a bit string width wider than the integer itself will sign-extend on the left. Some examples:

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Note that casting to just "bit" means casting to `bit(1)`, and so will deliver only the least significant bit of the integer.

9.7. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), and POSIX-style regular expressions. Aside from the basic "does this string match this pattern?" operators, functions are available to extract or replace matching substrings and to split a string at matching locations.

> **Tip**
>
> If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

> **Caution**
>
> While most regular-expression searches can be executed very quickly, regular expressions can be contrived that take arbitrary amounts of time and memory to process. Be wary of accepting regular-expression search patterns from hostile sources. If you must do so, it is advisable to impose a statement timeout.
>
> Searches using `SIMILAR TO` patterns have the same security hazards, since `SIMILAR TO` provides many of the same capabilities as POSIX-style regular expressions.
>
> `LIKE` searches, being much simpler than the other two options, are safer to use with possibly-hostile pattern sources.

The pattern matching operators of all three kinds do not support nondeterministic collations. If required, apply a different collation to the expression to work around this limitation.

9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

The LIKE expression returns true if the *string* matches the supplied *pattern*. (As expected, the NOT LIKE expression returns false if LIKE returns true, and vice versa. An equivalent expression is NOT (*string* LIKE *pattern*).)

If *pattern* does not contain percent signs or underscores, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (%) matches any sequence of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *pattern* must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

> **Note**
>
> If you have standard_conforming_strings turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

According to the SQL standard, omitting ESCAPE means there is no escape character (rather than defaulting to a backslash), and a zero-length ESCAPE value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

The key word ILIKE can be used instead of LIKE to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator ~~ is equivalent to LIKE, and ~~* corresponds to ILIKE. There are also !~~ and !~~* operators that represent NOT LIKE and NOT ILIKE, respectively. All of these operators are PostgreSQL-specific. You may see these operator names in EXPLAIN output and similar places, since the parser actually translates LIKE et al. to these operators.

The phrases LIKE, ILIKE, NOT LIKE, and NOT ILIKE are generally treated as operators in PostgreSQL syntax; for example they can be used in *expression operator* ANY (*subquery*) constructs, although an ESCAPE clause cannot be included there. In some obscure cases it may be necessary to use the underlying operator names instead.

Also see the starts-with operator ^@ and the corresponding `starts_with()` function, which are useful in cases where simply matching the beginning of a string is needed.

9.7.2. SIMILAR TO Regular Expressions

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between `LIKE` notation and common (POSIX) regular expression notation.

Like `LIKE`, the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like `LIKE`, `SIMILAR TO` uses _ and % as wildcard characters denoting any single character and any string, respectively (these are comparable to `.` and `.*` in POSIX regular expressions).

In addition to these facilities borrowed from `LIKE`, `SIMILAR TO` supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- `|` denotes alternation (either of two alternatives).
- `*` denotes repetition of the previous item zero or more times.
- `+` denotes repetition of the previous item one or more times.
- `?` denotes repetition of the previous item zero or one time.
- `{`*m*`}` denotes repetition of the previous item exactly *m* times.
- `{`*m*`,}` denotes repetition of the previous item *m* or more times.
- `{`*m*`,`*n*`}` denotes repetition of the previous item at least *m* and not more than *n* times.
- Parentheses `()` can be used to group items into a single logical item.
- A bracket expression `[...]` specifies a character class, just as in POSIX regular expressions.

Notice that the period (`.`) is not a metacharacter for `SIMILAR TO`.

As with `LIKE`, a backslash disables the special meaning of any of these metacharacters. A different escape character can be specified with `ESCAPE`, or the escape capability can be disabled by writing `ESCAPE ''`.

According to the SQL standard, omitting `ESCAPE` means there is no escape character (rather than defaulting to a backslash), and a zero-length `ESCAPE` value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

Another nonstandard extension is that following the escape character with a letter or digit provides access to the escape sequences defined for POSIX regular expressions; see Table 9.20, Table 9.21, and Table 9.22 below.

Some examples:

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
```

```
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

The `substring` function with three parameters provides extraction of a substring that matches an SQL regular expression pattern. The function can be written according to standard SQL syntax:

```
substring(string similar pattern escape escape-character)
```

or using the now obsolete SQL:1999 syntax:

```
substring(string from pattern for escape-character)
```

or as a plain three-argument function:

```
substring(string, pattern, escape-character)
```

As with `SIMILAR TO`, the specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern for which the matching data sub-string is of interest, the pattern should contain two occurrences of the escape character followed by a double quote (`"`). The text matching the portion of the pattern between these separators is returned when the match is successful.

The escape-double-quote separators actually divide `substring`'s pattern into three independent regular expressions; for example, a vertical bar (`|`) in any of the three sections affects only that section. Also, the first and third of these regular expressions are defined to match the smallest possible amount of text, not the largest, when there is any ambiguity about how much of the data string matches which pattern. (In POSIX parlance, the first and third regular expressions are forced to be non-greedy.)

As an extension to the SQL standard, PostgreSQL allows there to be just one escape-double-quote separator, in which case the third regular expression is taken as empty; or no separators, in which case the first and third regular expressions are taken as empty.

Some examples, with `#"` delimiting the return string:

```
substring('foobar' similar '%#"o_b#"%' escape '#')   oob
substring('foobar' similar '#"o_b#"%' escape '#')    NULL
```

9.7.3. POSIX Regular Expressions

Table 9.16 lists the available operators for pattern matching using POSIX regular expressions.

Table 9.16. Regular Expression Match Operators

Operator **Description** **Example(s)**

`text ~ text → boolean`
String matches regular expression, case sensitively
`'thomas' ~ 't.*ma' → t`

`text ~* text → boolean`
String matches regular expression, case-insensitively
`'thomas' ~* 'T.*ma' → t`

`text !~ text → boolean` String does not match regular expression, case sensitively

Operator Description Example(s)
`'thomas' !~ 't.*max'` → `t`

`text !~* text` → `boolean`
String does not match regular expression, case-insensitively
`'thomas' !~* 'T.*ma'` → `f`

POSIX regular expressions provide a more powerful means for pattern matching than the `LIKE` and `SIMILAR TO` operators. Many Unix tools such as `egrep`, `sed`, or `awk` use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular set*). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with `LIKE`, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than `LIKE` does. Unlike `LIKE` patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

```
'abcd' ~ 'bc'     true
'abcd' ~ 'a.c'    true — dot matches any character
'abcd' ~ 'a.*d'   true — * repeats the preceding pattern item
'abcd' ~ '(b|x)'  true — | means OR, parentheses group
'abcd' ~ '^a'     true — ^ anchors to start of string
'abcd' ~ '^(b|c)' false — would match except for anchoring
```

The POSIX pattern language is described in much greater detail below.

The `substring` function with two parameters, `substring(`*`string`* `from` *`pattern`*`)`, provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the first portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

The `regexp_count` function counts the number of places where a POSIX regular expression pattern matches a string. It has the syntax `regexp_count(`*`string`*`,` *`pattern`* `[, ` *`start`* `[, ` *`flags`* `]])`. *`pattern`* is searched for in *`string`*, normally from the beginning of the string, but if the *`start`* parameter is provided then beginning from that character index. The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. For example, including `i` in *`flags`* specifies case-insensitive matching. Supported flags are described in Table 9.24.

Some examples:

```
regexp_count('ABCABCAXYaxy', 'A.')          3
regexp_count('ABCABCAXYaxy', 'A.', 1, 'i')  4
```

The `regexp_instr` function returns the starting or ending position of the *N*'th match of a POSIX regular expression pattern to a string, or zero if there is no such match. It has the syntax `regexp_instr(`*`string`*`, `*`pattern`*` [, `*`start`*` [, `*`N`*` [, `*`endoption`*` [, `*`flags`*` [, `*`subexpr`*` ]]]]]). *`pattern`* is searched for in *`string`*, normally from the beginning of the string, but if the *`start`* parameter is provided then beginning from that character index. If *`N`* is specified then the *`N`*'th match of the pattern is located, otherwise the first match is located. If the *`endoption`* parameter is omitted or specified as zero, the function returns the position of the first character of the match. Otherwise, *`endoption`* must be one, and the function returns the position of the character following the match. The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24. For a pattern containing parenthesized subexpressions, *`subexpr`* is an integer indicating which subexpression is of interest: the result identifies the position of the substring matching that subexpression. Subexpressions are numbered in the order of their leading parentheses. When *`subexpr`* is omitted or zero, the result identifies the position of the whole match regardless of parenthesized subexpressions.

Some examples:

```
regexp_instr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                    23
regexp_instr('ABCDEFGHI', '(c..)(...)', 1, 1, 0, 'i', 2)
                                    6
```

The `regexp_like` function checks whether a match of a POSIX regular expression pattern occurs within a string, returning boolean true or false. It has the syntax `regexp_like(`*`string`*`, `*`pattern`*` [, `*`flags`*` ]). The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24. This function has the same results as the `~` operator if no flags are specified. If only the `i` flag is specified, it has the same results as the `~*` operator.

Some examples:

```
regexp_like('Hello World', 'world')       false
regexp_like('Hello World', 'world', 'i')  true
```

The `regexp_match` function returns a text array of matching substring(s) within the first match of a POSIX regular expression pattern to a string. It has the syntax `regexp_match(`*`string`*`, `*`pattern`*` [, `*`flags`*` ]). If there is no match, the result is `NULL`. If a match is found, and the *`pattern`* contains no parenthesized subexpressions, then the result is a single-element text array containing the substring matching the whole pattern. If a match is found, and the *`pattern`* contains parenthesized subexpressions, then the result is a text array whose *`n`*'th element is the substring matching the *`n`*'th parenthesized subexpression of the *`pattern`* (not counting “non-capturing” parentheses; see below for details). The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24.

Some examples:

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

> **Tip**
>
> In the common case where you just want the whole matching substring or `NULL` for no match, the best solution is to use `regexp_substr()`. However, `regexp_substr()` only exists in PostgreSQL version 15 and up. When working in older versions, you can extract the first element of `regexp_match()`'s result, for example:
>
> ```
> SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
>  regexp_match
> --------------
>  barbeque
> (1 row)
> ```

The `regexp_matches` function returns a set of text arrays of matching substring(s) within matches of a POSIX regular expression pattern to a string. It has the same syntax as `regexp_match`. This function returns no rows if there is no match, one row if there is a match and the `g` flag is not given, or *`N`* rows if there are *`N`* matches and the `g` flag is given. Each returned row is a text array containing the whole matched substring or the substrings matching parenthesized subexpressions of the *`pattern`*, just as described above for `regexp_match`. `regexp_matches` accepts all the flags shown in Table 9.24, plus the `g` flag which commands it to return all matches, not just the first one.

Some examples:

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)
(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

> **Tip**
>
> In most cases `regexp_matches()` should be used with the `g` flag, since if you only want the first match, it's easier and more efficient to use `regexp_match()`. However, `regexp_match()` only exists in PostgreSQL version 10 and up. When working in older versions, a common trick is to place a `regexp_matches()` call in a sub-select, for example:
>
> ```
> SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)'))
>  FROM tab;
> ```
>
> This produces a text array if there's a match, or `NULL` if not, the same as `regexp_match()` would do. Without the sub-select, this query would produce no output at all for table rows without a match, which is typically not the desired behavior.

The `regexp_replace` function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax `regexp_replace(`*`source`*`, `*`pattern`*`, `*`replacement`* [, *`start`* [, *`N`*]] [, *`flags`*]). (Notice that *`N`* cannot be specified unless *`start`* is, but *`flags`* can be given in any case.) The *`source`* string is returned unchanged if there is no match to the *`pat-`*

tern. If there is a match, the *source* string is returned with the *replacement* string substituted for the matching substring. The *replacement* string can contain *n*, where *n* is 1 through 9, to indicate that the source substring matching the *n*'th parenthesized subexpression of the pattern should be inserted, and it can contain \& to indicate that the substring matching the entire pattern should be inserted. Write \\ if you need to put a literal backslash in the replacement text. *pattern* is searched for in *string*, normally from the beginning of the string, but if the *start* parameter is provided then beginning from that character index. By default, only the first match of the pattern is replaced. If *N* is specified and is greater than zero, then the *N*'th match of the pattern is replaced. If the g flag is given, or if *N* is specified and is zero, then all matches at or after the *start* position are replaced. (The g flag is ignored when *N* is specified.) The *flags* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags (though not g) are described in Table 9.24.

Some examples:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                    fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                    fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                    fooXarYXazY
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 0,
 'i')
                                    X PXstgrXSQL fXnctXXn
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 3,
 'i')
                                    A PostgrXSQL function
```

The regexp_split_to_table function splits a string using a POSIX regular expression pattern as a delimiter. It has the syntax regexp_split_to_table(*string*, *pattern* [, *flags*]). If there is no match to the *pattern*, the function returns the *string*. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string. The *flags* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. regexp_split_to_table supports the flags described in Table 9.24.

The regexp_split_to_array function behaves the same as regexp_split_to_table, except that regexp_split_to_array returns its result as an array of text. It has the syntax regexp_split_to_array(*string*, *pattern* [, *flags*]). The parameters are the same as for regexp_split_to_table.

Some examples:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps
 over the lazy dog', '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)
```

```
SELECT regexp_split_to_array('the quick brown fox jumps over the
 lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*')
 AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

As the last example demonstrates, the regexp split functions ignore zero-length matches that occur at the start or end of the string or immediately after a previous match. This is contrary to the strict definition of regexp matching that is implemented by the other regexp functions, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

The `regexp_substr` function returns the substring that matches a POSIX regular expression pattern, or `NULL` if there is no match. It has the syntax `regexp_substr(`*`string`*`,`*`pattern`*` [, `*`start`*` [, `*`N`*` [, `*`flags`*` [, `*`subexpr`*` ]]]])`. *`pattern`* is searched for in *`string`*, normally from the beginning of the string, but if the *`start`* parameter is provided then beginning from that character index. If *`N`* is specified then the *`N`*'th match of the pattern is returned, otherwise the first match is returned. The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24. For a pattern containing parenthesized subexpressions, *`subexpr`* is an integer indicating which subexpression is of interest: the result is the substring matching that subexpression. Subexpressions are numbered in the order of their leading parentheses. When *`subexpr`* is omitted or zero, the result is the whole match regardless of parenthesized subexpressions.

Some examples:

```
regexp_substr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                    town zip
regexp_substr('ABCDEFGHI', '(c..)(...)', 1, 1, 'i', 2)
                                   FGH
```

9.7.3.1. Regular Expression Details

PostgreSQL's regular expressions are implemented using a software package written by Henry Spencer. Much of the description of regular expressions below is copied verbatim from his manual.

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: *extended* REs or EREs (roughly those of `egrep`), and *basic* REs or BREs (roughly those of `ed`). PostgreSQL supports both forms, and also implements some extensions that are not in the POSIX standard, but have become widely used due to their availability in programming languages such as Perl and Tcl. REs using these non-POSIX extensions are called *advanced* REs or AREs in this documentation. AREs are almost an exact superset of EREs, but BREs have several notational incompatibilities (as well as being much more limited). We first describe the ARE and ERE forms, noting features that apply only to AREs, and then describe how BREs differ.

> **Note**
>
> PostgreSQL always initially presumes that a regular expression follows the ARE rules. However, the more limited ERE or BRE rules can be chosen by prepending an *embedded option* to the RE pattern, as described in Section 9.7.3.4. This can be useful for compatibility with applications that expect exactly the POSIX 1003.2 rules.

A regular expression is defined as one or more *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more *quantified atoms* or *constraints*, concatenated. It matches a match for the first, followed by a match for the second, etc.; an empty branch matches the empty string.

A quantified atom is an *atom* possibly followed by a single *quantifier*. Without a quantifier, it matches a match for the atom. With a quantifier, it can match some number of matches of the atom. An *atom* can be any of the possibilities shown in Table 9.17. The possible quantifiers and their meanings are shown in Table 9.18.

A *constraint* matches an empty string, but matches only when specific conditions are met. A constraint can be used where an atom could be used, except it cannot be followed by a quantifier. The simple constraints are shown in Table 9.19; some more constraints are described later.

Table 9.17. Regular Expression Atoms

Atom	Description
`(`*re*`)`	(where *re* is any regular expression) matches a match for *re*, with the match noted for possible reporting
`(?:`*re*`)`	as above, but the match is not noted for reporting (a “non-capturing” set of parentheses) (AREs only)
`.`	matches any single character
`[`*chars*`]`	a *bracket expression*, matching any one of the *chars* (see Section 9.7.3.2 for more detail)
`\`*k*	(where *k* is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., `\\` matches a backslash character
`\`*c*	where *c* is alphanumeric (possibly followed by other characters) is an *escape*, see Section 9.7.3.3 (AREs only; in EREs and BREs, this matches *c*)
`{`	when followed by a character other than a digit, matches the left-brace character `{`; when followed by a digit, it is the beginning of a *bound* (see below)

Atom	Description
x	where *x* is a single character with no other significance, matches that character

An RE cannot end with a backslash (\).

> **Note**
>
> If you have standard_conforming_strings turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

Table 9.18. Regular Expression Quantifiers

Quantifier	Matches
*	a sequence of 0 or more matches of the atom
+	a sequence of 1 or more matches of the atom
?	a sequence of 0 or 1 matches of the atom
{*m*}	a sequence of exactly *m* matches of the atom
{*m*,}	a sequence of *m* or more matches of the atom
{*m*,*n*}	a sequence of *m* through *n* (inclusive) matches of the atom; *m* cannot exceed *n*
*?	non-greedy version of *
+?	non-greedy version of +
??	non-greedy version of ?
{*m*}?	non-greedy version of {*m*}
{*m*,}?	non-greedy version of {*m*,}
{*m*,*n*}?	non-greedy version of {*m*,*n*}

The forms using {...} are known as *bounds*. The numbers *m* and *n* within a bound are unsigned decimal integers with permissible values from 0 to 255 inclusive.

Non-greedy quantifiers (available in AREs only) match the same possibilities as their corresponding normal (*greedy*) counterparts, but prefer the smallest number rather than the largest number of matches. See Section 9.7.3.5 for more detail.

> **Note**
>
> A quantifier cannot immediately follow another quantifier, e.g., ** is invalid. A quantifier cannot begin an expression or subexpression or follow ^ or |.

Table 9.19. Regular Expression Constraints

Constraint	Description
^	matches at the beginning of the string
$	matches at the end of the string
(?=*re*)	*positive lookahead* matches at any point where a substring matching *re* begins (AREs only)
(?!*re*)	*negative lookahead* matches at any point where no substring matching *re* begins (AREs only)

Constraint	Description
(?<=*re*)	*positive lookbehind* matches at any point where a substring matching *re* ends (AREs only)
(?<!*re*)	*negative lookbehind* matches at any point where no substring matching *re* ends (AREs only)

Lookahead and lookbehind constraints cannot contain *back references* (see Section 9.7.3.3), and all parentheses within them are considered non-capturing.

9.7.3.2. Bracket Expressions

A *bracket expression* is a list of characters enclosed in `[]`. It normally matches any single character from the list (but see below). If the list begins with `^`, it matches any single character *not* from the rest of the list. If two characters in the list are separated by `-`, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g., `[0-9]` in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g., `a-c-e`. Ranges are very collating-sequence-dependent, so portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (after `^`, if that is used). To include a literal `-`, make it the first or last character, or the second endpoint of a range. To use a literal `-` as the first endpoint of a range, enclose it in `[.` and `.]` to make it a collating element (see below). With the exception of these characters, some combinations using `[` (see next paragraphs), and escapes (AREs only), all other special characters lose their special significance within a bracket expression. In particular, `\` is not special when following ERE or BRE rules, though it is special (as introducing an escape) in AREs.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in `[.` and `.]` stands for the sequence of characters of that collating element. The sequence is treated as a single element of the bracket expression's list. This allows a bracket expression containing a multiple-character collating element to match more than one character, e.g., if the collating sequence includes a `ch` collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

> **Note**
>
> PostgreSQL currently does not support multi-character collating elements. This information describes possible future behavior.

Within a bracket expression, a collating element enclosed in `[=` and `=]` is an *equivalence class*, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.` and `.]`.) For example, if `o` and `^` are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class cannot be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters belonging to that class. A character class cannot be used as an endpoint of a range. The POSIX standard defines these character class names: `alnum` (letters and numeric digits), `alpha` (letters), `blank` (space and tab), `cntrl` (control characters), `digit` (numeric digits), `graph` (printable characters except space), `lower` (lower-case letters), `print` (printable characters including space), `punct` (punctuation), `space` (any white space), `upper` (upper-case letters), and `xdigit` (hexadecimal digits). The behavior of these standard character classes is generally consistent across platforms for characters in the 7-bit ASCII set. Whether a given non-ASCII character is considered to belong to one of these classes depends on the *collation* that is used for the regular-expression function or operator (see Section 23.2), or by default on the database's `LC_CTYPE` locale setting (see Section 23.1). The classification of non-ASCII characters can vary across platforms even in similarly-named locales. (But the `C` locale never considers any non-ASCII characters to belong to any of these classes.) In addition to these standard character classes, PostgreSQL defines the `word` character class, which is

the same as `alnum` plus the underscore (_) character, and the `ascii` character class, which contains exactly the 7-bit ASCII set.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is any character belonging to the `word` character class, that is, any letter, digit, or underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. The constraint escapes described below are usually preferable; they are no more standard, but are easier to type.

9.7.3.3. Regular Expression Escapes

Escapes are special sequences beginning with \ followed by an alphanumeric character. Escapes come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes exist to make it easier to specify non-printing and other inconvenient characters in REs. They are shown in Table 9.20.

Class-shorthand escapes provide shorthands for certain commonly-used character classes. They are shown in Table 9.21.

A *constraint escape* is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in Table 9.22.

A *back reference* (`\`*n*) matches the same string matched by the previous parenthesized subexpression specified by the number *n* (see Table 9.23). For example, `([bc])\1` matches `bb` or `cc` but not `bc` or `cb`. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions. The back reference considers only the string characters matched by the referenced subexpression, not any constraints contained in it. For example, `(^\d)\1` will match `22`.

Table 9.20. Regular Expression Character-Entry Escapes

Escape	Description
`\a`	alert (bell) character, as in C
`\b`	backspace, as in C
`\B`	synonym for backslash (\) to help reduce the need for backslash doubling
`\c`*X*	(where *X* is any character) the character whose low-order 5 bits are the same as those of *X*, and whose other bits are all zero
`\e`	the character whose collating-sequence name is `ESC`, or failing that, the character with octal value `033`
`\f`	form feed, as in C
`\n`	newline, as in C
`\r`	carriage return, as in C
`\t`	horizontal tab, as in C
`\u`*wxyz*	(where *wxyz* is exactly four hexadecimal digits) the character whose hexadecimal value is `0x`*wxyz*

Escape	Description
`\U`*stuvwxyz*	(where *stuvwxyz* is exactly eight hexadecimal digits) the character whose hexadecimal value is `0x`*stuvwxyz*
`\v`	vertical tab, as in C
`\x`*hhh*	(where *hhh* is any sequence of hexadecimal digits) the character whose hexadecimal value is `0x`*hhh* (a single character no matter how many hexadecimal digits are used)
`\0`	the character whose value is `0` (the null byte)
`\`*xy*	(where *xy* is exactly two octal digits, and is not a *back reference*) the character whose octal value is `0`*xy*
`\`*xyz*	(where *xyz* is exactly three octal digits, and is not a *back reference*) the character whose octal value is `0`*xyz*

Hexadecimal digits are `0-9`, `a-f`, and `A-F`. Octal digits are `0-7`.

Numeric character-entry escapes specifying values outside the ASCII range (0–127) have meanings dependent on the database encoding. When the encoding is UTF-8, escape values are equivalent to Unicode code points, for example `\u1234` means the character `U+1234`. For other multibyte encodings, character-entry escapes usually just specify the concatenation of the byte values for the character. If the escape value does not correspond to any legal character in the database encoding, no error will be raised, but it will never match any data.

The character-entry escapes are always taken as ordinary characters. For example, `\135` is `]` in ASCII, but `\135` does not terminate a bracket expression.

Table 9.21. Regular Expression Class-Shorthand Escapes

Escape	Description
`\d`	matches any digit, like `[[:digit:]]`
`\s`	matches any whitespace character, like `[[:space:]]`
`\w`	matches any word character, like `[[:word:]]`
`\D`	matches any non-digit, like `[^[:digit:]]`
`\S`	matches any non-whitespace character, like `[^[:space:]]`
`\W`	matches any non-word character, like `[^[:word:]]`

The class-shorthand escapes also work within bracket expressions, although the definitions shown above are not quite syntactically valid in that context. For example, `[a-c\d]` is equivalent to `[a-c[:digit:]]`.

Table 9.22. Regular Expression Constraint Escapes

Escape	Description
`\A`	matches only at the beginning of the string (see Section 9.7.3.5 for how this differs from `^`)
`\m`	matches only at the beginning of a word
`\M`	matches only at the end of a word

Escape	Description
\y	matches only at the beginning or end of a word
\Y	matches only at a point that is not the beginning or end of a word
\Z	matches only at the end of the string (see Section 9.7.3.5 for how this differs from $)

A word is defined as in the specification of [[:<:]] and [[:>:]] above. Constraint escapes are illegal within bracket expressions.

Table 9.23. Regular Expression Back References

Escape	Description
m	(where *m* is a nonzero digit) a back reference to the *m*'th subexpression
mnn	(where *m* is a nonzero digit, and *nn* is some more digits, and the decimal value *mnn* is not greater than the number of closing capturing parentheses seen so far) a back reference to the *mnn*'th subexpression

> **Note**
>
> There is an inherent ambiguity between octal character-entry escapes and back references, which is resolved by the following heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e., the number is in the legal range for a back reference), and otherwise is taken as octal.

9.7.3.4. Regular Expression Metasyntax

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

An RE can begin with one of two special *director* prefixes. If an RE begins with ***:, the rest of the RE is taken as an ARE. (This normally has no effect in PostgreSQL, since REs are assumed to be AREs; but it does have an effect if ERE or BRE mode had been specified by the *flags* parameter to a regex function.) If an RE begins with ***=, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE can begin with *embedded options*: a sequence (?*xyz*) (where *xyz* is one or more alphabetic characters) specifies options affecting the rest of the RE. These options override any previously determined options — in particular, they can override the case-sensitivity behavior implied by a regex operator, or the *flags* parameter to a regex function. The available option letters are shown in Table 9.24. Note that these same option letters are used in the *flags* parameters of regex functions.

Table 9.24. ARE Embedded-Option Letters

Option	Description
b	rest of RE is a BRE
c	case-sensitive matching (overrides operator type)
e	rest of RE is an ERE

Option	Description
i	case-insensitive matching (see Section 9.7.3.5) (overrides operator type)
m	historical synonym for n
n	newline-sensitive matching (see Section 9.7.3.5)
p	partial newline-sensitive matching (see Section 9.7.3.5)
q	rest of RE is a literal (“quoted”) string, all ordinary characters
s	non-newline-sensitive matching (default)
t	tight syntax (default; see below)
w	inverse partial newline-sensitive (“weird”) matching (see Section 9.7.3.5)
x	expanded syntax (see below)

Embedded options take effect at the) terminating the sequence. They can appear only at the start of an ARE (after the ***: director if any).

In addition to the usual (*tight*) RE syntax, in which all characters are significant, there is an *expanded* syntax, available by specifying the embedded x option. In the expanded syntax, white-space characters in the RE are ignored, as are all characters between a # and the following newline (or the end of the RE). This permits paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

- a white-space character or # preceded by \ is retained
- white space or # within a bracket expression is retained
- white space and comments cannot appear within multi-character symbols, such as (?:

For this purpose, white-space characters are blank, tab, newline, and any character that belongs to the *space* character class.

Finally, in an ARE, outside bracket expressions, the sequence (?#*ttt*) (where *ttt* is any text not containing a)) is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols, like (?:. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if an initial ***= director has specified that the user's input be treated as a literal string rather than as an RE.

9.7.3.5. Regular Expression Matching Rules

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is *greedy* or *non-greedy*.

Whether an RE is greedy or not is determined by the following rules:

- Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).
- Adding parentheses around an RE does not change its greediness.
- A quantified atom with a fixed-repetition quantifier ({*m*} or {*m*}?) has the same greediness (possibly none) as the atom itself.

- A quantified atom with other normal quantifiers (including `{m,n}` with *m* equal to *n*) is greedy (prefers longest match).
- A quantified atom with a non-greedy quantifier (including `{m,n}?` with *m* equal to *n*) is non-greedy (prefers shortest match).
- A branch — that is, an RE that has no top-level `|` operator — has the same greediness as the first quantified atom in it that has a greediness attribute.
- An RE consisting of two or more branches connected by the `|` operator is always greedy.

The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring *as a whole*. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

In the first case, the RE as a whole is greedy because `Y*` is greedy. It can match beginning at the `Y`, and it matches the longest possible string starting there, i.e., `Y123`. The output is the parenthesized part of that, or `123`. In the second case, the RE as a whole is non-greedy because `Y*?` is non-greedy. It can match beginning at the `Y`, and it matches the shortest possible string starting there, i.e., `Y1`. The subexpression `[0-9]{1,3}` is greedy but it cannot change the decision as to the overall match length; so it is forced to match just `1`.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to “eat” relative to each other.

The quantifiers `{1,1}` and `{1,1}?` can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE. This is useful when you need the whole RE to have a greediness attribute different from what's deduced from its elements. As an example, suppose that we are trying to separate a string containing some digits into the digits and the parts before and after them. We might try to do that like this:

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
Result: {abc0123,4,xyz}
```

That didn't work: the first `.*` is greedy so it “eats” as much as it can, leaving the `\d+` to match at the last possible place, the last digit. We might try to fix that by making it non-greedy:

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
Result: {abc,0,""}
```

That didn't work either, because now the RE as a whole is non-greedy and so it ends the overall match as soon as possible. We can get what we want by forcing the RE as a whole to be greedy:

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Result: {abc,01234,xyz}
```

Controlling the RE's overall greediness separately from its components' greediness allows great flexibility in handling variable-length patterns.

When deciding what is a longer or shorter match, match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: `bb*` matches the three middle characters of `abbbc`; `(week|wee)(night|knights)` matches all ten characters of `weeknights`; when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters; and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g., `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g., `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

If newline-sensitive matching is specified, `.` and bracket expressions using `^` will never match the newline character (so that matches will not cross lines unless the RE explicitly includes a newline) and `^` and `$` will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes `\A` and `\Z` continue to match beginning or end of string *only*. Also, the character class shorthands `\D` and `\W` will match a newline regardless of this mode. (Before PostgreSQL 14, they did not match newlines when in newline-sensitive mode. Write `[^[:digit:]]` or `[^[:word:]]` to get the old behavior.)

If partial newline-sensitive matching is specified, this affects `.` and bracket expressions as with newline-sensitive matching, but not `^` and `$`.

If inverse partial newline-sensitive matching is specified, this affects `^` and `$` as with newline-sensitive matching, but not `.` and bracket expressions. This isn't very useful but is provided for symmetry.

9.7.3.6. Limits and Compatibility

No particular limit is imposed on the length of REs in this implementation. However, programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that `\` does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the `***` syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include `\b`, `\B`, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead/lookbehind constraints, and the longest/shortest-match (rather than first-match) matching semantics.

9.7.3.7. Basic Regular Expressions

BREs differ from EREs in several respects. In BREs, `|`, `+`, and `?` are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are `\{` and `\}`, with `{` and `}` by themselves ordinary characters. The parentheses for nested subexpressions are `\(` and `\)`, with `(` and `)` by themselves ordinary characters. `^` is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, `$` is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and `*` is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading `^`). Finally, single-digit back references are available, and `\<` and `\>` are synonyms for `[[:<:]]` and `[[:>:]]` respectively; no other escapes are available in BREs.

9.7.3.8. Differences from SQL Standard and XQuery

Since SQL:2008, the SQL standard includes regular expression operators and functions that performs pattern matching according to the XQuery regular expression standard:

- `LIKE_REGEX`
- `OCCURRENCES_REGEX`
- `POSITION_REGEX`
- `SUBSTRING_REGEX`
- `TRANSLATE_REGEX`

PostgreSQL does not currently implement these operators and functions. You can get approximately equivalent functionality in each case as shown in Table 9.25. (Various optional clauses on both sides have been omitted in this table.)

Table 9.25. Regular Expression Functions Equivalencies

SQL standard	PostgreSQL
string `LIKE_REGEX` *pattern*	`regexp_like(`*string*`,` *pattern*`)` or *string* `~` *pattern*
`OCCURRENCES_REGEX(`*pattern* `IN` *string*`)`	`regexp_count(`*string*`,` *pattern*`)`
`POSITION_REGEX(`*pattern* `IN` *string*`)`	`regexp_instr(`*string*`,` *pattern*`)`
`SUBSTRING_REGEX(`*pattern* `IN` *string*`)`	`regexp_substr(`*string*`,` *pattern*`)`
`TRANSLATE_REGEX(`*pattern* `IN` *string* `WITH` *replacement*`)`	`regexp_replace(`*string*`,` *pattern*`,` *replacement*`)`

Regular expression functions similar to those provided by PostgreSQL are also available in a number of other SQL implementations, whereas the SQL-standard functions are not as widely implemented. Some of the details of the regular expression syntax will likely differ in each implementation.

The SQL-standard operators and functions use XQuery regular expressions, which are quite close to the ARE syntax described above. Notable differences between the existing POSIX-based regular-expression feature and XQuery regular expressions include:

- XQuery character class subtraction is not supported. An example of this feature is using the following to match only English consonants: `[a-z-[aeiou]]`.
- XQuery character class shorthands `\c`, `\C`, `\i`, and `\I` are not supported.
- XQuery character class elements using `\p{UnicodeProperty}` or the inverse `\P{UnicodeProperty}` are not supported.
- POSIX interprets character classes such as `\w` (see Table 9.21) according to the prevailing locale (which you can control by attaching a `COLLATE` clause to the operator or function). XQuery specifies these classes by reference to Unicode character properties, so equivalent behavior is obtained only with a locale that follows the Unicode rules.
- The SQL standard (not XQuery itself) attempts to cater for more variants of "newline" than POSIX does. The newline-sensitive matching options described above consider only ASCII NL (`\n`) to be a newline, but SQL would have us treat CR (`\r`), CRLF (`\r\n`) (a Windows-style newline), and some Unicode-only characters like LINE SEPARATOR (U+2028) as newlines as well. Notably, `.` and `\s` should count `\r\n` as one character not two according to SQL.
- Of the character-entry escapes described in Table 9.20, XQuery supports only `\n`, `\r`, and `\t`.

- XQuery does not support the `[:name:]` syntax for character classes within bracket expressions.
- XQuery does not have lookahead or lookbehind constraints, nor any of the constraint escapes described in Table 9.22.
- The metasyntax forms described in Section 9.7.3.4 do not exist in XQuery.
- The regular expression flag letters defined by XQuery are related to but not the same as the option letters for POSIX (Table 9.24). While the `i` and `q` options behave the same, others do not:
  - XQuery's `s` (allow dot to match newline) and `m` (allow `^` and `$` to match at newlines) flags provide access to the same behaviors as POSIX's `n`, `p` and `w` flags, but they do *not* match the behavior of POSIX's `s` and `m` flags. Note in particular that dot-matches-newline is the default behavior in POSIX but not XQuery.
  - XQuery's `x` (ignore whitespace in pattern) flag is noticeably different from POSIX's expanded-mode flag. POSIX's `x` flag also allows # to begin a comment in the pattern, and POSIX will not ignore a whitespace character after a backslash.

9.8. Data Type Formatting Functions

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. Table 9.26 lists them. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

Table 9.26. Formatting Functions

Function **Description** **Example(s)**

`to_char ( timestamp, text ) → text`
`to_char ( timestamp with time zone, text ) → text`
Converts time stamp to string according to the given format.
`to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12`

`to_char ( interval, text ) → text`
Converts interval to string according to the given format.
`to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12`

`to_char ( numeric_type, text ) → text`
Converts number to string according to the given format; available for `integer`, `bigint`, `numeric`, `real`, `double precision`.
`to_char(125, '999') → 125`
`to_char(125.8::real, '999D9') → 125.8`
`to_char(-125.8, '999D99S') → 125.80-`

`to_date ( text, text ) → date`
Converts string to date according to the given format.
`to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05`

`to_number ( text, text ) → numeric`
Converts string to numeric according to the given format.
`to_number('12,454.8-', '99G999D9S') → -12454.8`

Function Description Example(s)

`to_timestamp ( text, text ) → timestamp with time zone`
Converts string to time stamp according to the given format. (See also `to_timestamp(double precision)` in Table 9.33.)
`to_timestamp('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05 00:00:00-05`

> **Tip**
>
> `to_timestamp` and `to_date` exist to handle input formats that cannot be converted by simple casting. For most standard date/time formats, simply casting the source string to the required data type works, and is much easier. Similarly, `to_number` is unnecessary for standard numeric representations.

In a `to_char` output template string, there are certain patterns that are recognized and replaced with appropriately-formatted data based on the given value. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for the other functions), template patterns identify the values to be supplied by the input data string. If there are characters in the template string that are not template patterns, the corresponding characters in the input data string are simply skipped over (whether or not they are equal to the template string characters).

Table 9.27 shows the template patterns available for formatting date and time values.

Table 9.27. Template Patterns for Date/Time Formatting

Pattern	Description
`HH`	hour of day (01–12)
`HH12`	hour of day (01–12)
`HH24`	hour of day (00–23)
`MI`	minute (00–59)
`SS`	second (00–59)
`MS`	millisecond (000–999)
`US`	microsecond (000000–999999)
`FF1`	tenth of second (0–9)
`FF2`	hundredth of second (00–99)
`FF3`	millisecond (000–999)
`FF4`	tenth of a millisecond (0000–9999)
`FF5`	hundredth of a millisecond (00000–99999)
`FF6`	microsecond (000000–999999)
`SSSS`, `SSSSS`	seconds past midnight (0–86399)
`AM`, `am`, `PM` or `pm`	meridiem indicator (without periods)
`A.M.`, `a.m.`, `P.M.` or `p.m.`	meridiem indicator (with periods)
`Y,YYY`	year (4 or more digits) with comma
`YYYY`	year (4 or more digits)
`YYY`	last 3 digits of year
`YY`	last 2 digits of year

Pattern	Description
Y	last digit of year
IYYY	ISO 8601 week-numbering year (4 or more digits)
IYY	last 3 digits of ISO 8601 week-numbering year
IY	last 2 digits of ISO 8601 week-numbering year
I	last digit of ISO 8601 week-numbering year
BC, bc, AD or ad	era indicator (without periods)
B.C., b.c., A.D. or a.d.	era indicator (with periods)
MONTH	full upper case month name (blank-padded to 9 chars)
Month	full capitalized month name (blank-padded to 9 chars)
month	full lower case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars in English, localized lengths vary)
Mon	abbreviated capitalized month name (3 chars in English, localized lengths vary)
mon	abbreviated lower case month name (3 chars in English, localized lengths vary)
MM	month number (01–12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full capitalized day name (blank-padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars in English, localized lengths vary)
Dy	abbreviated capitalized day name (3 chars in English, localized lengths vary)
dy	abbreviated lower case day name (3 chars in English, localized lengths vary)
DDD	day of year (001–366)
IDDD	day of ISO 8601 week-numbering year (001–371; day 1 of the year is Monday of the first ISO week)
DD	day of month (01–31)
D	day of the week, Sunday (1) to Saturday (7)
ID	ISO 8601 day of the week, Monday (1) to Sunday (7)
W	week of month (1–5) (the first week starts on the first day of the month)
WW	week number of year (1–53) (the first week starts on the first day of the year)

Pattern	Description
IW	week number of ISO 8601 week-numbering year (01–53; the first Thursday of the year is in week 1)
CC	century (2 digits) (the twenty-first century starts on 2001-01-01)
J	Julian Date (integer days since November 24, 4714 BC at local midnight; see Section B.7)
Q	quarter
RM	month in upper case Roman numerals (I–XII; I=January)
rm	month in lower case Roman numerals (i–xii; i=January)
TZ	upper case time-zone abbreviation
tz	lower case time-zone abbreviation
TZH	time-zone hours
TZM	time-zone minutes
OF	time-zone offset from UTC (*HH* or *HH*:*MM*)

Modifiers can be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. Table 9.28 shows the modifier patterns for date/time formatting.

Table 9.28. Template Pattern Modifiers for Date/Time Formatting

Modifier	Description	Example
FM prefix	fill mode (suppress leading zeroes and padding blanks)	FMMonth
TH suffix	upper case ordinal number suffix	DDTH, e.g., 12TH
th suffix	lower case ordinal number suffix	DDth, e.g., 12th
FX prefix	fixed format global option (see usage notes)	FX Month DD Day
TM prefix	translation mode (use localized day and month names based on lc_time)	TMMonth
SP suffix	spell mode (not implemented)	DDSP

Usage notes for date/time formatting:

- FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width. In PostgreSQL, FM modifies only the next specification, while in Oracle FM affects all subsequent specifications, and repeated FM modifiers toggle fill mode on and off.
- TM suppresses trailing blanks whether or not FM is specified.
- to_timestamp and to_date ignore letter case in the input; so for example MON, Mon, and mon all accept the same strings. When using the TM modifier, case-folding is done according to the rules of the function's input collation (see Section 23.2).
- to_timestamp and to_date skip multiple blank spaces at the beginning of the input string and around date and time values unless the FX option is used. For example, to_timestamp(' 2000 JUN', 'YYYY MON') and to_timestamp('2000 - JUN', 'YYYY-

MON') work, but to_timestamp('2000 JUN', 'FXYYYY MON') returns an error because to_timestamp expects only a single space. FX must be specified as the first item in the template.

- A separator (a space or non-letter/non-digit character) in the template string of to_timestamp and to_date matches any single separator in the input string or is skipped, unless the FX option is used. For example, to_timestamp('2000JUN', 'YYYY///MON') and to_timestamp('2000/JUN', 'YYYY MON') work, but to_timestamp('2000//JUN', 'YYYY/MON') returns an error because the number of separators in the input string exceeds the number of separators in the template.

  If FX is specified, a separator in the template string matches exactly one character in the input string. But note that the input string character is not required to be the same as the separator from the template string. For example, to_timestamp('2000/JUN', 'FXYYYY MON') works, but to_timestamp('2000/JUN', 'FXYYYY MON') returns an error because the second space in the template string consumes the letter J from the input string.

- A TZH template pattern can match a signed number. Without the FX option, minus signs may be ambiguous, and could be interpreted as a separator. This ambiguity is resolved as follows: If the number of separators before TZH in the template string is less than the number of separators before the minus sign in the input string, the minus sign is interpreted as part of TZH. Otherwise, the minus sign is considered to be a separator between values. For example, to_timestamp('2000 -10', 'YYYY TZH') matches -10 to TZH, but to_timestamp('2000 -10', 'YYYY TZH') matches 10 to TZH.

- Ordinary text is allowed in to_char templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains template patterns. For example, in '"Hello Year "YYYY', the YYYY will be replaced by the year data, but the single Y in Year will not be. In to_date, to_number, and to_timestamp, literal text and double-quoted strings result in skipping the number of characters contained in the string; for example "XX" skips two input characters (whether or not they are XX).

  > **Tip**
  >
  > Prior to PostgreSQL 12, it was possible to skip arbitrary text in the input string using non-letter or non-digit characters. For example, to_timestamp('2000y6m1d', 'yyyy-MM-DD') used to work. Now you can only use letter characters for this purpose. For example, to_timestamp('2000y6m1d', 'yyyytMMtDDt') and to_timestamp('2000y6m1d', 'yyyy"y"MM"m"DD"d"') skip y, m, and d.

- If you want to have a double quote in the output you must precede it with a backslash, for example '\"YYYY Month\"'. Backslashes are not otherwise special outside of double-quoted strings. Within a double-quoted string, a backslash causes the next character to be taken literally, whatever it is (but this has no special effect unless the next character is a double quote or another backslash).

- In to_timestamp and to_date, if the year format specification is less than four digits, e.g., YYY, and the supplied year is less than four digits, the year will be adjusted to be nearest to the year 2020, e.g., 95 becomes 1995.

- In to_timestamp and to_date, negative years are treated as signifying BC. If you write both a negative year and an explicit BC field, you get AD again. An input of year zero is treated as 1 BC.

- In to_timestamp and to_date, the YYYY conversion has a restriction when processing years with more than 4 digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as 4 digits. For example (with the year 20000): to_date('200001130', 'YYYYMMDD') will be interpreted as a 4-digit year; instead use a non-digit separator after the year, like to_date('20000-1130', 'YYYY-MMDD') or to_date('20000Nov30', 'YYYYMonDD').

- In `to_timestamp` and `to_date`, the `CC` (century) field is accepted but ignored if there is a `YYY`, `YYYY` or `Y,YYY` field. If `CC` is used with `YY` or `Y` then the result is computed as that year in the specified century. If the century is specified but the year is not, the first year of the century is assumed.

- In `to_timestamp` and `to_date`, weekday names or numbers (`DAY`, `D`, and related field types) are accepted but are ignored for purposes of computing the result. The same is true for quarter (`Q`) fields.

- In `to_timestamp` and `to_date`, an ISO 8601 week-numbering date (as distinct from a Gregorian date) can be specified in one of two ways:

  - Year, week number, and weekday: for example `to_date('2006-42-4', 'IYYY-IW-ID')` returns the date `2006-10-19`. If you omit the weekday it is assumed to be 1 (Monday).

  - Year and day of year: for example `to_date('2006-291', 'IYYY-IDDD')` also returns `2006-10-19`.

  Attempting to enter a date using a mixture of ISO 8601 week-numbering fields and Gregorian date fields is nonsensical, and will cause an error. In the context of an ISO 8601 week-numbering year, the concept of a “month” or “day of month” has no meaning. In the context of a Gregorian year, the ISO week has no meaning.

  > **Caution**
  >
  > While `to_date` will reject a mixture of Gregorian and ISO week-numbering date fields, `to_char` will not, since output format specifications like `YYYY-MM-DD (IYYY-IDDD)` can be useful. But avoid writing something like `IYYY-MM-DD`; that would yield surprising results near the start of the year. (See Section 9.9.1 for more information.)

- In `to_timestamp`, millisecond (`MS`) or microsecond (`US`) fields are used as the seconds digits after the decimal point. For example `to_timestamp('12.3', 'SS.MS')` is not 3 milliseconds, but 300, because the conversion treats it as 12 + 0.3 seconds. So, for the format `SS.MS`, the input values `12.3`, `12.30`, and `12.300` specify the same number of milliseconds. To get three milliseconds, one must write `12.003`, which the conversion treats as 12 + 0.003 = 12.003 seconds.

  Here is a more complex example: `to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')` is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.

- `to_char(..., 'ID')`'s day of the week numbering matches the `extract(isodow from ...)` function, but `to_char(..., 'D')`'s does not match `extract(dow from ...)`'s day numbering.

- `to_char(interval)` formats `HH` and `HH12` as shown on a 12-hour clock, for example zero hours and 36 hours both output as `12`, while `HH24` outputs the full hour value, which can exceed 23 in an `interval` value.

Table 9.29 shows the template patterns available for formatting numeric values.

Table 9.29. Template Patterns for Numeric Formatting

Pattern	Description
`9`	digit position (can be dropped if insignificant)
`0`	digit position (will not be dropped, even if insignificant)
`.` (period)	decimal point

Pattern	Description
, (comma)	group (thousands) separator
PR	negative value in angle brackets
S	sign anchored to number (uses locale)
L	currency symbol (uses locale)
D	decimal point (uses locale)
G	group separator (uses locale)
MI	minus sign in specified position (if number < 0)
PL	plus sign in specified position (if number > 0)
SG	plus/minus sign in specified position
RN	Roman numeral (input between 1 and 3999)
TH or th	ordinal number suffix
V	shift specified number of digits (see notes)
EEEE	exponent for scientific notation

Usage notes for numeric formatting:

- `0` specifies a digit position that will always be printed, even if it contains a leading/trailing zero. `9` also specifies a digit position, but if it is a leading zero then it will be replaced by a space, while if it is a trailing zero and fill mode is specified then it will be deleted. (For `to_number()`, these two pattern characters are equivalent.)

- If the format provides fewer fractional digits than the number being formatted, `to_char()` will round the number to the specified number of fractional digits.

- The pattern characters `S`, `L`, `D`, and `G` represent the sign, currency symbol, decimal point, and thousands separator characters defined by the current locale (see lc_monetary and lc_numeric). The pattern characters period and comma represent those exact characters, with the meanings of decimal point and thousands separator, regardless of locale.

- If no explicit provision is made for a sign in `to_char()`'s pattern, one column will be reserved for the sign, and it will be anchored to (appear just left of) the number. If `S` appears just left of some `9`'s, it will likewise be anchored to the number.

- A sign formatted using `SG`, `PL`, or `MI` is not anchored to the number; for example, `to_char(-12, 'MI9999')` produces `'-  12'` but `to_char(-12, 'S9999')` produces `'  -12'`. (The Oracle implementation does not allow the use of `MI` before `9`, but rather requires that `9` precede `MI`.)

- `TH` does not convert values less than zero and does not convert fractional numbers.

- `PL`, `SG`, and `TH` are PostgreSQL extensions.

- In `to_number`, if non-data template patterns such as `L` or `TH` are used, the corresponding number of input characters are skipped, whether or not they match the template pattern, unless they are data characters (that is, digits, sign, decimal point, or comma). For example, `TH` would skip two non-data characters.

- `V` with `to_char` multiplies the input values by `10^`*n*, where *n* is the number of digits following `V`. `V` with `to_number` divides in a similar manner. `to_char` and `to_number` do not support the use of `V` combined with a decimal point (e.g., `99.9V99` is not allowed).

- `EEEE` (scientific notation) cannot be used in combination with any of the other formatting patterns or modifiers other than digit and decimal point patterns, and must be at the end of the format string (e.g., `9.99EEEE` is a valid pattern).

Certain modifiers can be applied to any template pattern to alter its behavior. For example, `FM99.99` is the `99.99` pattern with the `FM` modifier. Table 9.30 shows the modifier patterns for numeric formatting.

Table 9.30. Template Pattern Modifiers for Numeric Formatting

Modifier	Description	Example
`FM` prefix	fill mode (suppress trailing zeroes and padding blanks)	`FM99.99`
`TH` suffix	upper case ordinal number suffix	`999TH`
`th` suffix	lower case ordinal number suffix	`999th`

Table 9.31 shows some examples of the use of the `to_char` function.

Table 9.31. `to_char` Examples

Expression	Result
`to_char(current_timestamp, 'Day, DD  HH12:MI:SS')`	`'Tuesday  , 06  05:39:18'`
`to_char(current_timestamp, 'FMDay, FMDD  HH12:MI:SS')`	`'Tuesday, 6  05:39:18'`
`to_char(current_timestamp AT TIME ZONE 'UTC', 'YYYY-MM-DD"T"HH24:MI:SS"Z"')`	`'2022-12-06T05:39:18Z'`, ISO 8601 extended format
`to_char(-0.1, '99.99')`	`'  -.10'`
`to_char(-0.1, 'FM9.99')`	`'-.1'`
`to_char(-0.1, 'FM90.99')`	`'-0.1'`
`to_char(0.1, '0.9')`	`' 0.1'`
`to_char(12, '9990999.9')`	`'    0012.0'`
`to_char(12, 'FM9990999.9')`	`'0012.'`
`to_char(485, '999')`	`' 485'`
`to_char(-485, '999')`	`'-485'`
`to_char(485, '9 9 9')`	`' 4 8 5'`
`to_char(1485, '9,999')`	`' 1,485'`
`to_char(1485, '9G999')`	`' 1 485'`
`to_char(148.5, '999.999')`	`' 148.500'`
`to_char(148.5, 'FM999.999')`	`'148.5'`
`to_char(148.5, 'FM999.990')`	`'148.500'`
`to_char(148.5, '999D999')`	`' 148,500'`
`to_char(3148.5, '9G999D999')`	`' 3 148,500'`
`to_char(-485, '999S')`	`'485-'`
`to_char(-485, '999MI')`	`'485-'`
`to_char(485, '999MI')`	`'485 '`
`to_char(485, 'FM999MI')`	`'485'`
`to_char(485, 'PL999')`	`'+485'`
`to_char(485, 'SG999')`	`'+485'`

Expression	Result
`to_char(-485, 'SG999')`	`'-485'`
`to_char(-485, '9SG99')`	`'4-85'`
`to_char(-485, '999PR')`	`'<485>'`
`to_char(485, 'L999')`	`'DM 485'`
`to_char(485, 'RN')`	`'        CDLXXXV'`
`to_char(485, 'FMRN')`	`'CDLXXXV'`
`to_char(5.2, 'FMRN')`	`'V'`
`to_char(482, '999th')`	`' 482nd'`
`to_char(485, '"Good number:"999')`	`'Good number: 485'`
`to_char(485.8, '"Pre:"999" Post:" .999')`	`'Pre: 485 Post: .800'`
`to_char(12, '99V999')`	`' 12000'`
`to_char(12.4, '99V999')`	`' 12400'`
`to_char(12.45, '99V9')`	`' 125'`
`to_char(0.0004859, '9.99EEEE')`	`' 4.86e-04'`

9.9. Date/Time Functions and Operators

Table 9.33 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9.32 illustrates the behaviors of the basic arithmetic operators (`+`, `*`, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

In addition, the usual comparison operators shown in Table 9.1 are available for the date/time types. Dates and timestamps (with or without time zone) are all comparable, while times (with or without time zone) and intervals can only be compared to other values of the same data type. When comparing a timestamp without time zone to a timestamp with time zone, the former value is assumed to be given in the time zone specified by the TimeZone configuration parameter, and is rotated to UTC for comparison to the latter value (which is already in UTC internally). Similarly, a date value is assumed to represent midnight in the `TimeZone` zone when comparing it to a timestamp.

All the functions and operators described below that take `time` or `timestamp` inputs actually come in two variants: one that takes `time with time zone` or `timestamp with time zone`, and one that takes `time without time zone` or `timestamp without time zone`. For brevity, these variants are not shown separately. Also, the `+` and `*` operators come in commutative pairs (for example both `date + integer` and `integer + date`); we show only one of each such pair.

Table 9.32. Date/Time Operators

Operator **Description** **Example(s)**

`date + integer → date`
Add a number of days to a date
`date '2001-09-28' + 7 → 2001-10-05`

`date + interval → timestamp`
Add an interval to a date
`date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00`

Operator **Description** **Example(s)**

`date + time → timestamp`
Add a time-of-day to a date
`date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00`

`interval + interval → interval`
Add intervals
`interval '1 day' + interval '1 hour' → 1 day 01:00:00`

`timestamp + interval → timestamp`
Add an interval to a timestamp
`timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00`

`time + interval → time`
Add an interval to a time
`time '01:00' + interval '3 hours' → 04:00:00`

`- interval → interval`
Negate an interval
`- interval '23 hours' → -23:00:00`

`date - date → integer`
Subtract dates, producing the number of days elapsed
`date '2001-10-01' - date '2001-09-28' → 3`

`date - integer → date`
Subtract a number of days from a date
`date '2001-10-01' - 7 → 2001-09-24`

`date - interval → timestamp`
Subtract an interval from a date
`date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00`

`time - time → interval`
Subtract times
`time '05:00' - time '03:00' → 02:00:00`

`time - interval → time`
Subtract an interval from a time
`time '05:00' - interval '2 hours' → 03:00:00`

`timestamp - interval → timestamp`
Subtract an interval from a timestamp
`timestamp '2001-09-28 23:00' - interval '23 hours' → 2001-09-28 00:00:00`

`interval - interval → interval`
Subtract intervals
`interval '1 day' - interval '1 hour' → 1 day -01:00:00`

`timestamp - timestamp → interval` Subtract timestamps (converting 24-hour intervals into days, similarly to `justify_hours()`)

Operator **Description** **Example(s)**
`timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00'` → `63 days 15:00:00`

`interval * double precision → interval`
Multiply an interval by a scalar
`interval '1 second' * 900 → 00:15:00`
`interval '1 day' * 21 → 21 days`
`interval '1 hour' * 3.5 → 03:30:00`

`interval / double precision → interval`
Divide an interval by a scalar
`interval '1 hour' / 1.5 → 00:40:00`

Table 9.33. Date/Time Functions

Function **Description** **Example(s)**

`age ( timestamp, timestamp ) → interval`
Subtract arguments, producing a "symbolic" result that uses years and months, rather than just days
`age(timestamp '2001-04-10', timestamp '1957-06-13') → 43 years 9 mons 27 days`

`age ( timestamp ) → interval`
Subtract argument from `current_date` (at midnight)
`age(timestamp '1957-06-13') → 62 years 6 mons 10 days`

`clock_timestamp ( ) → timestamp with time zone`
Current date and time (changes during statement execution); see Section 9.9.5
`clock_timestamp() → 2019-12-23 14:39:53.662522-05`

`current_date → date`
Current date; see Section 9.9.5
`current_date → 2019-12-23`

`current_time → time with time zone`
Current time of day; see Section 9.9.5
`current_time → 14:39:53.662522-05`

`current_time ( integer ) → time with time zone`
Current time of day, with limited precision; see Section 9.9.5
`current_time(2) → 14:39:53.66-05`

`current_timestamp → timestamp with time zone`
Current date and time (start of current transaction); see Section 9.9.5
`current_timestamp → 2019-12-23 14:39:53.662522-05`

`current_timestamp ( integer ) → timestamp with time zone`
Current date and time (start of current transaction), with limited precision; see Section 9.9.5
`current_timestamp(0) → 2019-12-23 14:39:53-05`

Function **Description** **Example(s)**

`date_add` (`timestamp with time zone`, `interval` [, `text`]) → `timestamp with time zone`
Add an `interval` to a `timestamp with time zone`, computing times of day and daylight-savings adjustments according to the time zone named by the third argument, or the current TimeZone setting if that is omitted. The form with two arguments is equivalent to the `timestamp with time zone + interval` operator.
`date_add('2021-10-31 00:00:00+02'::timestamptz, '1 day'::interval, 'Europe/Warsaw')` → `2021-10-31 23:00:00+00`

`date_bin` (`interval`, `timestamp`, `timestamp`) → `timestamp`
Bin input into specified interval aligned with specified origin; see Section 9.9.3
`date_bin('15 minutes', timestamp '2001-02-16 20:38:40', timestamp '2001-02-16 20:05:00')` → `2001-02-16 20:35:00`

`date_part` (`text`, `timestamp`) → `double precision`
Get timestamp subfield (equivalent to `extract`); see Section 9.9.1
`date_part('hour', timestamp '2001-02-16 20:38:40')` → `20`

`date_part` (`text`, `interval`) → `double precision`
Get interval subfield (equivalent to `extract`); see Section 9.9.1
`date_part('month', interval '2 years 3 months')` → `3`

`date_subtract` (`timestamp with time zone`, `interval` [, `text`]) → `timestamp with time zone`
Subtract an `interval` from a `timestamp with time zone`, computing times of day and daylight-savings adjustments according to the time zone named by the third argument, or the current TimeZone setting if that is omitted. The form with two arguments is equivalent to the `timestamp with time zone - interval` operator.
`date_subtract('2021-11-01 00:00:00+01'::timestamptz, '1 day'::interval, 'Europe/Warsaw')` → `2021-10-30 22:00:00+00`

`date_trunc` (`text`, `timestamp`) → `timestamp`
Truncate to specified precision; see Section 9.9.2
`date_trunc('hour', timestamp '2001-02-16 20:38:40')` → `2001-02-16 20:00:00`

`date_trunc` (`text`, `timestamp with time zone`, `text`) → `timestamp with time zone`
Truncate to specified precision in the specified time zone; see Section 9.9.2
`date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')` → `2001-02-16 13:00:00+00`

`date_trunc` (`text`, `interval`) → `interval`
Truncate to specified precision; see Section 9.9.2
`date_trunc('hour', interval '2 days 3 hours 40 minutes')` → `2 days 03:00:00`

`extract` (*`field`* `from timestamp`) → `numeric`
Get timestamp subfield; see Section 9.9.1
`extract(hour from timestamp '2001-02-16 20:38:40')` → `20`

`extract` (*`field`* `from interval`) → `numeric` Get interval subfield; see Section 9.9.1

Function **Description** **Example(s)**
`extract(month from interval '2 years 3 months')` → `3`

`isfinite ( date )` → `boolean`
Test for finite date (not +/-infinity)
`isfinite(date '2001-02-16')` → `true`

`isfinite ( timestamp )` → `boolean`
Test for finite timestamp (not +/-infinity)
`isfinite(timestamp 'infinity')` → `false`

`isfinite ( interval )` → `boolean`
Test for finite interval (not +/-infinity)
`isfinite(interval '4 hours')` → `true`

`justify_days ( interval )` → `interval`
Adjust interval, converting 30-day time periods to months
`justify_days(interval '1 year 65 days')` → `1 year 2 mons 5 days`

`justify_hours ( interval )` → `interval`
Adjust interval, converting 24-hour time periods to days
`justify_hours(interval '50 hours 10 minutes')` → `2 days 02:10:00`

`justify_interval ( interval )` → `interval`
Adjust interval using `justify_days` and `justify_hours`, with additional sign adjustments
`justify_interval(interval '1 mon -1 hour')` → `29 days 23:00:00`

`localtime` → `time`
Current time of day; see Section 9.9.5
`localtime` → `14:39:53.662522`

`localtime ( integer )` → `time`
Current time of day, with limited precision; see Section 9.9.5
`localtime(0)` → `14:39:53`

`localtimestamp` → `timestamp`
Current date and time (start of current transaction); see Section 9.9.5
`localtimestamp` → `2019-12-23 14:39:53.662522`

`localtimestamp ( integer )` → `timestamp`
Current date and time (start of current transaction), with limited precision; see Section 9.9.5
`localtimestamp(2)` → `2019-12-23 14:39:53.66`

`make_date ( year int, month int, day int )` → `date`
Create date from year, month and day fields (negative years signify BC)
`make_date(2013, 7, 15)` → `2013-07-15`

`make_interval ( [ years int [, months int [, weeks int [, days int [, hours int [, mins int [, secs double precision ]]]]]] )` → `interval`

Function **Description** **Example(s)**
Create interval from years, months, weeks, days, hours, minutes and seconds fields, each of which can default to zero `make_interval(days => 10)` → `10 days`

`make_time ( hour int, min int, sec double precision ) → time`
Create time from hour, minute and seconds fields
`make_time(8, 15, 23.5)` → `08:15:23.5`

`make_timestamp ( year int, month int, day int, hour int, min int, sec double precision ) → timestamp`
Create timestamp from year, month, day, hour, minute and seconds fields (negative years signify BC)
`make_timestamp(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5`

`make_timestamptz ( year int, month int, day int, hour int, min int, sec double precision [, timezone text ] ) → timestamp with time zone`
Create timestamp with time zone from year, month, day, hour, minute and seconds fields (negative years signify BC). If *timezone* is not specified, the current time zone is used; the examples assume the session time zone is `Europe/London`
`make_timestamptz(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5+01`
`make_timestamptz(2013, 7, 15, 8, 15, 23.5, 'America/New_York')` → `2013-07-15 13:15:23.5+01`

`now ( ) → timestamp with time zone`
Current date and time (start of current transaction); see Section 9.9.5
`now()` → `2019-12-23 14:39:53.662522-05`

`statement_timestamp ( ) → timestamp with time zone`
Current date and time (start of current statement); see Section 9.9.5
`statement_timestamp()` → `2019-12-23 14:39:53.662522-05`

`timeofday ( ) → text`
Current date and time (like `clock_timestamp`, but as a `text` string); see Section 9.9.5
`timeofday()` → `Mon Dec 23 14:39:53.662522 2019 EST`

`transaction_timestamp ( ) → timestamp with time zone`
Current date and time (start of current transaction); see Section 9.9.5
`transaction_timestamp()` → `2019-12-23 14:39:53.662522-05`

`to_timestamp ( double precision ) → timestamp with time zone`
Convert Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp with time zone
`to_timestamp(1284352323)` → `2010-09-13 04:32:03+00`

In addition to these functions, the SQL `OVERLAPS` operator is supported:

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval. When a pair of values is provided, either the start or the end can be written first; OVERLAPS automatically takes the earlier value of the pair as the start. Each time period is considered to represent the half-open interval *start* <= *time* < *end*, unless *start* and *end* are equal in which case it represents that single time instant. This means for instance that two time periods with only an endpoint in common do not overlap.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

When adding an interval value to (or subtracting an interval value from) a timestamp or timestamp with time zone value, the months, days, and microseconds fields of the interval value are handled in turn. First, a nonzero months field advances or decrements the date of the timestamp by the indicated number of months, keeping the day of month the same unless it would be past the end of the new month, in which case the last day of that month is used. (For example, March 31 plus 1 month becomes April 30, but March 31 plus 2 months becomes May 31.) Then the days field advances or decrements the date of the timestamp by the indicated number of days. In both these steps the local time of day is kept the same. Finally, if there is a nonzero microseconds field, it is added or subtracted literally. When doing arithmetic on a timestamp with time zone value in a time zone that recognizes DST, this means that adding or subtracting (say) interval '1 day' does not necessarily have the same result as adding or subtracting interval '24 hours'. For example, with the session time zone set to America/Denver:

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval
 '1 day';
Result: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval
 '24 hours';
Result: 2005-04-03 13:00:00-06
```

This happens because an hour was skipped due to a change in daylight saving time at 2005-04-03 02:00:00 in time zone America/Denver.

Note there can be ambiguity in the months field returned by age because different months have different numbers of days. PostgreSQL's approach uses the month from the earlier of the two dates when calculating partial months. For example, age('2004-06-01', '2004-04-30') uses April to yield 1 mon 1 day, while using May would yield 1 mon 2 days because May has 31 days, while April has only 30.

Subtraction of dates and timestamps can also be complex. One conceptually simple way to perform subtraction is to convert each value to a number of seconds using EXTRACT(EPOCH FROM ...), then subtract the results; this produces the number of *seconds* between the two values. This will adjust for the number of days in each month, timezone changes, and daylight saving time adjustments. Subtraction of date or timestamp values with the "-" operator returns the number of days (24-hours) and hours/minutes/seconds between the values, making the same adjustments. The age function returns years, months, days, and hours/minutes/seconds, performing field-by-field subtraction and then ad-

justing for negative field values. The following queries illustrate the differences in these approaches. The sample results were produced with `timezone = 'US/Eastern'`; there is a daylight saving time change between the two dates used:

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Result: 10537200.000000
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
Result: 121.9583333333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01
 12:00:00';
Result: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz
 '2013-03-01 12:00:00');
Result: 4 mons
```

9.9.1. EXTRACT, `date_part`

```
EXTRACT(field FROM source)
```

The `extract` function retrieves subfields such as year or hour from date/time values. *`source`* must be a value expression of type `timestamp`, `date`, `time`, or `interval`. (Timestamps and times can be with or without time zone.) *`field`* is an identifier or string that selects what field to extract from the source value. Not all fields are valid for every input data type; for example, fields smaller than a day cannot be extracted from a `date`, while fields of a day or more cannot be extracted from a `time`. The `extract` function returns values of type `numeric`.

The following are valid field names:

`century`

The century; for `interval` values, the year field divided by 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21
SELECT EXTRACT(CENTURY FROM DATE '0001-01-01 AD');
Result: 1
SELECT EXTRACT(CENTURY FROM DATE '0001-12-31 BC');
Result: -1
SELECT EXTRACT(CENTURY FROM INTERVAL '2001 years');
Result: 20
```

`day`

The day of the month (1–31); for `interval` values, the number of days

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Result: 40
```

`decade`

The year field divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

`dow`

The day of the week as Sunday (`0`) to Saturday (`6`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

Note that `extract`'s day of the week numbering differs from that of the `to_char(..., 'D')` function.

`doy`

The day of the year (1–365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

`epoch`

For `timestamp with time zone` values, the number of seconds since 1970-01-01 00:00:00 UTC (negative for timestamps before that); for `date` and `timestamp` values, the nominal number of seconds since 1970-01-01 00:00:00, without regard to timezone or daylight-savings rules; for `interval` values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40.12-08');
Result: 982384720.120000
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
Result: 982355920.120000
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800.000000
```

You can convert an epoch value back to a `timestamp with time zone` with `to_timestamp`:

```
SELECT to_timestamp(982384720.12);
Result: 2001-02-17 04:38:40.12+00
```

Beware that applying `to_timestamp` to an epoch extracted from a `date` or `timestamp` value could produce a misleading result: the result will effectively assume that the original value had been given in UTC, which might not be the case.

`hour`

The hour field (0–23 in timestamps, unrestricted in intervals)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

`isodow`

The day of the week as Monday (`1`) to Sunday (`7`)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Result: 7
```

This is identical to `dow` except for Sunday. This matches the ISO 8601 day of the week numbering.

`isoyear`

The ISO 8601 week-numbering year that the date falls in

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Result: 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Result: 2006
```

Each ISO 8601 week-numbering year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. See the `week` field for more information.

`julian`

The *Julian Date* corresponding to the date or timestamp. Timestamps that are not local midnight result in a fractional value. See Section B.7 for more information.

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
Result: 2453737
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
Result: 2453737.50000000000000000000
```

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000; note that this includes full seconds

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The millennium; for `interval` values, the year field divided by 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
SELECT EXTRACT(MILLENNIUM FROM INTERVAL '2001 years');
Result: 2
```

Years in the 1900s are in the second millennium. The third millennium started January 1, 2001.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500.000
```

`minute`

The minutes field (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

The number of the month within the year (1–12); for `interval` values, the number of months modulo 12 (0–11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

`quarter`

The quarter of the year (1–4) that the date is in

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

`second`

The seconds field, including any fractional seconds

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40.000000
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.500000
```

`timezone`

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC. (Technically, PostgreSQL does not use UTC because leap seconds are not handled.)

`timezone_hour`

The hour component of the time zone offset

`timezone_minute`

The minute component of the time zone offset

`week`

The number of the ISO 8601 week-numbering week of the year. By definition, ISO weeks start on Mondays and the first week of a year contains January 4 of that year. In other words, the first Thursday of a year is in week 1 of that year.

In the ISO week-numbering system, it is possible for early-January dates to be part of the 52nd or 53rd week of the previous year, and for late-December dates to be part of the first week of the next year. For example, `2005-01-01` is part of the 53rd week of year 2004, and `2006-01-01` is part of the 52nd week of year 2005, while `2012-12-31` is part of the first week of 2013. It's recommended to use the `isoyear` field together with `week` to get consistent results.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

`year`

The year field. Keep in mind there is no `0 AD`, so subtracting `BC` years from `AD` years should be done with care.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

When processing an `interval` value, the `extract` function produces field values that match the interpretation used by the interval output function. This can produce surprising results if one starts with a non-normalized interval representation, for example:

```
SELECT INTERVAL '80 minutes';
Result: 01:20:00
SELECT EXTRACT(MINUTES FROM INTERVAL '80 minutes');
Result: 20
```

> **Note**
>
> When the input value is +/-Infinity, `extract` returns +/-Infinity for monotonically-increasing fields (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century`, and `millennium` for `timestamp` inputs; `epoch`, `hour`, `day`, `year`, `decade`, `century`, and `millennium` for `interval` inputs). For other fields, NULL is returned. PostgreSQL versions before 9.6 returned zero for all cases of infinite input.

The `extract` function is primarily intended for computational processing. For formatting date/time values for display, see Section 9.8.

The `date_part` function is modeled on the traditional Ingres equivalent to the SQL-standard function `extract`:

```
date_part('field', source)
```

Note that here the *`field`* parameter needs to be a string value, not a name. The valid field names for `date_part` are the same as for `extract`. For historical reasons, the `date_part` function returns values of type `double precision`. This can result in a loss of precision in certain uses. Using `extract` is recommended instead.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

9.9.2. `date_trunc`

The function `date_trunc` is conceptually similar to the `trunc` function for numbers.

```
date_trunc(field, source [, time_zone ])
```

`source` is a value expression of type `timestamp`, `timestamp with time zone`, or `interval`. (Values of type `date` and `time` are cast automatically to `timestamp` or `interval`, respectively.) *`field`* selects to which precision to truncate the input value. The return value is likewise of type `timestamp`, `timestamp with time zone`, or `interval`, and it has all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for *`field`* are:

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

When the input value is of type `timestamp with time zone`, the truncation is performed with respect to a particular time zone; for example, truncation to `day` produces a value that is midnight in that zone. By default, truncation is done with respect to the current TimeZone setting, but the optional *`time_zone`* argument can be provided to specify a different time zone. The time zone name can be specified in any of the ways described in Section 8.5.3.

A time zone cannot be specified when processing `timestamp without time zone` or `interval` inputs. These are always taken at face value.

Examples (assuming the local time zone is `America/New_York`):

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40+00');
Result: 2001-02-16 00:00:00-05
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40+00', 'Australia/Sydney');
Result: 2001-02-16 08:00:00-05
SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Result: 3 days 02:00:00
```

9.9.3. `date_bin`

The function `date_bin` “bins” the input timestamp into the specified interval (the *stride*) aligned with a specified origin.

```
date_bin(stride, source, origin)
```

source is a value expression of type `timestamp` or `timestamp with time zone`. (Values of type `date` are cast automatically to `timestamp`.) *stride* is a value expression of type `interval`. The return value is likewise of type `timestamp` or `timestamp with time zone`, and it marks the beginning of the bin into which the *source* is placed.

Examples:

```
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17',
 TIMESTAMP '2001-01-01');
Result: 2020-02-11 15:30:00
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17',
 TIMESTAMP '2001-01-01 00:02:30');
Result: 2020-02-11 15:32:30
```

In the case of full units (1 minute, 1 hour, etc.), it gives the same result as the analogous `date_trunc` call, but the difference is that `date_bin` can truncate to an arbitrary interval.

The *stride* interval must be greater than zero and cannot contain units of month or larger.

9.9.4. AT TIME ZONE and AT LOCAL

The `AT TIME ZONE` operator converts time stamp *without* time zone to/from time stamp *with* time zone, and `time with time zone` values to different time zones. Table 9.34 shows its variants.

Table 9.34. `AT TIME ZONE` and `AT LOCAL` Variants

Operator Description Example(s)

`timestamp without time zone` `AT TIME ZONE` *zone* → `timestamp with time zone`
Converts given time stamp *without* time zone to time stamp *with* time zone, assuming the given value is in the named time zone.
`timestamp '2001-02-16 20:38:40' at time zone 'America/Denver'` → `2001-02-17 03:38:40+00`

`timestamp without time zone` `AT LOCAL` → `timestamp with time zone`
Converts given time stamp *without* time zone to time stamp *with* the session's `TimeZone` value as time zone.
`timestamp '2001-02-16 20:38:40' at local` → `2001-02-17 03:38:40+00`

`timestamp with time zone` `AT TIME ZONE` *zone* → `timestamp without time zone`
Converts given time stamp *with* time zone to time stamp *without* time zone, as the time would appear in that zone.
`timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver'` → `2001-02-16 18:38:40`

`timestamp with time zone` `AT LOCAL` → `timestamp without time zone`
Converts given time stamp *with* time zone to time stamp *without* time zone, as the time would appear with the session's `TimeZone` value as time zone.
`timestamp with time zone '2001-02-16 20:38:40-05' at local` → `2001-02-16 18:38:40`

Operator **Description** **Example(s)**

`time with time zone AT TIME ZONE zone → time with time zone`
Converts given time *with* time zone to a new time zone. Since no date is supplied, this uses the currently active UTC offset for the named destination zone.
`time with time zone '05:34:17-05' at time zone 'UTC' → 10:34:17+00`

`time with time zone AT LOCAL → time with time zone`
Converts given time *with* time zone to a new time zone. Since no date is supplied, this uses the currently active UTC offset for the session's `TimeZone` value.
Assuming the session's `TimeZone` is set to `UTC`:
`time with time zone '05:34:17-05' at local → 10:34:17+00`

In these expressions, the desired time zone *zone* can be specified either as a text value (e.g., `'America/Los_Angeles'`) or as an interval (e.g., `INTERVAL '-08:00'`). In the text case, a time zone name can be specified in any of the ways described in Section 8.5.3. The interval case is only useful for zones that have fixed offsets from UTC, so it is not very common in practice.

The syntax `AT LOCAL` may be used as shorthand for `AT TIME ZONE` *local*, where *local* is the session's `TimeZone` value.

Examples (assuming the current TimeZone setting is `America/Los_Angeles`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/
Denver';
Result: 2001-02-16 19:38:40-08
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME
 ZONE 'America/Denver';
Result: 2001-02-16 18:38:40
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT
 TIME ZONE 'America/Chicago';
Result: 2001-02-16 05:38:40
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT LOCAL;
Result: 2001-02-16 17:38:40
SELECT TIME WITH TIME ZONE '20:38:40-05' AT LOCAL;
Result: 17:38:40
```

The first example adds a time zone to a value that lacks it, and displays the value using the current `TimeZone` setting. The second example shifts the time stamp with time zone value to the specified time zone, and returns the value without a time zone. This allows storage and display of values different from the current `TimeZone` setting. The third example converts Tokyo time to Chicago time. The fourth example shifts the time stamp with time zone value to the time zone currently specified by the `TimeZone` setting and returns the value without a time zone.

The fifth example is a cautionary tale. Due to the fact that there is no date associated with the input value, the conversion is made using the current date of the session. Therefore, this static example may show a wrong result depending on the time of the year it is viewed because `'America/Los_Angeles'` observes Daylight Savings Time.

The function `timezone(`*zone*`, `*timestamp*`)` is equivalent to the SQL-conforming construct *timestamp* `AT TIME ZONE` *zone*.

The function `timezone(`*zone*`, `*time*`)` is equivalent to the SQL-conforming construct *time* `AT TIME ZONE` *zone*.

The function `timezone(timestamp)` is equivalent to the SQL-conforming construct `timestamp AT LOCAL`.

The function `timezone(time)` is equivalent to the SQL-conforming construct `time AT LOCAL`.

9.9.5. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` deliver values with time zone; `LOCALTIME` and `LOCALTIMESTAMP` deliver values without time zone.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, and `LOCALTIMESTAMP` can optionally take a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:

```
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05
SELECT CURRENT_DATE;
Result: 2019-12-23
SELECT CURRENT_TIMESTAMP;
Result: 2019-12-23 14:39:53.662522-05
SELECT CURRENT_TIMESTAMP(2);
Result: 2019-12-23 14:39:53.66-05
SELECT LOCALTIMESTAMP;
Result: 2019-12-23 14:39:53.662522
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp.

> **Note**
>
> Other database systems might advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

```
transaction_timestamp()
```

```
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` is equivalent to `CURRENT_TIMESTAMP`, but is named to clearly reflect what it returns. `statement_timestamp()` returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). `statement_timestamp()` and `transaction_timestamp()` return the same value during the first command of a transaction, but might differ during subsequent commands. `clock_timestamp()` returns the actual current time, and therefore its value changes even within a single SQL command. `timeofday()` is a historical PostgreSQL function. Like `clock_timestamp()`, it returns the actual current time, but as a formatted `text` string rather than a `timestamp with time zone` value. `now()` is a traditional PostgreSQL equivalent to `transaction_timestamp()`.

All the date/time data types also accept the special literal value `now` to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- but see tip below
```

> **Tip**
>
> Do not use the third form when specifying a value to be evaluated later, for example in a `DEFAULT` clause for a table column. The system will convert `now` to a `timestamp` as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion. (See also Section 8.5.1.4.)

9.9.6. Delaying Execution

The following functions are available to delay execution of the server process:

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

`pg_sleep` makes the current session's process sleep until the given number of seconds have elapsed. Fractional-second delays can be specified. `pg_sleep_for` is a convenience function to allow the sleep time to be specified as an `interval`. `pg_sleep_until` is a convenience function for when a specific wake-up time is desired. For example:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

> **Note**
>
> The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It might be longer depending on

factors such as server load. In particular, `pg_sleep_until` is not guaranteed to wake up exactly at the specified time, but it will not wake up any earlier.

Warning

Make sure that your session does not hold more locks than necessary when calling `pg_sleep` or its variants. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

9.10. Enum Support Functions

For enum types (described in Section 8.7), there are several functions that allow cleaner programming without hard-coding particular values of an enum type. These are listed in Table 9.35. The examples assume an enum type created as:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green',
 'blue', 'purple');
```

Table 9.35. Enum Support Functions

Function Description Example(s)

`enum_first` (`anyenum`) → `anyenum`
Returns the first value of the input enum type.
`enum_first(null::rainbow)` → `red`

`enum_last` (`anyenum`) → `anyenum`
Returns the last value of the input enum type.
`enum_last(null::rainbow)` → `purple`

`enum_range` (`anyenum`) → `anyarray`
Returns all values of the input enum type in an ordered array.
`enum_range(null::rainbow)` → `{red,orange,yellow,green,blue,purple}`

`enum_range` (`anyenum`, `anyenum`) → `anyarray`
Returns the range between the two given enum values, as an ordered array. The values must be from the same enum type. If the first parameter is null, the result will start with the first value of the enum type. If the second parameter is null, the result will end with the last value of the enum type.
`enum_range('orange'::rainbow, 'green'::rainbow)` → `{orange,yellow,green}`
`enum_range(NULL, 'green'::rainbow)` → `{red,orange,yellow,green}`
`enum_range('orange'::rainbow, NULL)` → `{orange,yellow,green,blue,purple}`

Notice that except for the two-argument form of `enum_range`, these functions disregard the specific value passed to them; they care only about its declared data type. Either null or a specific value of the type can be passed, with the same result. It is more common to apply these functions to a table column or function argument than to a hardwired type name as used in the examples.

9.11. Geometric Functions and Operators

The geometric types `point`, `box`, `lseg`, `line`, `path`, `polygon`, and `circle` have a large set of native support functions and operators, shown in Table 9.36, Table 9.37, and Table 9.38.

Table 9.36. Geometric Operators

Operator **Description** **Example(s)**
geometric_type + `point` → *geometric_type* Adds the coordinates of the second `point` to those of each point of the first argument, thus performing translation. Available for `point`, `box`, `path`, `circle`. `box '(1,1),(0,0)' + point '(2,0)'` → `(3,1),(2,0)`

`path` + `path` → `path`
Concatenates two open paths (returns NULL if either path is closed).
`path '[(0,0),(1,1)]' + path '[(2,2),(3,3),(4,4)]'` → `[(0,0),(1,1),(2,2),(3,3),(4,4)]`

geometric_type - `point` → *geometric_type* Subtracts the coordinates of the second `point` from those of each point of the first argument, thus performing translation. Available for `point`, `box`, `path`, `circle`. `box '(1,1),(0,0)' - point '(2,0)'` → `(-1,1),(-2,0)`

geometric_type * `point` → *geometric_type*
Multiplies each point of the first argument by the second `point` (treating a point as being a complex number represented by real and imaginary parts, and performing standard complex multiplication). If one interprets the second `point` as a vector, this is equivalent to scaling the object's size and distance from the origin by the length of the vector, and rotating it counterclockwise around the origin by the vector's angle from the *x* axis. Available for `point`, `box`,[a] `path`, `circle`.
`path '((0,0),(1,0),(1,1))' * point '(3.0,0)'` → `((0,0),(3,0),(3,3))`
`path '((0,0),(1,0),(1,1))' * point(cosd(45), sind(45))` → `((0,0),(0.7071067811865475,0.7071067811865475),(0,1.414213562373095))`

geometric_type / `point` → *geometric_type*
Divides each point of the first argument by the second `point` (treating a point as being a complex number represented by real and imaginary parts, and performing standard complex division). If one interprets the second `point` as a vector, this is equivalent to scaling the object's size and distance from the origin down by the length of the vector, and rotating it clockwise around the origin by the vector's angle from the *x* axis. Available for `point`, `box`,[a] `path`, `circle`.
`path '((0,0),(1,0),(1,1))' / point '(2.0,0)'` → `((0,0),(0.5,0),(0.5,0.5))`
`path '((0,0),(1,0),(1,1))' / point(cosd(45), sind(45))` → `((0,0),(0.7071067811865476,-0.7071067811865476),(1.4142135623730951,0))`

@-@ *geometric_type* → `double precision` Computes the total length. Available for `lseg`, `path`. `@-@ path '[(0,0),(1,0),(1,1)]'` → `2`
@@ *geometric_type* → `point`

Operator **Description** **Example(s)**
Computes the center point. Available for `box`, `lseg`, `polygon`, `circle`. `@@ box '(2,2),(0,0)'` → `(1,1)`
`geometric_type` → `integer` Returns the number of points. Available for `path`, `polygon`. `# path '((1,0),(0,1),(-1,0))'` → `3`
`geometric_type` # *`geometric_type`* → `point` Computes the point of intersection, or NULL if there is none. Available for `lseg`, `line`. `lseg '[(0,0),(1,1)]' # lseg '[(1,0),(0,1)]'` → `(0.5,0.5)`

`box` # `box` → `box`
Computes the intersection of two boxes, or NULL if there is none.
`box '(2,2),(-1,-1)' # box '(1,1),(-2,-2)'` → `(1,1),(-1,-1)`

`geometric_type` ## *`geometric_type`* → `point` Computes the closest point to the first object on the second object. Available for these pairs of types: (`point`, `box`), (`point`, `lseg`), (`point`, `line`), (`lseg`, `box`), (`lseg`, `lseg`), (`line`, `lseg`). `point '(0,0)' ## lseg '[(2,0),(0,2)]'` → `(1,1)`
`geometric_type` <-> *`geometric_type`* → `double precision` Computes the distance between the objects. Available for all seven geometric types, for all combinations of `point` with another geometric type, and for these additional pairs of types: (`box`, `lseg`), (`lseg`, `line`), (`polygon`, `circle`) (and the commutator cases). `circle '<(0,0),1>' <-> circle '<(5,0),1>'` → `3`
`geometric_type` @> *`geometric_type`* → `boolean` Does first object contain second? Available for these pairs of types: (`box`, `point`), (`box`, `box`), (`path`, `point`), (`polygon`, `point`), (`polygon`, `polygon`), (`circle`, `point`), (`circle`, `circle`). `circle '<(0,0),2>' @> point '(1,1)'` → `t`
`geometric_type` <@ *`geometric_type`* → `boolean` Is first object contained in or on second? Available for these pairs of types: (`point`, `box`), (`point`, `lseg`), (`point`, `line`), (`point`, `path`), (`point`, `polygon`), (`point`, `circle`), (`box`, `box`), (`lseg`, `box`), (`lseg`, `line`), (`polygon`, `polygon`), (`circle`, `circle`). `point '(1,1)' <@ circle '<(0,0),2>'` → `t`
`geometric_type` && *`geometric_type`* → `boolean` Do these objects overlap? (One point in common makes this true.) Available for `box`, `polygon`, `circle`. `box '(1,1),(0,0)' && box '(2,2),(0,0)'` → `t`
`geometric_type` << *`geometric_type`* → `boolean` Is first object strictly left of second? Available for `point`, `box`, `polygon`, `circle`. `circle '<(0,0),1>' << circle '<(5,0),1>'` → `t`
`geometric_type` >> *`geometric_type`* → `boolean` Is first object strictly right of second? Available for `point`, `box`, `polygon`, `circle`. `circle '<(5,0),1>' >> circle '<(0,0),1>'` → `t`

Operator **Description** **Example(s)**
geometric_type &< *geometric_type* → boolean Does first object not extend to the right of second? Available for box, polygon, circle. box '(1,1),(0,0)' &< box '(2,2),(0,0)' → t
geometric_type &> *geometric_type* → boolean Does first object not extend to the left of second? Available for box, polygon, circle. box '(3,3),(0,0)' &> box '(2,2),(0,0)' → t
geometric_type <<| *geometric_type* → boolean Is first object strictly below second? Available for point, box, polygon, circle. box '(3,3),(0,0)' <<| box '(5,5),(3,4)' → t
geometric_type |>> *geometric_type* → boolean Is first object strictly above second? Available for point, box, polygon, circle. box '(5,5),(3,4)' |>> box '(3,3),(0,0)' → t
geometric_type &<| *geometric_type* → boolean Does first object not extend above second? Available for box, polygon, circle. box '(1,1),(0,0)' &<| box '(2,2),(0,0)' → t
geometric_type |&> *geometric_type* → boolean Does first object not extend below second? Available for box, polygon, circle. box '(3,3),(0,0)' |&> box '(2,2),(0,0)' → t
box <^ box → boolean Is first object below second (allows edges to touch)? box '((1,1),(0,0))' <^ box '((2,2),(1,1))' → t
box >^ box → boolean Is first object above second (allows edges to touch)? box '((2,2),(1,1))' >^ box '((1,1),(0,0))' → t
geometric_type ?# *geometric_type* → boolean Do these objects intersect? Available for these pairs of types: (box, box), (lseg, box), (lseg, lseg), (lseg, line), (line, box), (line, line), (path, path). lseg '[(-1,0),(1,0)]' ?# box '(2,2),(-2,-2)' → t
?- line → boolean ?- lseg → boolean Is line horizontal? ?- lseg '[(-1,0),(1,0)]' → t
point ?- point → boolean Are points horizontally aligned (that is, have same y coordinate)? point '(1,0)' ?- point '(0,0)' → t
?| line → boolean ?| lseg → boolean Is line vertical? ?| lseg '[(-1,0),(1,0)]' → f

Operator Description Example(s)

`point ?| point` → `boolean`
Are points vertically aligned (that is, have same x coordinate)?
`point '(0,1)' ?| point '(0,0)'` → `t`

`line ?-| line` → `boolean`
`lseg ?-| lseg` → `boolean`
Are lines perpendicular?
`lseg '[(0,0),(0,1)]' ?-| lseg '[(0,0),(1,0)]'` → `t`

`line ?|| line` → `boolean`
`lseg ?|| lseg` → `boolean`
Are lines parallel?
`lseg '[(-1,0),(1,0)]' ?|| lseg '[(-1,2),(1,2)]'` → `t`

geometric_type `~=` *geometric_type* → `boolean` Are these objects the same? Available for `point`, `box`, `polygon`, `circle`. `polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'` → `t`

[a] "Rotating" a box with these operators only moves its corner points: the box is still considered to have sides parallel to the axes. Hence the box's size is not preserved, as a true rotation would do.

> **Caution**
>
> Note that the "same as" operator, `~=`, represents the usual notion of equality for the `point`, `box`, `polygon`, and `circle` types. Some of the geometric types also have an `=` operator, but `=` compares for equal *areas* only. The other scalar comparison operators (`<=` and so on), where available for these types, likewise compare areas.

> **Note**
>
> Before PostgreSQL 14, the point is strictly below/above comparison operators `point <<\| point` and `point \|>> point` were respectively called `<^` and `>^`. These names are still available, but are deprecated and will eventually be removed.

Table 9.37. Geometric Functions

Function Description Example(s)

`area(` *geometric_type* `)` → `double precision`
Computes area. Available for `box`, `path`, `circle`. A `path` input must be closed, else NULL is returned. Also, if the `path` is self-intersecting, the result may be meaningless.
`area(box '(2,2),(0,0)')` → `4`

`center(` *geometric_type* `)` → `point`
Computes center point. Available for `box`, `circle`.
`center(box '(1,2),(0,0)')` → `(0.5,1)`

`diagonal(box)` → `lseg` Extracts box's diagonal as a line segment (same as `lseg(box)`).

Function Description Example(s)
`diagonal(box '(1,2),(0,0)')` → `[(1,2),(0,0)]`

`diameter ( circle )` → `double precision`
Computes diameter of circle.
`diameter(circle '<(0,0),2>')` → `4`

`height ( box )` → `double precision`
Computes vertical size of box.
`height(box '(1,2),(0,0)')` → `2`

`isclosed ( path )` → `boolean`
Is path closed?
`isclosed(path '((0,0),(1,1),(2,0))')` → `t`

`isopen ( path )` → `boolean`
Is path open?
`isopen(path '[(0,0),(1,1),(2,0)]')` → `t`

`length ( geometric_type )` → `double precision`
Computes the total length. Available for `lseg`, `path`.
`length(path '((-1,0),(1,0))')` → `4`

`npoints ( geometric_type )` → `integer`
Returns the number of points. Available for `path`, `polygon`.
`npoints(path '[(0,0),(1,1),(2,0)]')` → `3`

`pclose ( path )` → `path`
Converts path to closed form.
`pclose(path '[(0,0),(1,1),(2,0)]')` → `((0,0),(1,1),(2,0))`

`popen ( path )` → `path`
Converts path to open form.
`popen(path '((0,0),(1,1),(2,0))')` → `[(0,0),(1,1),(2,0)]`

`radius ( circle )` → `double precision`
Computes radius of circle.
`radius(circle '<(0,0),2>')` → `2`

`slope ( point, point )` → `double precision`
Computes slope of a line drawn through the two points.
`slope(point '(0,0)', point '(2,1)')` → `0.5`

`width ( box )` → `double precision`
Computes horizontal size of box.
`width(box '(1,2),(0,0)')` → `1`

Table 9.38. Geometric Type Conversion Functions

Function Description Example(s)
`box ( circle )` → `box` Computes box inscribed within the circle.

Function **Description** **Example(s)**
`box(circle '<(0,0),2>')` → `(1.414213562373095,1.414213562373095),(-1.414213562373095,-1.414213562373095)`

`box ( point )` → `box`
Converts point to empty box.
`box(point '(1,0)')` → `(1,0),(1,0)`

`box ( point, point )` → `box`
Converts any two corner points to box.
`box(point '(0,1)', point '(1,0)')` → `(1,1),(0,0)`

`box ( polygon )` → `box`
Computes bounding box of polygon.
`box(polygon '((0,0),(1,1),(2,0))')` → `(2,1),(0,0)`

`bound_box ( box, box )` → `box`
Computes bounding box of two boxes.
`bound_box(box '(1,1),(0,0)', box '(4,4),(3,3)')` → `(4,4),(0,0)`

`circle ( box )` → `circle`
Computes smallest circle enclosing box.
`circle(box '(1,1),(0,0)')` → `<(0.5,0.5),0.7071067811865476>`

`circle ( point, double precision )` → `circle`
Constructs circle from center and radius.
`circle(point '(0,0)', 2.0)` → `<(0,0),2>`

`circle ( polygon )` → `circle`
Converts polygon to circle. The circle's center is the mean of the positions of the polygon's points, and the radius is the average distance of the polygon's points from that center.
`circle(polygon '((0,0),(1,3),(2,0))')` → `<(1,1),1.6094757082487299>`

`line ( point, point )` → `line`
Converts two points to the line through them.
`line(point '(-1,0)', point '(1,0)')` → `{0,-1,0}`

`lseg ( box )` → `lseg`
Extracts box's diagonal as a line segment.
`lseg(box '(1,0),(-1,0)')` → `[(1,0),(-1,0)]`

`lseg ( point, point )` → `lseg`
Constructs line segment from two endpoints.
`lseg(point '(-1,0)', point '(1,0)')` → `[(-1,0),(1,0)]`

`path ( polygon )` → `path`
Converts polygon to a closed path with the same list of points.
`path(polygon '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))`

`point ( double precision, double precision )` → `point` Constructs point from its coordinates.

Function **Description** **Example(s)**
`point(23.4, -44.5)` → `(23.4,-44.5)`

`point ( box )` → `point`
Computes center of box.
`point(box '(1,0),(-1,0)')` → `(0,0)`

`point ( circle )` → `point`
Computes center of circle.
`point(circle '<(0,0),2>')` → `(0,0)`

`point ( lseg )` → `point`
Computes center of line segment.
`point(lseg '[(-1,0),(1,0)]')` → `(0,0)`

`point ( polygon )` → `point`
Computes center of polygon (the mean of the positions of the polygon's points).
`point(polygon '((0,0),(1,1),(2,0))')` → `(1,0.3333333333333333)`

`polygon ( box )` → `polygon`
Converts box to a 4-point polygon.
`polygon(box '(1,1),(0,0)')` → `((0,0),(0,1),(1,1),(1,0))`

`polygon ( circle )` → `polygon`
Converts circle to a 12-point polygon.
`polygon(circle '<(0,0),2>')` → `((-2,0), (-1.7320508075688774,0.9999999999999999), (-1.0000000000000002,1.7320508075688772), (-1.2246063538223773e-16,2), (0.9999999999999996,1.7320508075688774), (1.732050807568877,1.0000000000000007), (2,2.4492127076447545e-16), (1.7320508075688776,-0.9999999999999994), (1.0000000000000009,-1.7320508075688767), (3.673819061467132e-16,-2), (-0.9999999999999987,-1.732050807568878), (-1.7320508075688767,-1.0000000000000009))`

`polygon ( integer, circle )` → `polygon`
Converts circle to an *n*-point polygon.
`polygon(4, circle '<(3,0),1>')` → `((2,0),(3,1), (4,1.2246063538223773e-16),(3,-1))`

`polygon ( path )` → `polygon`
Converts closed path to a polygon with the same list of points.
`polygon(path '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))`

It is possible to access the two component numbers of a `point` as though the point were an array with indexes 0 and 1. For example, if `t.p` is a `point` column then `SELECT p[0] FROM t` retrieves the X coordinate and `UPDATE t SET p[1] = ...` changes the Y coordinate. In the same way, a value of type `box` or `lseg` can be treated as an array of two `point` values.

9.12. Network Address Functions and Operators

The IP network address types, `cidr` and `inet`, support the usual comparison operators shown in Table 9.1 as well as the specialized operators and functions shown in Table 9.39 and Table 9.40.

Any `cidr` value can be cast to `inet` implicitly; therefore, the operators and functions shown below as operating on `inet` also work on `cidr` values. (Where there are separate functions for `inet` and `cidr`, it is because the behavior should be different for the two cases.) Also, it is permitted to cast an `inet` value to `cidr`. When this is done, any bits to the right of the netmask are silently zeroed to create a valid `cidr` value.

Table 9.39. IP Address Operators

Operator **Description** **Example(s)**

`inet << inet → boolean`
Is subnet strictly contained by subnet? This operator, and the next four, test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any bits to the right of the netmasks) and determine whether one network is identical to or a subnet of the other.
`inet '192.168.1.5' << inet '192.168.1/24' → t`
`inet '192.168.0.5' << inet '192.168.1/24' → f`
`inet '192.168.1/24' << inet '192.168.1/24' → f`

`inet <<= inet → boolean`
Is subnet contained by or equal to subnet?
`inet '192.168.1/24' <<= inet '192.168.1/24' → t`

`inet >> inet → boolean`
Does subnet strictly contain subnet?
`inet '192.168.1/24' >> inet '192.168.1.5' → t`

`inet >>= inet → boolean`
Does subnet contain or equal subnet?
`inet '192.168.1/24' >>= inet '192.168.1/24' → t`

`inet && inet → boolean`
Does either subnet contain or equal the other?
`inet '192.168.1/24' && inet '192.168.1.80/28' → t`
`inet '192.168.1/24' && inet '192.168.2.0/28' → f`

`~ inet → inet`
Computes bitwise NOT.
`~ inet '192.168.1.6' → 63.87.254.249`

`inet & inet → inet`
Computes bitwise AND.
`inet '192.168.1.6' & inet '0.0.0.255' → 0.0.0.6`

`inet | inet → inet`
Computes bitwise OR.
`inet '192.168.1.6' | inet '0.0.0.255' → 192.168.1.255`

Operator Description Example(s)

`inet + bigint → inet`
Adds an offset to an address.
`inet '192.168.1.6' + 25 → 192.168.1.31`

`bigint + inet → inet`
Adds an offset to an address.
`200 + inet '::ffff:fff0:1' → ::ffff:255.240.0.201`

`inet - bigint → inet`
Subtracts an offset from an address.
`inet '192.168.1.43' - 36 → 192.168.1.7`

`inet - inet → bigint`
Computes the difference of two addresses.
`inet '192.168.1.43' - inet '192.168.1.19' → 24`
`inet '::1' - inet '::ffff:1' → -4294901760`

Table 9.40. IP Address Functions

Function Description Example(s)

`abbrev ( inet ) → text`
Creates an abbreviated display format as text. (The result is the same as the `inet` output function produces; it is "abbreviated" only in comparison to the result of an explicit cast to `text`, which for historical reasons will never suppress the netmask part.)
`abbrev(inet '10.1.0.0/32') → 10.1.0.0`

`abbrev ( cidr ) → text`
Creates an abbreviated display format as text. (The abbreviation consists of dropping all-zero octets to the right of the netmask; more examples are in Table 8.22.)
`abbrev(cidr '10.1.0.0/16') → 10.1/16`

`broadcast ( inet ) → inet`
Computes the broadcast address for the address's network.
`broadcast(inet '192.168.1.5/24') → 192.168.1.255/24`

`family ( inet ) → integer`
Returns the address's family: `4` for IPv4, `6` for IPv6.
`family(inet '::1') → 6`

`host ( inet ) → text`
Returns the IP address as text, ignoring the netmask.
`host(inet '192.168.1.0/24') → 192.168.1.0`

`hostmask ( inet ) → inet`
Computes the host mask for the address's network.
`hostmask(inet '192.168.23.20/30') → 0.0.0.3`

`inet_merge ( inet, inet ) → cidr` Computes the smallest network that includes both of the given networks.

Function Description Example(s)
`inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24')` → `192.168.0.0/22`

`inet_same_family` (`inet`, `inet`) → `boolean`
Tests whether the addresses belong to the same IP family.
`inet_same_family(inet '192.168.1.5/24', inet '::1')` → `f`

`masklen` (`inet`) → `integer`
Returns the netmask length in bits.
`masklen(inet '192.168.1.5/24')` → `24`

`netmask` (`inet`) → `inet`
Computes the network mask for the address's network.
`netmask(inet '192.168.1.5/24')` → `255.255.255.0`

`network` (`inet`) → `cidr`
Returns the network part of the address, zeroing out whatever is to the right of the netmask. (This is equivalent to casting the value to `cidr`.)
`network(inet '192.168.1.5/24')` → `192.168.1.0/24`

`set_masklen` (`inet`, `integer`) → `inet`
Sets the netmask length for an `inet` value. The address part does not change.
`set_masklen(inet '192.168.1.5/24', 16)` → `192.168.1.5/16`

`set_masklen` (`cidr`, `integer`) → `cidr`
Sets the netmask length for a `cidr` value. Address bits to the right of the new netmask are set to zero.
`set_masklen(cidr '192.168.1.0/24', 16)` → `192.168.0.0/16`

`text` (`inet`) → `text`
Returns the unabbreviated IP address and netmask length as text. (This has the same result as an explicit cast to `text`.)
`text(inet '192.168.1.5')` → `192.168.1.5/32`

> **Tip**
>
> The `abbrev`, `host`, and `text` functions are primarily intended to offer alternative display formats for IP addresses.

The MAC address types, `macaddr` and `macaddr8`, support the usual comparison operators shown in Table 9.1 as well as the specialized functions shown in Table 9.41. In addition, they support the bitwise logical operators ~, & and | (NOT, AND and OR), just as shown above for IP addresses.

Table 9.41. MAC Address Functions

Function Description Example(s)
`trunc` (`macaddr`) → `macaddr` Sets the last 3 bytes of the address to zero. The remaining prefix can be associated with a particular manufacturer (using data not included in PostgreSQL).

Function Description Example(s)
`trunc(macaddr '12:34:56:78:90:ab')` → `12:34:56:00:00:00`

`trunc ( macaddr8 )` → `macaddr8`
Sets the last 5 bytes of the address to zero. The remaining prefix can be associated with a particular manufacturer (using data not included in PostgreSQL).
`trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')` → `12:34:56:00:00:00:00:00`

`macaddr8_set7bit ( macaddr8 )` → `macaddr8`
Sets the 7th bit of the address to one, creating what is known as modified EUI-64, for inclusion in an IPv6 address.
`macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef')` → `02:34:56:ff:fe:ab:cd:ef`

9.13. Text Search Functions and Operators

Table 9.42, Table 9.43 and Table 9.44 summarize the functions and operators that are provided for full text searching. See Chapter 12 for a detailed explanation of PostgreSQL's text search facility.

Table 9.42. Text Search Operators

Operator Description Example(s)

`tsvector @@ tsquery` → `boolean`
`tsquery @@ tsvector` → `boolean`
Does `tsvector` match `tsquery`? (The arguments can be given in either order.)
`to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')` → `t`

`text @@ tsquery` → `boolean`
Does text string, after implicit invocation of `to_tsvector()`, match `tsquery`?
`'fat cats ate rats' @@ to_tsquery('cat & rat')` → `t`

`tsvector || tsvector` → `tsvector`
Concatenates two `tsvector`s. If both inputs contain lexeme positions, the second input's positions are adjusted accordingly.
`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector` → `'a':1 'b':2,5 'c':3 'd':4`

`tsquery && tsquery` → `tsquery`
ANDs two `tsquery`s together, producing a query that matches documents that match both input queries.
`'fat | rat'::tsquery && 'cat'::tsquery` → `( 'fat' | 'rat' ) & 'cat'`

`tsquery || tsquery` → `tsquery`
ORs two `tsquery`s together, producing a query that matches documents that match either input query.
`'fat | rat'::tsquery || 'cat'::tsquery` → `'fat' | 'rat' | 'cat'`

Operator **Description** **Example(s)**

`!! tsquery → tsquery`
Negates a `tsquery`, producing a query that matches documents that do not match the input query.
`!! 'cat'::tsquery → !'cat'`

`tsquery <-> tsquery → tsquery`
Constructs a phrase query, which matches if the two input queries match at successive lexemes.
`to_tsquery('fat') <-> to_tsquery('rat') → 'fat' <-> 'rat'`

`tsquery @> tsquery → boolean`
Does first `tsquery` contain the second? (This considers only whether all the lexemes appearing in one query appear in the other, ignoring the combining operators.)
`'cat'::tsquery @> 'cat & rat'::tsquery → f`

`tsquery <@ tsquery → boolean`
Is first `tsquery` contained in the second? (This considers only whether all the lexemes appearing in one query appear in the other, ignoring the combining operators.)
`'cat'::tsquery <@ 'cat & rat'::tsquery → t`
`'cat'::tsquery <@ '!cat & rat'::tsquery → t`

In addition to these specialized operators, the usual comparison operators shown in Table 9.1 are available for types `tsvector` and `tsquery`. These are not very useful for text searching but allow, for example, unique indexes to be built on columns of these types.

Table 9.43. Text Search Functions

Function **Description** **Example(s)**

`array_to_tsvector ( text[] ) → tsvector`
Converts an array of text strings to a `tsvector`. The given strings are used as lexemes as-is, without further processing. Array elements must not be empty strings or `NULL`.
`array_to_tsvector('{fat,cat,rat}'::text[]) → 'cat' 'fat' 'rat'`

`get_current_ts_config ( ) → regconfig`
Returns the OID of the current default text search configuration (as set by default_text_search_config).
`get_current_ts_config() → english`

`length ( tsvector ) → integer`
Returns the number of lexemes in the `tsvector`.
`length('fat:2,4 cat:3 rat:5A'::tsvector) → 3`

`numnode ( tsquery ) → integer`
Returns the number of lexemes plus operators in the `tsquery`.
`numnode('(fat & rat) | cat'::tsquery) → 5`

`plainto_tsquery ( [ config regconfig, ] query text ) → tsquery` Converts text to a `tsquery`, normalizing words according to the specified or default configuration. Any punctuation in the string is ignored (it does not determine query operators). The resulting query matches documents containing all non-stopwords in the text.

Function Description Example(s)
`plainto_tsquery('english', 'The Fat Rats')` → `'fat' & 'rat'`

`phraseto_tsquery` ([*`config`* `regconfig`,] *`query`* `text`) → `tsquery`
Converts text to a `tsquery`, normalizing words according to the specified or default configuration. Any punctuation in the string is ignored (it does not determine query operators). The resulting query matches phrases containing all non-stopwords in the text.
`phraseto_tsquery('english', 'The Fat Rats')` → `'fat' <-> 'rat'`
`phraseto_tsquery('english', 'The Cat and Rats')` → `'cat' <2> 'rat'`

`websearch_to_tsquery` ([*`config`* `regconfig`,] *`query`* `text`) → `tsquery`
Converts text to a `tsquery`, normalizing words according to the specified or default configuration. Quoted word sequences are converted to phrase tests. The word “or” is understood as producing an OR operator, and a dash produces a NOT operator; other punctuation is ignored. This approximates the behavior of some common web search tools.
`websearch_to_tsquery('english', '"fat rat" or cat dog')` → `'fat' <-> 'rat' | 'cat' & 'dog'`

`querytree` (`tsquery`) → `text`
Produces a representation of the indexable portion of a `tsquery`. A result that is empty or just `T` indicates a non-indexable query.
`querytree('foo & ! bar'::tsquery)` → `'foo'`

`setweight` (*`vector`* `tsvector`, *`weight`* `"char"`) → `tsvector`
Assigns the specified *`weight`* to each element of the *`vector`*.
`setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')` → `'cat':3A 'fat':2A,4A 'rat':5A`

`setweight` (*`vector`* `tsvector`, *`weight`* `"char"`, *`lexemes`* `text[]`) → `tsvector`
Assigns the specified *`weight`* to elements of the *`vector`* that are listed in *`lexemes`*. The strings in *`lexemes`* are taken as lexemes as-is, without further processing. Strings that do not match any lexeme in *`vector`* are ignored.
`setweight('fat:2,4 cat:3 rat:5,6B'::tsvector, 'A', '{cat,rat}')` → `'cat':3A 'fat':2,4 'rat':5A,6A`

`strip` (`tsvector`) → `tsvector`
Removes positions and weights from the `tsvector`.
`strip('fat:2,4 cat:3 rat:5A'::tsvector)` → `'cat' 'fat' 'rat'`

`to_tsquery` ([*`config`* `regconfig`,] *`query`* `text`) → `tsquery`
Converts text to a `tsquery`, normalizing words according to the specified or default configuration. The words must be combined by valid `tsquery` operators.
`to_tsquery('english', 'The & Fat & Rats')` → `'fat' & 'rat'`

`to_tsvector` ([*`config`* `regconfig`,] *`document`* `text`) → `tsvector`
Converts text to a `tsvector`, normalizing words according to the specified or default configuration. Position information is included in the result.
`to_tsvector('english', 'The Fat Rats')` → `'fat':2 'rat':3`

`to_tsvector` ([*`config`* `regconfig`,] *`document`* `json`) → `tsvector` `to_tsvector` ([*`config`* `regconfig`,] *`document`* `jsonb`) → `tsvector`

Function Description Example(s)

Converts each string value in the JSON document to a `tsvector`, normalizing words according to the specified or default configuration. The results are then concatenated in document order to produce the output. Position information is generated as though one stopword exists between each pair of string values. (Beware that "document order" of the fields of a JSON object is implementation-dependent when the input is `jsonb`; observe the difference in the examples.)
`to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json)` → `'dog':5 'fat':2 'rat':3`
`to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb)` → `'dog':1 'fat':4 'rat':5`

`json_to_tsvector` ([*`config`* `regconfig`,] *`document`* `json`, *`filter`* `jsonb`) → `tsvector`
`jsonb_to_tsvector` ([*`config`* `regconfig`,] *`document`* `jsonb`, *`filter`* `jsonb`) → `tsvector`
Selects each item in the JSON document that is requested by the *`filter`* and converts each one to a `tsvector`, normalizing words according to the specified or default configuration. The results are then concatenated in document order to produce the output. Position information is generated as though one stopword exists between each pair of selected items. (Beware that "document order" of the fields of a JSON object is implementation-dependent when the input is `jsonb`.) The *`filter`* must be a `jsonb` array containing zero or more of these keywords: `"string"` (to include all string values), `"numeric"` (to include all numeric values), `"boolean"` (to include all boolean values), `"key"` (to include all keys), or `"all"` (to include all the above). As a special case, the *`filter`* can also be a simple JSON value that is one of these keywords.
`json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')` → `'123':5 'fat':2 'rat':3`
`json_to_tsvector('english', '{"cat": "The Fat Rats", "dog": 123}'::json, '"all"')` → `'123':9 'cat':1 'dog':7 'fat':4 'rat':5`

`ts_delete` (*`vector`* `tsvector`, *`lexeme`* `text`) → `tsvector`
Removes any occurrence of the given *`lexeme`* from the *`vector`*. The *`lexeme`* string is treated as a lexeme as-is, without further processing.
`ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')` → `'cat':3 'rat':5A`

`ts_delete` (*`vector`* `tsvector`, *`lexemes`* `text[]`) → `tsvector`
Removes any occurrences of the lexemes in *`lexemes`* from the *`vector`*. The strings in *`lexemes`* are taken as lexemes as-is, without further processing. Strings that do not match any lexeme in *`vector`* are ignored.
`ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])` → `'cat':3`

`ts_filter` (*`vector`* `tsvector`, *`weights`* `"char"[]`) → `tsvector`
Selects only elements with the given *`weights`* from the *`vector`*.
`ts_filter('fat:2,4 cat:3b,7c rat:5A'::tsvector, '{a,b}')` → `'cat':3B 'rat':5A`

`ts_headline` ([*`config`* `regconfig`,] *`document`* `text`, *`query`* `tsquery` [, *`options`* `text`]) → `text`

Function Description Example(s)
Displays, in an abbreviated form, the match(es) for the *query* in the *document*, which must be raw text not a tsvector. Words in the document are normalized according to the specified or default configuration before matching to the query. Use of this function is discussed in Section 12.3.4, which also describes the available *options*. ts_headline('The fat cat ate the rat.', 'cat') → The fat cat ate the rat.
ts_headline ([*config* regconfig,] *document* json, *query* tsquery [, *options* text]) → text ts_headline ([*config* regconfig,] *document* jsonb, *query* tsquery [, *options* text]) → text Displays, in an abbreviated form, match(es) for the *query* that occur in string values within the JSON *document*. See Section 12.3.4 for more details. ts_headline('{"cat":"raining cats and dogs"}'::jsonb, 'cat') → {"cat": "raining cats and dogs"}
ts_rank ([*weights* real[],] *vector* tsvector, *query* tsquery [, *normalization* integer]) → real Computes a score showing how well the *vector* matches the *query*. See Section 12.3.3 for details. ts_rank(to_tsvector('raining cats and dogs'), 'cat') → 0.06079271
ts_rank_cd ([*weights* real[],] *vector* tsvector, *query* tsquery [, *normalization* integer]) → real Computes a score showing how well the *vector* matches the *query*, using a cover density algorithm. See Section 12.3.3 for details. ts_rank_cd(to_tsvector('raining cats and dogs'), 'cat') → 0.1
ts_rewrite (*query* tsquery, *target* tsquery, *substitute* tsquery) → tsquery Replaces occurrences of *target* with *substitute* within the *query*. See Section 12.4.2.1 for details. ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery) → 'b' & ('foo' | 'bar')
ts_rewrite (*query* tsquery, *select* text) → tsquery Replaces portions of the *query* according to target(s) and substitute(s) obtained by executing a SELECT command. See Section 12.4.2.1 for details. SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases') → 'b' & ('foo' | 'bar')
tsquery_phrase (*query1* tsquery, *query2* tsquery) → tsquery Constructs a phrase query that searches for matches of *query1* and *query2* at successive lexemes (same as <-> operator). tsquery_phrase(to_tsquery('fat'), to_tsquery('cat')) → 'fat' <-> 'cat'
tsquery_phrase (*query1* tsquery, *query2* tsquery, *distance* integer) → tsquery Constructs a phrase query that searches for matches of *query1* and *query2* that occur exactly *distance* lexemes apart.

Function Description Example(s)
`tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)` → `'fat' <10> 'cat'`

`tsvector_to_array(tsvector)` → `text[]`
Converts a `tsvector` to an array of lexemes.
`tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)` → `{cat,fat,rat}`

`unnest(tsvector)` → `setof record(`*`lexeme`* `text,`*`positions`* `smallint[],`*`weights`* `text)`
Expands a `tsvector` into a set of rows, one per lexeme.
`select * from unnest('cat:3 fat:2,4 rat:5A'::tsvector)` →
 	
`lexeme | positions | weights`
`--------+-----------+---------`
`cat | {3} | {D}`
`fat | {2,4} | {D,D}`
`rat | {5} | {A}`

> **Note**
>
> All the text search functions that accept an optional `regconfig` argument will use the configuration specified by default_text_search_config when that argument is omitted.

The functions in Table 9.44 are listed separately because they are not usually used in everyday text searching operations. They are primarily helpful for development and debugging of new text search configurations.

Table 9.44. Text Search Debugging Functions

Function Description Example(s)

`ts_debug([`*`config`* `regconfig,]`*`document`* `text)` → `setof record(`*`alias`* `text,`*`description`* `text,`*`token`* `text,`*`dictionaries`* `regdictionary[],`*`dictionary`* `regdictionary,`*`lexemes`* `text[])`
Extracts and normalizes tokens from the *`document`* according to the specified or default text search configuration, and returns information about how each token was processed. See Section 12.8.1 for details.
`ts_debug('english', 'The Brightest supernovaes')` → `(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...`

`ts_lexize(`*`dict`* `regdictionary,`*`token`* `text)` → `text[]`
Returns an array of replacement lexemes if the input token is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or NULL if it is not a known word. See Section 12.8.3 for details.
`ts_lexize('english_stem', 'stars')` → `{star}`

`ts_parse(`*`parser_name`* `text,`*`document`* `text)` → `setof record(`*`tokid`* `integer,`*`token`* `text)`

Function Description Example(s)
Extracts tokens from the *document* using the named parser. See Section 12.8.2 for details. `ts_parse('default', 'foo - bar')` → `(1,foo) ...`

`ts_parse(` *parser_oid* `oid,` *document* `text )` → `setof record(` *tokid* `integer,` *token* `text )`
Extracts tokens from the *document* using a parser specified by OID. See Section 12.8.2 for details.
`ts_parse(3722, 'foo - bar')` → `(1,foo) ...`

`ts_token_type(` *parser_name* `text )` → `setof record(` *tokid* `integer,` *alias* `text,` *description* `text )`
Returns a table that describes each type of token the named parser can recognize. See Section 12.8.2 for details.
`ts_token_type('default')` → `(1,asciiword,"Word, all ASCII") ...`

`ts_token_type(` *parser_oid* `oid )` → `setof record(` *tokid* `integer,` *alias* `text,` *description* `text )`
Returns a table that describes each type of token a parser specified by OID can recognize. See Section 12.8.2 for details.
`ts_token_type(3722)` → `(1,asciiword,"Word, all ASCII") ...`

`ts_stat(` *sqlquery* `text [,` *weights* `text ])` → `setof record(` *word* `text,` *ndoc* `integer,` *nentry* `integer )`
Executes the *sqlquery*, which must return a single `tsvector` column, and returns statistics about each distinct lexeme contained in the data. See Section 12.4.4 for details.
`ts_stat('SELECT vector FROM apod')` → `(foo,10,15) ...`

9.14. UUID Functions

PostgreSQL includes one function to generate a UUID:

`gen_random_uuid () → uuid`

This function returns a version 4 (random) UUID. This is the most commonly used type of UUID and is appropriate for most applications.

The uuid-ossp module provides additional functions that implement other standard algorithms for generating UUIDs.

There are also functions to extract data from UUIDs:

`uuid_extract_timestamp (uuid) → timestamp with time zone`

This function extracts a `timestamp with time zone` from UUID version 1. For other versions, this function returns null. Note that the extracted timestamp is not necessarily exactly equal to the time the UUID was generated; this depends on the implementation that generated the UUID.

`uuid_extract_version (uuid) → smallint`

This function extracts the version from a UUID of the variant described by RFC 4122[2]. For other variants, this function returns null. For example, for a UUID generated by `gen_random_uuid`, this function will return 4.

PostgreSQL also provides the usual comparison operators shown in Table 9.1 for UUIDs.

9.15. XML Functions

The functions and function-like expressions described in this section operate on values of type `xml`. See Section 8.13 for information about the `xml` type. The function-like expressions `xmlparse` and `xmlserialize` for converting to and from type `xml` are documented there, not in this section.

Use of most of these functions requires PostgreSQL to have been built with `configure --with-libxml`.

9.15.1. Producing XML Content

A set of functions and function-like expressions is available for producing XML content from SQL data. As such, they are particularly suitable for formatting query results into XML documents for processing in client applications.

9.15.1.1. `xmltext`

```
xmltext ( text ) → xml
```

The function `xmltext` returns an XML value with a single text node containing the input argument as its content. Predefined entities like ampersand (&), left and right angle brackets (< >), and quotation marks ("") are escaped.

Example:

```
SELECT xmltext('< foo & bar >');
         xmltext
-------------------------
 &lt; foo &amp; bar &gt;
```

9.15.1.2. `xmlcomment`

```
xmlcomment ( text ) → xml
```

The function `xmlcomment` creates an XML value containing an XML comment with the specified text as content. The text cannot contain "`--`" or end with a "`-`", otherwise the resulting construct would not be a valid XML comment. If the argument is null, the result is null.

Example:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.15.1.3. `xmlconcat`

[2] https://datatracker.ietf.org/doc/html/rfc4122

```
xmlconcat ( xml [, ...] ) → xml
```

The function `xmlconcat` concatenates a list of individual XML values to create a single value containing an XML content fragment. Null values are omitted; the result is only null if there are no non-null arguments.

Example:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML declarations, if present, are combined as follows. If all argument values have the same XML version declaration, that version is used in the result, else no version is used. If all argument values have the standalone declaration value "yes", then that value is used in the result. If all argument values have a standalone declaration value and at least one is "no", then that is used in the result. Else the result will have no standalone declaration. If the result is determined to require a standalone declaration but no version declaration, a version declaration with version 1.0 will be used because XML requires an XML declaration to contain a version declaration. Encoding declarations are ignored and removed in all cases.

Example:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml
 version="1.1" standalone="no"?><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.15.1.4. `xmlelement`

```
xmlelement ( NAME name [, XMLATTRIBUTES ( attvalue [ AS attname ]
 [, ...] ) ] [, content [, ...]] ) → xml
```

The `xmlelement` expression produces an XML element with the given name, attributes, and content. The *`name`* and *`attname`* items shown in the syntax are simple identifiers, not values. The *`attvalue`* and *`content`* items are expressions, which can yield any PostgreSQL data type. The argument(s) within `XMLATTRIBUTES` generate attributes of the XML element; the *`content`* value(s) are concatenated to form its content.

Examples:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
```

```
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar),
 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

Element and attribute names that are not valid XML names are escaped by replacing the offending characters by the sequence _x*HHHH*_, where *HHHH* is the character's Unicode codepoint in hexadecimal notation. For example:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

An explicit attribute name need not be specified if the attribute value is a column reference, in which case the column's name will be used as the attribute name by default. In other cases, the attribute must be given an explicit name. So this example is valid:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

But these are not:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM
 test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Element content, if specified, will be formatted according to its data type. If the content is itself of type `xml`, complex XML documents can be constructed. For example:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Content of other types will be formatted into valid XML character data. This means in particular that the characters <, >, and & will be converted to entities. Binary data (data type `bytea`) will be represented in base64 or hex encoding, depending on the setting of the configuration parameter xmlbinary. The particular behavior for individual data types is expected to evolve in order to align the PostgreSQL mappings with those specified in SQL:2006 and later, as discussed in Section D.3.1.3.

9.15.1.5. `xmlforest`

```
xmlforest ( content [ AS name ] [, ...] ) → xml
```

The `xmlforest` expression produces an XML forest (sequence) of elements using the given names and content. As for `xmlelement`, each *`name`* must be a simple identifier, while the *`content`* expressions can have any data type.

Examples:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                xmlforest
----------------------------------
----------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</
column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</
column_name>
 ...
```

As seen in the second example, the element name can be omitted if the content value is a column reference, in which case the column name is used by default. Otherwise, a name must be specified.

Element names that are not valid XML names are escaped as shown for `xmlelement` above. Similarly, content data is escaped to make valid XML content, unless it is already of type `xml`.

Note that XML forests are not valid XML documents if they consist of more than one element, so it might be useful to wrap `xmlforest` expressions in `xmlelement`.

9.15.1.6. `xmlpi`

```
xmlpi ( NAME name [, content ] ) → xml
```

The `xmlpi` expression creates an XML processing instruction. As for `xmlelement`, the *`name`* must be a simple identifier, while the *`content`* expression can have any data type. The *`content`*, if present, must not contain the character sequence `?>`.

Example:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.15.1.7. `xmlroot`

```
xmlroot ( xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO
 VALUE} ] ) → xml
```

The `xmlroot` expression alters the properties of the root node of an XML value. If a version is specified, it replaces the value in the root node's version declaration; if a standalone setting is specified, it replaces the value in the root node's standalone declaration.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?
><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
--------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.15.1.8. `xmlagg`

```
xmlagg ( xml ) → xml
```

The function `xmlagg` is, unlike the other functions described here, an aggregate function. It concatenates the input values to the aggregate function call, much like `xmlconcat` does, except that concatenation occurs across rows rather than across expressions in a single row. See Section 9.21 for additional information about aggregate functions.

Example:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

To determine the order of the concatenation, an `ORDER BY` clause may be added to the aggregate call as described in Section 4.2.7. For example:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

The following non-standard approach used to be recommended in previous versions, and may still be useful in specific cases:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.15.2. XML Predicates

The expressions described in this section check properties of `xml` values.

9.15.2.1. `IS DOCUMENT`

```
xml IS DOCUMENT → boolean
```

The expression `IS DOCUMENT` returns true if the argument XML value is a proper XML document, false if it is not (that is, it is a content fragment), or null if the argument is null. See Section 8.13 about the difference between documents and content fragments.

9.15.2.2. `IS NOT DOCUMENT`

```
xml IS NOT DOCUMENT → boolean
```

The expression `IS NOT DOCUMENT` returns false if the argument XML value is a proper XML document, true if it is not (that is, it is a content fragment), or null if the argument is null.

9.15.2.3. `XMLEXISTS`

```
XMLEXISTS ( text PASSING [BY {REF|VALUE}] xml [BY
 {REF|VALUE}] ) → boolean
```

The function `xmlexists` evaluates an XPath 1.0 expression (the first argument), with the passed XML value as its context item. The function returns false if the result of that evaluation yields an empty node-set, true if it yields any other value. The function returns null if any argument is null. A nonnull value passed as the context item must be an XML document, not a content fragment or any non-XML value.

Example:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE
 '<towns><town>Toronto</town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

The `BY REF` and `BY VALUE` clauses are accepted in PostgreSQL, but are ignored, as discussed in Section D.3.2.

In the SQL standard, the `xmlexists` function evaluates an expression in the XML Query language, but PostgreSQL allows only an XPath 1.0 expression, as discussed in Section D.3.1.

9.15.2.4. `xml_is_well_formed`

```
xml_is_well_formed ( text ) → boolean
xml_is_well_formed_document ( text ) → boolean
xml_is_well_formed_content ( text ) → boolean
```

These functions check whether a `text` string represents well-formed XML, returning a Boolean result. `xml_is_well_formed_document` checks for a well-formed document, while `xml_is_well_formed_content` checks for well-formed content. `xml_is_well_formed` does the former if the xmloption configuration parameter is set to `DOCUMENT`, or the latter if it is set to `CONTENT`. This means that `xml_is_well_formed` is useful for seeing whether a simple cast to type `xml` will succeed, whereas the other two functions are useful for seeing whether the corresponding variants of `XMLPARSE` will succeed.

Examples:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://
postgresql.org/stuff">bar</pg:foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://
postgresql.org/stuff">bar</my:foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

The last example shows that the checks include whether namespaces are correctly matched.

9.15.3. Processing XML

To process values of data type `xml`, PostgreSQL offers the functions `xpath` and `xpath_exists`, which evaluate XPath 1.0 expressions, and the `XMLTABLE` table function.

9.15.3.1. `xpath`

`xpath ( `*`xpath`* `text, `*`xml`* `xml [, `*`nsarray`* `text[] ] ) → xml[]`

The function `xpath` evaluates the XPath 1.0 expression *`xpath`* (given as text) against the XML value *`xml`*. It returns an array of XML values corresponding to the node-set produced by the XPath expression. If the XPath expression returns a scalar value rather than a node-set, a single-element array is returned.

The second argument must be a well formed XML document. In particular, it must have a single root node element.

The optional third argument of the function is an array of namespace mappings. This array should be a two-dimensional `text` array with the length of the second axis being equal to 2 (i.e., it should be an array of arrays, each of which consists of exactly 2 elements). The first element of each array entry

is the namespace name (alias), the second the namespace URI. It is not required that aliases provided in this array be the same as those being used in the XML document itself (in other words, both in the XML document and in the `xpath` function context, aliases are *local*).

Example:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://
example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

To deal with default (anonymous) namespaces, do something like this:

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://
example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.15.3.2. `xpath_exists`

```
xpath_exists ( xpath text, xml xml [, nsarray text[] ] ) → boolean
```

The function `xpath_exists` is a specialized form of the `xpath` function. Instead of returning the individual XML values that satisfy the XPath 1.0 expression, this function returns a Boolean indicating whether the query was satisfied or not (specifically, whether it produced any value other than an empty node-set). This function is equivalent to the `XMLEXISTS` predicate, except that it also offers support for a namespace mapping argument.

Example:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://
example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.15.3.3. `xmltable`

```
XMLTABLE (
    [ XMLNAMESPACES ( namespace_uri AS namespace_name [, ...] ), ]
    row_expression PASSING [BY {REF|VALUE}] document_expression [BY
 {REF|VALUE}]
    COLUMNS name { type [PATH column_expression]
 [DEFAULT default_expression] [NOT NULL | NULL]
```

```
                    | FOR ORDINALITY }
            [, ...]
) → setof record
```

The `xmltable` expression produces a table based on an XML value, an XPath filter to extract rows, and a set of column definitions. Although it syntactically resembles a function, it can only appear as a table in a query's `FROM` clause.

The optional `XMLNAMESPACES` clause gives a comma-separated list of namespace definitions, where each *`namespace_uri`* is a `text` expression and each *`namespace_name`* is a simple identifier. It specifies the XML namespaces used in the document and their aliases. A default namespace specification is not currently supported.

The required *`row_expression`* argument is an XPath 1.0 expression (given as `text`) that is evaluated, passing the XML value *`document_expression`* as its context item, to obtain a set of XML nodes. These nodes are what `xmltable` transforms into output rows. No rows will be produced if the *`document_expression`* is null, nor if the *`row_expression`* produces an empty node-set or any value other than a node-set.

`document_expression` provides the context item for the *`row_expression`*. It must be a well-formed XML document; fragments/forests are not accepted. The `BY REF` and `BY VALUE` clauses are accepted but ignored, as discussed in Section D.3.2.

In the SQL standard, the `xmltable` function evaluates expressions in the XML Query language, but PostgreSQL allows only XPath 1.0 expressions, as discussed in Section D.3.1.

The required `COLUMNS` clause specifies the column(s) that will be produced in the output table. See the syntax summary above for the format. A name is required for each column, as is a data type (unless `FOR ORDINALITY` is specified, in which case type `integer` is implicit). The path, default and nullability clauses are optional.

A column marked `FOR ORDINALITY` will be populated with row numbers, starting with 1, in the order of nodes retrieved from the *`row_expression`*'s result node-set. At most one column may be marked `FOR ORDINALITY`.

> **Note**
>
> XPath 1.0 does not specify an order for nodes in a node-set, so code that relies on a particular order of the results will be implementation-dependent. Details can be found in Section D.3.1.2.

The *`column_expression`* for a column is an XPath 1.0 expression that is evaluated for each row, with the current node from the *`row_expression`* result as its context item, to find the value of the column. If no *`column_expression`* is given, then the column name is used as an implicit path.

If a column's XPath expression returns a non-XML value (which is limited to string, boolean, or double in XPath 1.0) and the column has a PostgreSQL type other than `xml`, the column will be set as if by assigning the value's string representation to the PostgreSQL type. (If the value is a boolean, its string representation is taken to be `1` or `0` if the output column's type category is numeric, otherwise `true` or `false`.)

If a column's XPath expression returns a non-empty set of XML nodes and the column's PostgreSQL type is `xml`, the column will be assigned the expression result exactly, if it is of document or content form. [3]

A non-XML result assigned to an `xml` output column produces content, a single text node with the string value of the result. An XML result assigned to a column of any other type may not have more

[3] A result containing more than one element node at the top level, or non-whitespace text outside of an element, is an example of content form. An XPath result can be of neither form, for example if it returns an attribute node selected from the element that contains it. Such a result will be put into content form with each such disallowed node replaced by its string value, as defined for the XPath 1.0 `string` function.

than one node, or an error is raised. If there is exactly one node, the column will be set as if by assigning the node's string value (as defined for the XPath 1.0 `string` function) to the PostgreSQL type.

The string value of an XML element is the concatenation, in document order, of all text nodes contained in that element and its descendants. The string value of an element with no descendant text nodes is an empty string (not `NULL`). Any `xsi:nil` attributes are ignored. Note that the whitespace-only `text()` node between two non-text elements is preserved, and that leading whitespace on a `text()` node is not flattened. The XPath 1.0 `string` function may be consulted for the rules defining the string value of other XML node types and non-XML values.

The conversion rules presented here are not exactly those of the SQL standard, as discussed in Section D.3.1.3.

If the path expression returns an empty node-set (typically, when it does not match) for a given row, the column will be set to `NULL`, unless a *`default_expression`* is specified; then the value resulting from evaluating that expression is used.

A *`default_expression`*, rather than being evaluated immediately when `xmltable` is called, is evaluated each time a default is needed for the column. If the expression qualifies as stable or immutable, the repeat evaluation may be skipped. This means that you can usefully use volatile functions like `nextval` in *`default_expression`*.

Columns may be marked `NOT NULL`. If the *`column_expression`* for a `NOT NULL` column does not match anything and there is no `DEFAULT` or the *`default_expression`* also evaluates to null, an error is reported.

Examples:

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit =
 "sq_km"]',
```

```
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ",
 SIZE[@unit!="sq_km"]/@unit)',
                        premier_name text PATH 'PREMIER_NAME'
 DEFAULT 'not specified');

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |
 size_other  | premier_name
----+------------+--------------+------------+------------
+--------------+--------------
  1 |          1 | Australia    | AU         |            |
      | not specified
  5 |          2 | Japan        | JP         |            | 145935
 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |
      | not specified
```

The following example shows concatenation of multiple text() nodes, usage of the column name as XPath filter, and the treatment of whitespace, XML comments and processing instructions:

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</
x>CC  </element>
  </root>
$$ AS data;

SELECT xmltable.*
  FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element
 text);
         element
-------------------------
   Hello2a2   bbbxxxCC
```

The following example illustrates how the `XMLNAMESPACES` clause can be used to specify a list of namespaces used in the XML document as well as in the XPath expressions:

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://
example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
             '/x:example/x:item'
                PASSING (SELECT data FROM xmldata)
                COLUMNS foo int PATH '@foo',
                  bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
```

```
   4 |   5
(3 rows)
```

9.15.4. Mapping Tables to XML

The following functions map the contents of relational tables to XML values. They can be thought of as XML export functionality:

```
table_to_xml ( table regclass, nulls boolean,
               tableforest boolean, targetns text ) → xml
query_to_xml ( query text, nulls boolean,
               tableforest boolean, targetns text ) → xml
cursor_to_xml ( cursor refcursor, count integer, nulls boolean,
                tableforest boolean, targetns text ) → xml
```

`table_to_xml` maps the content of the named table, passed as parameter *`table`*. The `regclass` type accepts strings identifying tables using the usual notation, including optional schema qualification and double quotes (see Section 8.19 for details). `query_to_xml` executes the query whose text is passed as parameter *`query`* and maps the result set. `cursor_to_xml` fetches the indicated number of rows from the cursor specified by the parameter *`cursor`*. This variant is recommended if large tables have to be mapped, because the result value is built up in memory by each function.

If *`tableforest`* is false, then the resulting XML document looks like this:

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

If *`tableforest`* is true, the result is an XML content fragment that looks like this:

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

If no table name is available, that is, when mapping a query or a cursor, the string `table` is used in the first format, `row` in the second format.

The choice between these formats is up to the user. The first format is a proper XML document, which will be important in many applications. The second format tends to be more useful in the `cur-`

sor_to_xml function if the result values are to be reassembled into one document later on. The functions for producing XML content discussed above, in particular xmlelement, can be used to alter the results to taste.

The data values are mapped in the same way as described for the function xmlelement above.

The parameter *nulls* determines whether null values should be included in the output. If true, null values in columns are represented as:

```
<columnname xsi:nil="true"/>
```

where xsi is the XML namespace prefix for XML Schema Instance. An appropriate namespace declaration will be added to the result value. If false, columns containing null values are simply omitted from the output.

The parameter *targetns* specifies the desired XML namespace of the result. If no particular namespace is wanted, an empty string should be passed.

The following functions return XML Schema documents describing the mappings performed by the corresponding functions above:

```
table_to_xmlschema ( table regclass, nulls boolean,
                     tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
                     tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
                      tableforest boolean, targetns text ) → xml
```

It is essential that the same parameters are passed in order to obtain matching XML data mappings and XML Schema documents.

The following functions produce XML data mappings and the corresponding XML Schema in one document (or forest), linked together. They can be useful where self-contained and self-describing results are wanted:

```
table_to_xml_and_xmlschema ( table regclass, nulls boolean,
                             tableforest boolean, targetns text
 ) → xml
query_to_xml_and_xmlschema ( query text, nulls boolean,
                             tableforest boolean, targetns text
 ) → xml
```

In addition, the following functions are available to produce analogous mappings of entire schemas or the entire current database:

```
schema_to_xml ( schema name, nulls boolean,
                tableforest boolean, targetns text ) → xml
schema_to_xmlschema ( schema name, nulls boolean,
                      tableforest boolean, targetns text ) → xml
schema_to_xml_and_xmlschema ( schema name, nulls boolean,
                              tableforest boolean, targetns text
 ) → xml

database_to_xml ( nulls boolean,
```

```
                    tableforest boolean, targetns text ) → xml
database_to_xmlschema ( nulls boolean,
                          tableforest boolean, targetns text ) → xml
database_to_xml_and_xmlschema ( nulls boolean,
                                  tableforest boolean, targetns text
 ) → xml
```

These functions ignore tables that are not readable by the current user. The database-wide functions additionally ignore schemas that the current user does not have USAGE (lookup) privilege for.

Note that these potentially produce a lot of data, which needs to be built up in memory. When requesting content mappings of large schemas or databases, it might be worthwhile to consider mapping the tables separately instead, possibly even through a cursor.

The result of a schema content mapping looks like this:

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

where the format of a table mapping depends on the *tableforest* parameter as explained above.

The result of a database content mapping looks like this:

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

where the schema mapping is as above.

As an example of using the output produced by these functions, Example 9.1 shows an XSLT stylesheet that converts the output of table_to_xml_and_xmlschema to an HTML document containing a tabular rendition of the table data. In a similar manner, the results from these functions can be converted into other XML-based formats.

Example 9.1. XSLT Stylesheet for Converting SQL/XML Output to HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/
xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=
$tabletypename]/xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=
$rowtypename]/xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.16. JSON Functions and Operators

This section describes:

- functions and operators for processing and creating JSON data
- the SQL/JSON path language
- the SQL/JSON query functions

To provide native support for JSON data types within the SQL environment, PostgreSQL implements the *SQL/JSON data model*. This model comprises sequences of items. Each item can hold SQL scalar

values, with an additional SQL/JSON null value, and composite data structures that use JSON arrays and objects. The model is a formalization of the implied data model in the JSON specification RFC 7159[4].

SQL/JSON allows you to handle JSON data alongside regular SQL data, with transaction support, including:

- Uploading JSON data into the database and storing it in regular SQL columns as character or binary strings.
- Generating JSON objects and arrays from relational data.
- Querying JSON data using SQL/JSON query functions and SQL/JSON path language expressions.

To learn more about the SQL/JSON standard, see [sqltr-19075-6]. For details on JSON types supported in PostgreSQL, see Section 8.14.

9.16.1. Processing and Creating JSON Data

Table 9.45 shows the operators that are available for use with JSON data types (see Section 8.14). In addition, the usual comparison operators shown in Table 9.1 are available for `jsonb`, though not for `json`. The comparison operators follow the ordering rules for B-tree operations outlined in Section 8.14.4. See also Section 9.21 for the aggregate function `json_agg` which aggregates record values as JSON, the aggregate function `json_object_agg` which aggregates pairs of values into a JSON object, and their `jsonb` equivalents, `jsonb_agg` and `jsonb_object_agg`.

Table 9.45. `json` and `jsonb` Operators

Operator **Description** **Example(s)**

`json -> integer → json`
`jsonb -> integer → jsonb`
Extracts *n*'th element of JSON array (array elements are indexed from zero, but negative integers count from the end).
`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> 2 → {"c":"baz"}`
`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> -3 → {"a":"foo"}`

`json -> text → json`
`jsonb -> text → jsonb`
Extracts JSON object field with the given key.
`'{"a": {"b":"foo"}}'::json -> 'a' → {"b":"foo"}`

`json ->> integer → text`
`jsonb ->> integer → text`
Extracts *n*'th element of JSON array, as `text`.
`'[1,2,3]'::json ->> 2 → 3`

`json ->> text → text`
`jsonb ->> text → text`
Extracts JSON object field with the given key, as `text`.
`'{"a":1,"b":2}'::json ->> 'b' → 2`

[4] https://datatracker.ietf.org/doc/html/rfc7159

Operator Description Example(s)

`json #> text[] → json`
`jsonb #> text[] → jsonb`
Extracts JSON sub-object at the specified path, where path elements can be either field keys or array indexes.
`'{"a": {"b": ["foo","bar"]}}'::json #> '{a,b,1}' → "bar"`

`json #>> text[] → text`
`jsonb #>> text[] → text`
Extracts JSON sub-object at the specified path as `text`.
`'{"a": {"b": ["foo","bar"]}}'::json #>> '{a,b,1}' → bar`

> **Note**
>
> The field/element/path extraction operators return NULL, rather than failing, if the JSON input does not have the right structure to match the request; for example if no such key or array element exists.

Some further operators exist only for `jsonb`, as shown in Table 9.46. Section 8.14.4 describes how these operators can be used to effectively search indexed `jsonb` data.

Table 9.46. Additional `jsonb` Operators

Operator Description Example(s)

`jsonb @> jsonb → boolean`
Does the first JSON value contain the second? (See Section 8.14.3 for details about containment.)
`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t`

`jsonb <@ jsonb → boolean`
Is the first JSON value contained in the second?
`'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb → t`

`jsonb ? text → boolean`
Does the text string exist as a top-level key or array element within the JSON value?
`'{"a":1, "b":2}'::jsonb ? 'b' → t`
`'["a", "b", "c"]'::jsonb ? 'b' → t`

`jsonb ?| text[] → boolean`
Do any of the strings in the text array exist as top-level keys or array elements?
`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → t`

`jsonb ?& text[] → boolean`
Do all of the strings in the text array exist as top-level keys or array elements?
`'["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t`

`jsonb || jsonb → jsonb` Concatenates two `jsonb` values. Concatenating two arrays generates an array containing all the elements of each input. Concatenating two objects generates an object containing

Operator Description Example(s)

the union of their keys, taking the second object's value when there are duplicate keys. All other cases are treated by converting a non-array input into a single-element array, and then proceeding as for two arrays. Does not operate recursively: only the top-level array or object structure is merged.
`'["a", "b"]'::jsonb || '["a", "d"]'::jsonb` → `["a", "b", "a", "d"]`
`'{"a": "b"}'::jsonb || '{"c": "d"}'::jsonb` → `{"a": "b", "c": "d"}`
`'[1, 2]'::jsonb || '3'::jsonb` → `[1, 2, 3]`
`'{"a": "b"}'::jsonb || '42'::jsonb` → `[{"a": "b"}, 42]`
To append an array to another array as a single entry, wrap it in an additional layer of array, for example:
`'[1, 2]'::jsonb || jsonb_build_array('[3, 4]'::jsonb)` → `[1, 2, [3, 4]]`

`jsonb - text` → `jsonb`
Deletes a key (and its value) from a JSON object, or matching string value(s) from a JSON array.
`'{"a": "b", "c": "d"}'::jsonb - 'a'` → `{"c": "d"}`
`'["a", "b", "c", "b"]'::jsonb - 'b'` → `["a", "c"]`

`jsonb - text[]` → `jsonb`
Deletes all matching keys or array elements from the left operand.
`'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]` → `{}`

`jsonb - integer` → `jsonb`
Deletes the array element with specified index (negative integers count from the end). Throws an error if JSON value is not an array.
`'["a", "b"]'::jsonb - 1` → `["a"]`

`jsonb #- text[]` → `jsonb`
Deletes the field or array element at the specified path, where path elements can be either field keys or array indexes.
`'["a", {"b":1}]'::jsonb #- '{1,b}'` → `["a", {}]`

`jsonb @? jsonpath` → `boolean`
Does JSON path return any item for the specified JSON value? (This is useful only with SQL-standard JSON path expressions, not predicate check expressions, since those always return a value.)
`'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)'` → `t`

`jsonb @@ jsonpath` → `boolean`
Returns the result of a JSON path predicate check for the specified JSON value. (This is useful only with predicate check expressions, not SQL-standard JSON path expressions, since it will return `NULL` if the path result is not a single boolean value.)
`'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2'` → `t`

Note

The `jsonpath` operators `@?` and `@@` suppress the following errors: missing object field or array element, unexpected JSON item type, datetime and numeric errors. The `jsonpath-`

related functions described below can also be told to suppress these types of errors. This behavior might be helpful when searching JSON document collections of varying structure.

Table 9.47 shows the functions that are available for constructing `json` and `jsonb` values. Some functions in this table have a `RETURNING` clause, which specifies the data type returned. It must be one of `json`, `jsonb`, `bytea`, a character string type (`text`, `char`, or `varchar`), or a type that can be cast to `json`. By default, the `json` type is returned.

Table 9.47. JSON Creation Functions

Function Description Example(s)

`to_json` (`anyelement`) → `json`
`to_jsonb` (`anyelement`) → `jsonb`
Converts any SQL value to `json` or `jsonb`. Arrays and composites are converted recursively to arrays and objects (multidimensional arrays become arrays of arrays in JSON). Otherwise, if there is a cast from the SQL data type to `json`, the cast function will be used to perform the conversion;[a] otherwise, a scalar JSON value is produced. For any scalar other than a number, a Boolean, or a null value, the text representation will be used, with escaping as necessary to make it a valid JSON string value.
`to_json('Fred said "Hi."'::text)` → `"Fred said \"Hi.\""`
`to_jsonb(row(42, 'Fred said "Hi."'::text))` → `{"f1": 42, "f2": "Fred said \"Hi.\""}`

`array_to_json` (`anyarray` [, `boolean`]) → `json`
Converts an SQL array to a JSON array. The behavior is the same as `to_json` except that line feeds will be added between top-level array elements if the optional boolean parameter is true.
`array_to_json('{{1,5},{99,100}}'::int[])` → `[[1,5],[99,100]]`

`json_array` ([{ *`value_expression`* [`FORMAT JSON`] } [, ...]] [{ `NULL` | `ABSENT` } `ON NULL`] [`RETURNING` *`data_type`* [`FORMAT JSON` [`ENCODING UTF8`]]])
`json_array` ([*`query_expression`*] [`RETURNING` *`data_type`* [`FORMAT JSON` [`ENCODING UTF8`]]])
Constructs a JSON array from either a series of *`value_expression`* parameters or from the results of *`query_expression`*, which must be a SELECT query returning a single column. If `ABSENT ON NULL` is specified, NULL values are ignored. This is always the case if a *`query_expression`* is used.
`json_array(1,true,json '{"a":null}')` → `[1, true, {"a":null}]`
`json_array(SELECT * FROM (VALUES(1),(2)) t)` → `[1, 2]`

`row_to_json` (`record` [, `boolean`]) → `json`
Converts an SQL composite value to a JSON object. The behavior is the same as `to_json` except that line feeds will be added between top-level elements if the optional boolean parameter is true.
`row_to_json(row(1,'foo'))` → `{"f1":1,"f2":"foo"}`

`json_build_array` (`VARIADIC "any"`) → `json`
`jsonb_build_array` (`VARIADIC "any"`) → `jsonb`
Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list. Each argument is converted as per `to_json` or `to_jsonb`.
`json_build_array(1, 2, 'foo', 4, 5)` → `[1, 2, "foo", 4, 5]`

`json_build_object` (`VARIADIC "any"`) → `json`

Function **Description** **Example(s)**

`jsonb_build_object ( VARIADIC "any" ) → jsonb`
Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating keys and values. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`.
`json_build_object('foo', 1, 2, row(3,'bar')) → {"foo" : 1, "2" : {"f1":3,"f2":"bar"}}`

`json_object ( [ { key_expression { VALUE | ':' } value_expression [ FORMAT JSON [ ENCODING UTF8 ] ] }[, ...] ] [ { NULL | ABSENT } ON NULL ] [ { WITH | WITHOUT } UNIQUE [ KEYS ] ] [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])`
Constructs a JSON object of all the key/value pairs given, or an empty object if none are given. *key_expression* is a scalar expression defining the JSON key, which is converted to the `text` type. It cannot be `NULL` nor can it belong to a type that has a cast to the `json` type. If `WITH UNIQUE KEYS` is specified, there must not be any duplicate *key_expression*. Any pair for which the *value_expression* evaluates to `NULL` is omitted from the output if `ABSENT ON NULL` is specified; if `NULL ON NULL` is specified or the clause omitted, the key is included with value `NULL`.
`json_object('code' VALUE 'P123', 'title': 'Jaws') → {"code" : "P123", "title" : "Jaws"}`

`json_object ( text[] ) → json`
`jsonb_object ( text[] ) → jsonb`
Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating key/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a key/value pair. All values are converted to JSON strings.
`json_object('{a, 1, b, "def", c, 3.5}') → {"a" : "1", "b" : "def", "c" : "3.5"}`
`json_object('{{a, 1}, {b, "def"}, {c, 3.5}}') → {"a" : "1", "b" : "def", "c" : "3.5"}`

`json_object ( keys text[], values text[] ) → json`
`jsonb_object ( keys text[], values text[] ) → jsonb`
This form of `json_object` takes keys and values pairwise from separate text arrays. Otherwise it is identical to the one-argument form.
`json_object('{a,b}', '{1,2}') → {"a": "1", "b": "2"}`

`json ( expression [ FORMAT JSON [ ENCODING UTF8 ]] [ { WITH | WITHOUT } UNIQUE [ KEYS ]] ) → json`
Converts a given expression specified as `text` or `bytea` string (in UTF8 encoding) into a JSON value. If *expression* is NULL, an SQL null value is returned. If `WITH UNIQUE` is specified, the *expression* must not contain any duplicate object keys.
`json('{"a":123, "b":[true,"foo"], "a":"bar"}') → {"a":123, "b":[true,"foo"], "a":"bar"}`

`json_scalar ( expression )`
Converts a given SQL scalar value into a JSON scalar value. If the input is NULL, an SQL null is returned. If the input is number or a boolean value, a corresponding JSON number or boolean value is returned. For any other value, a JSON string is returned.
`json_scalar(123.45) → 123.45`

Function Description Example(s)
`json_scalar(CURRENT_TIMESTAMP)` → `"2022-05-10T10:51:04.62128-04:00"`

`json_serialize(`*`expression`*`[ FORMAT JSON [ ENCODING UTF8 ]][ RETURNING` *`data_type`*`[ FORMAT JSON [ ENCODING UTF8 ]]])`
Converts an SQL/JSON expression into a character or binary string. The *`expression`* can be of any JSON type, any character string type, or `bytea` in UTF8 encoding. The returned type used in `RETURNING` can be any character string type or `bytea`. The default is `text`.
`json_serialize('{ "a" : 1 } ' RETURNING bytea)` → `\x7b20226122203a2031207d20`

[a] For example, the hstore extension has a cast from `hstore` to `json`, so that `hstore` values converted via the JSON creation functions will be represented as JSON objects, not as primitive string values.

Table 9.48 details SQL/JSON facilities for testing JSON.

Table 9.48. SQL/JSON Testing Functions

Function signature Description Example(s)
`expression` `IS [ NOT ] JSON [ { VALUE | SCALAR | ARRAY | OBJECT } ] [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]` This predicate tests whether *`expression`* can be parsed as JSON, possibly of a specified type. If `SCALAR` or `ARRAY` or `OBJECT` is specified, the test is whether or not the JSON is of that particular type. If `WITH UNIQUE KEYS` is specified, then any object in the *`expression`* is also tested to see if it has duplicate keys. `SELECT js,` `  js IS JSON "json?",` `  js IS JSON SCALAR "scalar?",` `  js IS JSON OBJECT "object?",` `  js IS JSON ARRAY "array?"` `FROM (VALUES` `  ('123'), ('"abc"'), ('{"a": "b"}'), ('[1,2]'),` `('abc')) foo(js);` ` js	json?	scalar?	object?	array?` `------------+-------+---------+---------+--------` ` 123	t	t	f	f` ` "abc"	t	t	f	f` ` {"a": "b"}	t	f	t	f` ` [1,2]	t	f	f	t` ` abc	f	f	f	f`  `SELECT js,` ` js IS JSON OBJECT "object?",` ` js IS JSON ARRAY "array?",` ` js IS JSON ARRAY WITH UNIQUE KEYS "array w. UK?",` ` js IS JSON ARRAY WITHOUT UNIQUE KEYS "array w/o UK?"` `FROM (VALUES ('[{"a":"1"},` ` {"b":"2","b":"3"}]')) foo(js);` `-[RECORD 1]-+--------------------` `js	[{"a":"1"}, +`

Function signature Description Example(s)

`                  |  {"b":"2","b":"3"}]`
`object?       | f`
`array?        | t`
`array w. UK?  | f`
`array w/o UK? | t`

Table 9.49 shows the functions that are available for processing `json` and `jsonb` values.

Table 9.49. JSON Processing Functions

Function Description Example(s)

`json_array_elements(json) → setof json`
`jsonb_array_elements(jsonb) → setof jsonb`
Expands the top-level JSON array into a set of JSON values.
`select * from json_array_elements('[1,true, [2,false]]') →`
 	
`   value`
`-----------`
` 1`
` true`
` [2,false]`

`json_array_elements_text(json) → setof text`
`jsonb_array_elements_text(jsonb) → setof text`
Expands the top-level JSON array into a set of `text` values.
`select * from json_array_elements_text('["foo", "bar"]') →`
 	
`   value`
`-----------`
` foo`
` bar`

`json_array_length(json) → integer`
`jsonb_array_length(jsonb) → integer`
Returns the number of elements in the top-level JSON array.
`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]') → 5`
`jsonb_array_length('[]') → 0`

`json_each(json) → setof record(key text, value json)`
`jsonb_each(jsonb) → setof record(key text, value jsonb)`
Expands the top-level JSON object into a set of key/value pairs.
`select * from json_each('{"a":"foo", "b":"bar"}') →`
 	
` key | value`
`-----+-------`
` a   | "foo"`
` b   | "bar"`

Function
Description
Example(s)

`json_each_text ( json ) → setof record ( key text, value text )`

`jsonb_each_text ( jsonb ) → setof record ( key text, value text )`

Expands the top-level JSON object into a set of key/value pairs. The returned *value*s will be of type `text`.

```
select * from json_each_text('{"a":"foo", "b":"bar"}') →


 key | value
-----+-------
 a   | foo
 b   | bar
```

`json_extract_path ( from_json json, VARIADIC path_elems text[] ) → json`

`jsonb_extract_path ( from_json jsonb, VARIADIC path_elems text[] ) → jsonb`

Extracts JSON sub-object at the specified path. (This is functionally equivalent to the #> operator, but writing the path out as a variadic list can be more convenient in some cases.)

```
json_extract_path('{"f2":{"f3":1},"f4":
{"f5":99,"f6":"foo"}}', 'f4', 'f6') → "foo"
```

`json_extract_path_text ( from_json json, VARIADIC path_elems text[] ) → text`

`jsonb_extract_path_text ( from_json jsonb, VARIADIC path_elems text[] ) → text`

Extracts JSON sub-object at the specified path as `text`. (This is functionally equivalent to the #>> operator.)

```
json_extract_path_text('{"f2":{"f3":1},"f4":
{"f5":99,"f6":"foo"}}', 'f4', 'f6') → foo
```

`json_object_keys ( json ) → setof text`

`jsonb_object_keys ( jsonb ) → setof text`

Returns the set of keys in the top-level JSON object.

```
select * from json_object_keys('{"f1":"abc","f2":{"f3":"a",
"f4":"b"}}') →


 json_object_keys
------------------
 f1
 f2
```

`json_populate_record ( base anyelement, from_json json ) → anyelement`

`jsonb_populate_record ( base anyelement, from_json jsonb ) → anyelement`

Expands the top-level JSON object to a row having the composite type of the *base* argument. The JSON object is scanned for fields whose names match column names of the output row type, and their values are inserted into those columns of the output. (Fields that do not correspond to any output column name are ignored.) In typical use, the value of *base* is just `NULL`, which means that any output columns that do not match any object field will be filled with nulls. However, if *base* isn't `NULL` then the values it contains will be used for unmatched columns.

Function
Description
Example(s)

To convert a JSON value to the SQL type of an output column, the following rules are applied in sequence:

- A JSON null value is converted to an SQL null in all cases.
- If the output column is of type `json` or `jsonb`, the JSON value is just reproduced exactly.
- If the output column is a composite (row) type, and the JSON value is a JSON object, the fields of the object are converted to columns of the output row type by recursive application of these rules.
- Likewise, if the output column is an array type and the JSON value is a JSON array, the elements of the JSON array are converted to elements of the output array by recursive application of these rules.
- Otherwise, if the JSON value is a string, the contents of the string are fed to the input conversion function for the column's data type.
- Otherwise, the ordinary text representation of the JSON value is fed to the input conversion function for the column's data type.

While the example below uses a constant JSON value, typical use would be to reference a `json` or `jsonb` column laterally from another table in the query's `FROM` clause. Writing `json_populate_record` in the `FROM` clause is good practice, since all of the extracted columns are available for use without duplicate function calls.

```
create type subrowtype as (d int, e text);create type myrowtype as (a int, b text[], c subrowtype);
select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}, "x": "foo"}') →


 a |   b       |      c
---+-----------+-------------
 1 | {2,"a b"} | (4,"a b c")
```

`jsonb_populate_record_valid` (*`base`* `anyelement`, *`from_json`* `json`) → `boolean`

Function for testing `jsonb_populate_record`. Returns `true` if the input `jsonb_populate_record` would finish without an error for the given input JSON object; that is, it's valid input, `false` otherwise.

```
create type jsb_char2 as (a char(2));
select jsonb_populate_record_valid(NULL::jsb_char2, '{"a": "aaa"}'); →


 jsonb_populate_record_valid
-----------------------------
 f
(1 row)

select * from jsonb_populate_record(NULL::jsb_char2, '{"a": "aaa"}') q; →


ERROR:  value too long for type character(2)
```

Function
Description
Example(s)

```
select jsonb_populate_record_valid(NULL::jsb_char2, '{"a":
"aa"}'); →


 jsonb_populate_record_valid
-----------------------------
 t
(1 row)

select * from jsonb_populate_record(NULL::jsb_char2, '{"a":
"aa"}') q; →


 a
----
 aa
(1 row)
```

`json_populate_recordset(` *`base`* `anyelement, ` *`from_json`* `json) → setof anyelement`

`jsonb_populate_recordset(` *`base`* `anyelement, ` *`from_json`* `jsonb) → setof anyelement`

Expands the top-level JSON array of objects to a set of rows having the composite type of the *`base`* argument. Each element of the JSON array is processed as described above for `json[b]_populate_record`.

```
create type twoints as (a int, b int);
select * from json_populate_recordset(null::twoints,
'[{"a":1,"b":2}, {"a":3,"b":4}]') →


 a | b
---+---
 1 | 2
 3 | 4
```

`json_to_record(json) → record`

`jsonb_to_record(jsonb) → record`

Expands the top-level JSON object to a row having the composite type defined by an AS clause. (As with all functions returning `record`, the calling query must explicitly define the structure of the record with an AS clause.) The output record is filled from fields of the JSON object, in the same way as described above for `json[b]_populate_record`. Since there is no input record value, unmatched columns are always filled with nulls.

```
create type myrowtype as (a int, b text);
select * from json_to_record('{"a":1,"b":[1,2,3],"c":
[1,2,3],"e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a
int, b text, c int[], d text, r myrowtype) →


 a |    b    |    c    | d |       r
---+---------+---------+---+---------------
 1 | [1,2,3] | {1,2,3} |   | (123,"a b c")
```

`json_to_recordset(json) → setof record`

Function **Description** **Example(s)**

`jsonb_to_recordset` (`jsonb`) → `setof record`
Expands the top-level JSON array of objects to a set of rows having the composite type defined by an AS clause. (As with all functions returning `record`, the calling query must explicitly define the structure of the record with an AS clause.) Each element of the JSON array is processed as described above for `json[b]_populate_record`.
`select * from json_to_recordset('[{"a":1,"b":"foo"}, {"a":"2","c":"bar"}]') as x(a int, b text)` →
`a | b`
`---+-----`
`1 | foo`
`2 |`

`jsonb_set` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean`]) → `jsonb`
Returns *`target`* with the item designated by *`path`* replaced by *`new_value`*, or with *`new_value`* added if *`create_if_missing`* is true (which is the default) and the item designated by *`path`* does not exist. All earlier steps in the path must exist, or the *`target`* is returned unchanged. As with the path oriented operators, negative integers that appear in the *`path`* count from the end of JSON arrays. If the last path step is an array index that is out of range, and *`create_if_missing`* is true, the new value is added at the beginning of the array if the index is negative, or at the end of the array if it is positive.
`jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', '[2,3,4]', false)` → `[{"f1": [2, 3, 4], "f2": null}, 2, null, 3]`
`jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}', '[2,3,4]')` → `[{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]`

`jsonb_set_lax` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean` [, *`null_value_treatment`* `text`]]) → `jsonb`
If *`new_value`* is not `NULL`, behaves identically to `jsonb_set`. Otherwise behaves according to the value of *`null_value_treatment`* which must be one of `'raise_exception'`, `'use_json_null'`, `'delete_key'`, or `'return_target'`. The default is `'use_json_null'`.
`jsonb_set_lax('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', null)` → `[{"f1": null, "f2": null}, 2, null, 3]`
`jsonb_set_lax('[{"f1":99,"f2":null},2]', '{0,f3}', null, true, 'return_target')` → `[{"f1": 99, "f2": null}, 2]`

`jsonb_insert` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`insert_after`* `boolean`]) → `jsonb` Returns *`target`* with *`new_value`* inserted. If the item designated by the *`path`* is an array element, *`new_value`* will be inserted before that item if *`insert_after`* is false (which is the default), or after it if *`insert_after`* is true. If the item designated by the *`path`* is an object field, *`new_value`* will be inserted only if the object does not already contain that key. All earlier steps in the path must exist, or the *`target`* is returned unchanged. As with the path oriented operators, negative integers that appear in the *`path`* count from the end of JSON arrays. If the last path step is an array index that is out of range, the new value is added at the beginning of the array if the index is negative, or at the end of the array if it is positive.

Function
Description
Example(s)

`jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"')` → `{"a": [0, "new_value", 1, 2]}`
`jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)` → `{"a": [0, 1, "new_value", 2]}`

`json_strip_nulls(json)` → `json`

`jsonb_strip_nulls(jsonb)` → `jsonb`

Deletes all object fields that have null values from the given JSON value, recursively. Null values that are not object fields are untouched.

`json_strip_nulls('[{"f1":1, "f2":null}, 2, null, 3]')` → `[{"f1":1},2,null,3]`

`jsonb_path_exists(target jsonb, path jsonpath [, vars jsonb [, silent boolean ]])` → `boolean`

Checks whether the JSON path returns any item for the specified JSON value. (This is useful only with SQL-standard JSON path expressions, not predicate check expressions, since those always return a value.) If the *vars* argument is specified, it must be a JSON object, and its fields provide named values to be substituted into the `jsonpath` expression. If the *silent* argument is specified and is `true`, the function suppresses the same errors as the `@?` and `@@` operators do.

`jsonb_path_exists('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `t`

`jsonb_path_match(target jsonb, path jsonpath [, vars jsonb [, silent boolean ]])` → `boolean`

Returns the SQL boolean result of a JSON path predicate check for the specified JSON value. (This is useful only with predicate check expressions, not SQL-standard JSON path expressions, since it will either fail or return `NULL` if the path result is not a single boolean value.) The optional *vars* and *silent* arguments act the same as for `jsonb_path_exists`.

`jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ >= $min && @ <= $max))', '{"min":2, "max":4}')` → `t`

`jsonb_path_query(target jsonb, path jsonpath [, vars jsonb [, silent boolean ]])` → `setof jsonb`

Returns all JSON items returned by the JSON path for the specified JSON value. For SQL-standard JSON path expressions it returns the JSON values selected from *target*. For predicate check expressions it returns the result of the predicate check: `true`, `false`, or `null`. The optional *vars* and *silent* arguments act the same as for `jsonb_path_exists`.

```
select * from jsonb_path_query('{"a":[1,2,3,4,5]}',
'$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')
→

 jsonb_path_query
------------------
 2
 3
 4
```

`jsonb_path_query_array(target jsonb, path jsonpath [, vars jsonb [, silent boolean ]])` → `jsonb`

Function
Description
Example(s)

Returns all JSON items returned by the JSON path for the specified JSON value, as a JSON array. The parameters are the same as for `jsonb_path_query`.
`jsonb_path_query_array('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `[2, 3, 4]`

`jsonb_path_query_first` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `jsonb`
Returns the first JSON item returned by the JSON path for the specified JSON value, or `NULL` if there are no results. The parameters are the same as for `jsonb_path_query`.
`jsonb_path_query_first('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `2`

`jsonb_path_exists_tz` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `boolean`
`jsonb_path_match_tz` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `boolean`
`jsonb_path_query_tz` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `setof jsonb`
`jsonb_path_query_array_tz` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `jsonb`
`jsonb_path_query_first_tz` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `jsonb`
These functions act like their counterparts described above without the `_tz` suffix, except that these functions support comparisons of date/time values that require time-zone-aware conversions. The example below requires interpretation of the date-only value `2015-08-02` as a timestamp with time zone, so the result depends on the current TimeZone setting. Due to this dependency, these functions are marked as stable, which means these functions cannot be used in indexes. Their counterparts are immutable, and so can be used in indexes; but they will throw errors if asked to make such comparisons.
`jsonb_path_exists_tz('["2015-08-01 12:00:00-05"]', '$[*] ? (@.datetime() < "2015-08-02".datetime())')` → `t`

`jsonb_pretty` (`jsonb`) → `text`
Converts the given JSON value to pretty-printed, indented text.
`jsonb_pretty('[{"f1":1,"f2":null}, 2]')` →

```
[
    {
        "f1": 1,
        "f2": null
    },
    2
]
```

`json_typeof` (`json`) → `text`
`jsonb_typeof` (`jsonb`) → `text`
Returns the type of the top-level JSON value as a text string. Possible types are `object`, `array`, `string`, `number`, `boolean`, and `null`. (The `null` result should not be confused with an SQL NULL; see the examples.)
`json_typeof('-123.4')` → `number`
`json_typeof('null'::json)` → `null`

Function Description Example(s)
`json_typeof(NULL::json) IS NULL → t`

9.16.2. The SQL/JSON Path Language

SQL/JSON path expressions specify item(s) to be retrieved from a JSON value, similarly to XPath expressions used for access to XML content. In PostgreSQL, path expressions are implemented as the `jsonpath` data type and can use any elements described in Section 8.14.7.

JSON query functions and operators pass the provided path expression to the *path engine* for evaluation. If the expression matches the queried JSON data, the corresponding JSON item, or set of items, is returned. If there is no match, the result will be `NULL`, `false`, or an error, depending on the function. Path expressions are written in the SQL/JSON path language and can include arithmetic expressions and functions.

A path expression consists of a sequence of elements allowed by the `jsonpath` data type. The path expression is normally evaluated from left to right, but you can use parentheses to change the order of operations. If the evaluation is successful, a sequence of JSON items is produced, and the evaluation result is returned to the JSON query function that completes the specified computation.

To refer to the JSON value being queried (the *context item*), use the $ variable in the path expression. The first element of a path must always be $. It can be followed by one or more accessor operators, which go down the JSON structure level by level to retrieve sub-items of the context item. Each accessor operator acts on the result(s) of the previous evaluation step, producing zero, one, or more output items from each input item.

For example, suppose you have some JSON data from a GPS tracker that you would like to parse, such as:

```
SELECT '{
  "track": {
    "segments": [
      {
        "location":   [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      },
      {
        "location":   [ 47.706, 13.2635 ],
        "start time": "2018-10-14 10:39:21",
        "HR": 135
      }
    ]
  }
}' AS json \gset
```

(The above example can be copied-and-pasted into psql to set things up for the following examples. Then psql will expand `:'json'` into a suitably-quoted string constant containing the JSON value.)

To retrieve the available track segments, you need to use the *`.key`* accessor operator to descend through surrounding JSON objects, for example:

```
=> select jsonb_path_query(:'json', '$.track.segments');

      jsonb_path_query
```

```
------------------------------------------------------------
------------------------------------------------------------
-------------------------------------------
 [{"HR": 73, "location": [47.763, 13.4034], "start time":
 "2018-10-14 10:05:14"}, {"HR": 135, "location": [47.706, 13.2635],
 "start time": "2018-10-14 10:39:21"}]
```

To retrieve the contents of an array, you typically use the `[*]` operator. The following example will return the location coordinates for all the available track segments:

```
=> select jsonb_path_query(:'json',
 '$.track.segments[*].location');
 jsonb_path_query
------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

Here we started with the whole JSON input value (`$`), then the `.track` accessor selected the JSON object associated with the `"track"` object key, then the `.segments` accessor selected the JSON array associated with the `"segments"` key within that object, then the `[*]` accessor selected each element of that array (producing a series of items), then the `.location` accessor selected the JSON array associated with the `"location"` key within each of those objects. In this example, each of those objects had a `"location"` key; but if any of them did not, the `.location` accessor would have simply produced no output for that input item.

To return the coordinates of the first segment only, you can specify the corresponding subscript in the `[]` accessor operator. Recall that JSON array indexes are 0-relative:

```
=> select jsonb_path_query(:'json',
 '$.track.segments[0].location');
 jsonb_path_query
------------------
 [47.763, 13.4034]
```

The result of each path evaluation step can be processed by one or more of the `jsonpath` operators and methods listed in Section 9.16.2.3. Each method name must be preceded by a dot. For example, you can get the size of an array:

```
=> select jsonb_path_query(:'json', '$.track.segments.size()');
 jsonb_path_query
------------------
 2
```

More examples of using `jsonpath` operators and methods within path expressions appear below in Section 9.16.2.3.

A path can also contain *filter expressions* that work similarly to the `WHERE` clause in SQL. A filter expression begins with a question mark and provides a condition in parentheses:

```
? (condition)
```

Filter expressions must be written just after the path evaluation step to which they should apply. The result of that step is filtered to include only those items that satisfy the provided condition. SQL/JSON defines three-valued logic, so the condition can produce `true`, `false`, or `unknown`. The `unknown` value plays the same role as SQL `NULL` and can be tested for with the `is unknown` predicate. Further path evaluation steps use only those items for which the filter expression returned `true`.

The functions and operators that can be used in filter expressions are listed in Table 9.51. Within a filter expression, the `@` variable denotes the value being considered (i.e., one result of the preceding path step). You can write accessor operators after `@` to retrieve component items.

For example, suppose you would like to retrieve all heart rate values higher than 130. You can achieve this as follows:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].HR ? (@ >
 130)');
 jsonb_path_query
------------------
 135
```

To get the start times of segments with such values, you have to filter out irrelevant segments before selecting the start times, so the filter expression is applied to the previous step, and the path used in the condition is different:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.HR >
 130)."start time"');
   jsonb_path_query
-----------------------
 "2018-10-14 10:39:21"
```

You can use several filter expressions in sequence, if required. The following example selects start times of all segments that contain locations with relevant coordinates and high heart rate values:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ?
 (@.location[1] < 13.4) ? (@.HR > 130)."start time"');
   jsonb_path_query
-----------------------
 "2018-10-14 10:39:21"
```

Using filter expressions at different nesting levels is also allowed. The following example first filters all segments by location, and then returns high heart rate values for these segments, if available:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ?
 (@.location[1] < 13.4).HR ? (@ > 130)');
 jsonb_path_query
------------------
 135
```

You can also nest filter expressions within each other. This example returns the size of the track if it contains any segments with high heart rate values, or an empty sequence otherwise:

```
=> select jsonb_path_query(:'json', '$.track ?
 (exists(@.segments[*] ? (@.HR > 130))).segments.size()');
 jsonb_path_query
------------------
 2
```

9.16.2.1. Deviations from the SQL Standard

PostgreSQL's implementation of the SQL/JSON path language has the following deviations from the SQL/JSON standard.

9.16.2.1.1. Boolean Predicate Check Expressions

As an extension to the SQL standard, a PostgreSQL path expression can be a Boolean predicate, whereas the SQL standard allows predicates only within filters. While SQL-standard path expressions return the relevant element(s) of the queried JSON value, predicate check expressions return the single three-valued `jsonb` result of the predicate: `true`, `false`, or `null`. For example, we could write this SQL-standard filter expression:

```
=> select jsonb_path_query(:'json', '$.track.segments ?(@[*].HR >
 130)');
                                  jsonb_path_query
-----------------------------------------------------------
----------------------
 {"HR": 135, "location": [47.706, 13.2635], "start time":
 "2018-10-14 10:39:21"}
```

The similar predicate check expression simply returns `true`, indicating that a match exists:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].HR >
 130');
 jsonb_path_query
------------------
 true
```

> **Note**
>
> Predicate check expressions are required in the `@@` operator (and the `jsonb_path_match` function), and should not be used with the `@?` operator (or the `jsonb_path_exists` function).

9.16.2.1.2. Regular Expression Interpretation

There are minor differences in the interpretation of regular expression patterns used in `like_regex` filters, as described in Section 9.16.2.4.

9.16.2.2. Strict and Lax Modes

When you query JSON data, the path expression may not match the actual JSON data structure. An attempt to access a non-existent member of an object or element of an array is defined as a structural error. SQL/JSON path expressions have two modes of handling structural errors:

- lax (default) — the path engine implicitly adapts the queried data to the specified path. Any structural errors that cannot be fixed as described below are suppressed, producing no match.
- strict — if a structural error occurs, an error is raised.

Lax mode facilitates matching of a JSON document and path expression when the JSON data does not conform to the expected schema. If an operand does not match the requirements of a particular operation, it can be automatically wrapped as an SQL/JSON array, or unwrapped by converting its elements into an SQL/JSON sequence before performing the operation. Also, comparison operators automatically unwrap their operands in lax mode, so you can compare SQL/JSON arrays out-of-the-box. An array of size 1 is considered equal to its sole element. Automatic unwrapping is not performed when:

- The path expression contains `type()` or `size()` methods that return the type and the number of elements in the array, respectively.

- The queried JSON data contain nested arrays. In this case, only the outermost array is unwrapped, while all the inner arrays remain unchanged. Thus, implicit unwrapping can only go one level down within each path evaluation step.

For example, when querying the GPS data listed above, you can abstract from the fact that it stores an array of segments when using lax mode:

```
=> select jsonb_path_query(:'json', 'lax
 $.track.segments.location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

In strict mode, the specified path must exactly match the structure of the queried JSON document, so using this path expression will cause an error:

```
=> select jsonb_path_query(:'json', 'strict
 $.track.segments.location');
ERROR:  jsonpath member accessor can only be applied to an object
```

To get the same result as in lax mode, you have to explicitly unwrap the `segments` array:

```
=> select jsonb_path_query(:'json', 'strict
 $.track.segments[*].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

The unwrapping behavior of lax mode can lead to surprising results. For instance, the following query using the `.**` accessor selects every `HR` value twice:

```
=> select jsonb_path_query(:'json', 'lax $.**.HR');
 jsonb_path_query
------------------
 73
 135
 73
 135
```

This happens because the `.**` accessor selects both the `segments` array and each of its elements, while the `.HR` accessor automatically unwraps arrays when using lax mode. To avoid surprising results, we recommend using the `.**` accessor only in strict mode. The following query selects each `HR` value just once:

```
=> select jsonb_path_query(:'json', 'strict $.**.HR');
 jsonb_path_query
------------------
 73
 135
```

The unwrapping of arrays can also lead to unexpected results. Consider this example, which selects all the `location` arrays:

```
=> select jsonb_path_query(:'json', 'lax
 $.track.segments[*].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
(2 rows)
```

As expected it returns the full arrays. But applying a filter expression causes the arrays to be unwrapped to evaluate each item, returning only the items that match the expression:

```
=> select jsonb_path_query(:'json', 'lax
 $.track.segments[*].location ?(@[*] > 15)');
 jsonb_path_query
------------------
 47.763
 47.706
(2 rows)
```

This despite the fact that the full arrays are selected by the path expression. Use strict mode to restore selecting the arrays:

```
=> select jsonb_path_query(:'json', 'strict
 $.track.segments[*].location ?(@[*] > 15)');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
(2 rows)
```

9.16.2.3. SQL/JSON Path Operators and Methods

Table 9.50 shows the operators and methods available in `jsonpath`. Note that while the unary operators and methods can be applied to multiple values resulting from a preceding path step, the binary operators (addition etc.) can only be applied to single values. In lax mode, methods applied to an array will be executed for each value in the array. The exceptions are `.type()` and `.size()`, which apply to the array itself.

Table 9.50. `jsonpath` Operators and Methods

Operator/Method **Description** **Example(s)**
number + *number* → *number* Addition `jsonb_path_query('[2]', '$[0] + 3')` → `5`
+ *number* → *number* Unary plus (no operation); unlike addition, this can iterate over multiple values `jsonb_path_query_array('{"x": [2,3,4]}', '+ $.x')` → `[2, 3, 4]`
number - *number* → *number* Subtraction `jsonb_path_query('[2]', '7 - $[0]')` → `5`
- *number* → *number*

Operator/Method **Description** **Example(s)**
Negation; unlike subtraction, this can iterate over multiple values `jsonb_path_query_array('{"x": [2,3,4]}', '- $.x')` → `[-2, -3, -4]`
number `*` *number* → *number* Multiplication `jsonb_path_query('[4]', '2 * $[0]')` → `8`
number `/` *number* → *number* Division `jsonb_path_query('[8.5]', '$[0] / 2')` → `4.2500000000000000`
number `%` *number* → *number* Modulo (remainder) `jsonb_path_query('[32]', '$[0] % 10')` → `2`
value `.type()` → *string* Type of the JSON item (see `json_typeof`) `jsonb_path_query_array('[1, "2", {}]', '$[*].type()')` → `["number", "string", "object"]`
value `.size()` → *number* Size of the JSON item (number of array elements, or 1 if not an array) `jsonb_path_query('{"m": [11, 15]}', '$.m.size()')` → `2`
value `.boolean()` → *boolean* Boolean value converted from a JSON boolean, number, or string `jsonb_path_query_array('[1, "yes", false]', '$[*].boolean()')` → `[true, true, false]`

value `.string()` → *string*
String value converted from a JSON boolean, number, string, or datetime
`jsonb_path_query_array('[1.23, "xyz", false]', '$[*].string()')` → `["1.23", "xyz", "false"]`
`jsonb_path_query('"2023-08-15 12:34:56"', '$.timestamp().string()')` → `"2023-08-15T12:34:56"`

value `.double()` → *number* Approximate floating-point number converted from a JSON number or string `jsonb_path_query('{"len": "1.9"}', '$.len.double() * 2')` → `3.8`
number `.ceiling()` → *number* Nearest integer greater than or equal to the given number `jsonb_path_query('{"h": 1.3}', '$.h.ceiling()')` → `2`
number `.floor()` → *number* Nearest integer less than or equal to the given number `jsonb_path_query('{"h": 1.7}', '$.h.floor()')` → `1`
number `.abs()` → *number* Absolute value of the given number `jsonb_path_query('{"z": -0.3}', '$.z.abs()')` → `0.3`

Operator/Method **Description** **Example(s)**
value . bigint() → *bigint* Big integer value converted from a JSON number or string jsonb_path_query('{"len": "9876543219"}', '$.len.bigint()') → 9876543219
value . decimal([*precision* [, *scale*]]) → *decimal* Rounded decimal value converted from a JSON number or string (precision and scale must be integer values) jsonb_path_query('1234.5678', '$.decimal(6, 2)') → 1234.57
value . integer() → *integer* Integer value converted from a JSON number or string jsonb_path_query('{"len": "12345"}', '$.len.integer()') → 12345
value . number() → *numeric* Numeric value converted from a JSON number or string jsonb_path_query('{"len": "123.45"}', '$.len.number()') → 123.45
string . datetime() → *datetime_type* (see note) Date/time value converted from a string jsonb_path_query('["2015-8-1", "2015-08-12"]', '$[*] ? (@.datetime() < "2015-08-2".datetime())') → "2015-8-1"
string . datetime(*template*) → *datetime_type* (see note) Date/time value converted from a string using the specified to_timestamp template jsonb_path_query_array('["12:30", "18:40"]', '$[*].datetime("HH24:MI")') → ["12:30:00", "18:40:00"]
string . date() → *date* Date value converted from a string jsonb_path_query('"2023-08-15"', '$.date()') → "2023-08-15"
string . time() → *time without time zone* Time without time zone value converted from a string jsonb_path_query('"12:34:56"', '$.time()') → "12:34:56"
string . time(*precision*) → *time without time zone* Time without time zone value converted from a string, with fractional seconds adjusted to the given precision jsonb_path_query('"12:34:56.789"', '$.time(2)') → "12:34:56.79"
string . time_tz() → *time with time zone* Time with time zone value converted from a string jsonb_path_query('"12:34:56 +05:30"', '$.time_tz()') → "12:34:56+05:30"
string . time_tz(*precision*) → *time with time zone* Time with time zone value converted from a string, with fractional seconds adjusted to the given precision

Operator/Method **Description** **Example(s)**
`jsonb_path_query('"12:34:56.789 +05:30"', '$.time_tz(2)')` → `"12:34:56.79+05:30"`
`string` `.timestamp()` → *`timestamp without time zone`* Timestamp without time zone value converted from a string `jsonb_path_query('"2023-08-15 12:34:56"', '$.timestamp()')` → `"2023-08-15T12:34:56"`
`string` `.timestamp(`*`precision`*`)` → *`timestamp without time zone`* Timestamp without time zone value converted from a string, with fractional seconds adjusted to the given precision `jsonb_path_query('"2023-08-15 12:34:56.789"', '$.timestamp(2)')` → `"2023-08-15T12:34:56.79"`
`string` `.timestamp_tz()` → *`timestamp with time zone`* Timestamp with time zone value converted from a string `jsonb_path_query('"2023-08-15 12:34:56 +05:30"', '$.timestamp_tz()')` → `"2023-08-15T12:34:56+05:30"`
`string` `.timestamp_tz(`*`precision`*`)` → *`timestamp with time zone`* Timestamp with time zone value converted from a string, with fractional seconds adjusted to the given precision `jsonb_path_query('"2023-08-15 12:34:56.789 +05:30"', '$.timestamp_tz(2)')` → `"2023-08-15T12:34:56.79+05:30"`
`object` `.keyvalue()` → *`array`* The object's key-value pairs, represented as an array of objects containing three fields: `"key"`, `"value"`, and `"id"`; `"id"` is a unique identifier of the object the key-value pair belongs to `jsonb_path_query_array('{"x": "20", "y": 32}', '$.keyvalue()')` → `[{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]`

Note

The result type of the `datetime()` and `datetime(`*`template`*`)` methods can be `date`, `timetz`, `time`, `timestamptz`, or `timestamp`. Both methods determine their result type dynamically.

The `datetime()` method sequentially tries to match its input string to the ISO formats for `date`, `timetz`, `time`, `timestamptz`, and `timestamp`. It stops on the first matching format and emits the corresponding data type.

The `datetime(`*`template`*`)` method determines the result type according to the fields used in the provided template string.

The `datetime()` and `datetime(`*`template`*`)` methods use the same parsing rules as the `to_timestamp` SQL function does (see Section 9.8), with three exceptions. First, these methods don't allow unmatched template patterns. Second, only the following separators are allowed in the template string: minus sign, period, solidus (slash), comma, apostrophe, semicolon, colon and space. Third, separators in the template string must exactly match the input string.

If different date/time types need to be compared, an implicit cast is applied. A `date` value can be cast to `timestamp` or `timestamptz`, `timestamp` can be cast to `timestamptz`,

and `time` to `timetz`. However, all but the first of these conversions depend on the current TimeZone setting, and thus can only be performed within timezone-aware `jsonpath` functions. Similarly, other date/time-related methods that convert strings to date/time types also do this casting, which may involve the current TimeZone setting. Therefore, these conversions can also only be performed within timezone-aware `jsonpath` functions.

Table 9.51 shows the available filter expression elements.

Table 9.51. `jsonpath` Filter Expression Elements

Predicate/Value **Description** **Example(s)**

value == *value* → `boolean`
Equality comparison (this, and the other comparison operators, work on all JSON scalar values)
`jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == 1)')` → `[1, 1]`
`jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == "a")')` → `["a"]`

value != *value* → `boolean`
value <> *value* → `boolean`
Non-equality comparison
`jsonb_path_query_array('[1, 2, 1, 3]', '$[*] ? (@ != 1)')` → `[2, 3]`
`jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <> "b")')` → `["a", "c"]`

value < *value* → `boolean` Less-than comparison `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ < 2)')` → `[1]`
value <= *value* → `boolean` Less-than-or-equal-to comparison `jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <= "b")')` → `["a", "b"]`
value > *value* → `boolean` Greater-than comparison `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ > 2)')` → `[3]`
value >= *value* → `boolean` Greater-than-or-equal-to comparison `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ >= 2)')` → `[2, 3]`

`true` → `boolean`
JSON constant `true`
`jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == true)')` → `{"name": "Chris", "parent": true}`

`false` → `boolean` JSON constant `false`

Predicate/Value **Description** **Example(s)**
`jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == false)')` → `{"name": "John", "parent": false}`
`null` → *value* JSON constant `null` (note that, unlike in SQL, comparison to `null` works normally) `jsonb_path_query('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '$[*] ? (@.job == null) .name')` → `"Mary"`
boolean `&&` *boolean* → `boolean` Boolean AND `jsonb_path_query('[1, 3, 7]', '$[*] ? (@ > 1 && @ < 5)')` → `3`
boolean `||` *boolean* → `boolean` Boolean OR `jsonb_path_query('[1, 3, 7]', '$[*] ? (@ < 1 || @ > 5)')` → `7`

`!` *boolean* → `boolean`
Boolean NOT
`jsonb_path_query('[1, 3, 7]', '$[*] ? (!(@ < 5))')` → `7`

boolean `is unknown` → `boolean` Tests whether a Boolean condition is `unknown`. `jsonb_path_query('[-1, 2, 7, "foo"]', '$[*] ? ((@ > 0) is unknown)')` → `"foo"`

string `like_regex` *string* [`flag` *string*] → `boolean`
Tests whether the first operand matches the regular expression given by the second operand, optionally with modifications described by a string of `flag` characters (see Section 9.16.2.4).
`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c")')` → `["abc", "abdacb"]`
`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c" flag "i")')` → `["abc", "aBdC", "abdacb"]`

string `starts with` *string* → `boolean` Tests whether the second operand is an initial substring of the first operand. `jsonb_path_query('["John Smith", "Mary Stone", "Bob Johnson"]', '$[*] ? (@ starts with "John")')` → `"John Smith"`

`exists (` *path_expression* `)` → `boolean`
Tests whether a path expression matches at least one SQL/JSON item. Returns `unknown` if the path expression would result in an error; the second example uses this to avoid a no-such-key error in strict mode.
`jsonb_path_query('{"x": [1, 2], "y": [2, 4]}', 'strict $.* ? (exists (@ ? (@[*] > 2)))')` → `[2, 4]`
`jsonb_path_query_array('{"value": 41}', 'strict $ ? (exists (@.name)) .name')` → `[]`

9.16.2.4. SQL/JSON Regular Expressions

SQL/JSON path expressions allow matching text to a regular expression with the `like_regex` filter. For example, the following SQL/JSON path query would case-insensitively match all strings in an array that start with an English vowel:

```
$[*] ? (@ like_regex "^[aeiou]" flag "i")
```

The optional `flag` string may include one or more of the characters `i` for case-insensitive match, `m` to allow `^` and `$` to match at newlines, `s` to allow `.` to match a newline, and `q` to quote the whole pattern (reducing the behavior to a simple substring match).

The SQL/JSON standard borrows its definition for regular expressions from the `LIKE_REGEX` operator, which in turn uses the XQuery standard. PostgreSQL does not currently support the `LIKE_REGEX` operator. Therefore, the `like_regex` filter is implemented using the POSIX regular expression engine described in Section 9.7.3. This leads to various minor discrepancies from standard SQL/JSON behavior, which are cataloged in Section 9.7.3.8. Note, however, that the flag-letter incompatibilities described there do not apply to SQL/JSON, as it translates the XQuery flag letters to match what the POSIX engine expects.

Keep in mind that the pattern argument of `like_regex` is a JSON path string literal, written according to the rules given in Section 8.14.7. This means in particular that any backslashes you want to use in the regular expression must be doubled. For example, to match string values of the root document that contain only digits:

```
$.* ? (@ like_regex "^\\d+$")
```

9.16.3. SQL/JSON Query Functions

SQL/JSON functions `JSON_EXISTS()`, `JSON_QUERY()`, and `JSON_VALUE()` described in Table 9.52 can be used to query JSON documents. Each of these functions apply a *`path_expression`* (an SQL/JSON path query) to a *`context_item`* (the document). See Section 9.16.2 for more details on what the *`path_expression`* can contain. The *`path_expression`* can also reference variables, whose values are specified with their respective names in the `PASSING` clause that is supported by each function. *`context_item`* can be a `jsonb` value or a character string that can be successfully cast to `jsonb`.

Table 9.52. SQL/JSON Query Functions

Function signature **Description** **Example(s)**
`JSON_EXISTS ( context_item, path_expression [ PASSING { value AS varname } [, ...]] [{ TRUE | FALSE | UNKNOWN | ERROR } ON ERROR ]) → boolean` • Returns true if the SQL/JSON *`path_expression`* applied to the *`context_item`* yields any items, false otherwise. • The `ON ERROR` clause specifies the behavior if an error occurs during *`path_expression`* evaluation. Specifying `ERROR` will cause an error to be thrown with the appropriate message. Other options include returning `boolean` values `FALSE` or `TRUE` or the value `UNKNOWN` which is actually an SQL NULL. The default when no `ON ERROR` clause is specified is to return the `boolean` value `FALSE`. Examples:

Function signature
Description
Example(s)

```
JSON_EXISTS(jsonb '{"key1": [1,2,3]}', 'strict $.key1[*] ?
(@ > $x)' PASSING 2 AS x) → t
JSON_EXISTS(jsonb '{"a": [1,2,3]}', 'lax $.a[5]' ERROR ON
ERROR) → f
JSON_EXISTS(jsonb '{"a": [1,2,3]}', 'strict $.a[5]' ERROR
ON ERROR) →


ERROR:  jsonpath array subscript is out of bounds
```

```
JSON_QUERY (
    context_item, path_expression
    [ PASSING { value AS varname } [, ...]]
    [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ]
    [ { WITHOUT | WITH { CONDITIONAL | [UNCONDITIONAL] } }
     [ ARRAY ] WRAPPER ]
    [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
    [ { ERROR | NULL | EMPTY { [ ARRAY ] | OBJECT }
     | DEFAULT expression } ON EMPTY ]
    [ { ERROR | NULL | EMPTY { [ ARRAY ] | OBJECT }
     | DEFAULT expression } ON ERROR ]) → jsonb
```

- Returns the result of applying the SQL/JSON *path_expression* to the *context_item*.
- By default, the result is returned as a value of type `jsonb`, though the `RETURNING` clause can be used to return as some other type to which it can be successfully coerced.
- If the path expression may return multiple values, it might be necessary to wrap those values using the `WITH WRAPPER` clause to make it a valid JSON string, because the default behavior is to not wrap them, as if `WITHOUT WRAPPER` were specified. The `WITH WRAPPER` clause is by default taken to mean `WITH UNCONDITIONAL WRAPPER`, which means that even a single result value will be wrapped. To apply the wrapper only when multiple values are present, specify `WITH CONDITIONAL WRAPPER`. Getting multiple values in result will be treated as an error if `WITHOUT WRAPPER` is specified.
- If the result is a scalar string, by default, the returned value will be surrounded by quotes, making it a valid JSON value. It can be made explicit by specifying `KEEP QUOTES`. Conversely, quotes can be omitted by specifying `OMIT QUOTES`. To ensure that the result is a valid JSON value, `OMIT QUOTES` cannot be specified when `WITH WRAPPER` is also specified.
- The `ON EMPTY` clause specifies the behavior if evaluating *path_expression* yields an empty set. The `ON ERROR` clause specifies the behavior if an error occurs when evaluating *path_expression*, when coercing the result value to the `RETURNING` type, or when evaluating the `ON EMPTY` expression if the *path_expression* evaluation returns an empty set.
- For both `ON EMPTY` and `ON ERROR`, specifying `ERROR` will cause an error to be thrown with the appropriate message. Other options include returning an SQL NULL, an empty array (`EMPTY [ARRAY]`), an empty object (`EMPTY OBJECT`), or a user-specified expression (`DEFAULT` *expression*) that can be coerced to jsonb or the type specified in `RETURNING`. The default when `ON EMPTY` or `ON ERROR` is not specified is to return an SQL NULL value.

  Examples:

Function signature
Description
Example(s)

```
JSON_QUERY(jsonb '[1,[2,3],null]', 'lax $[*][$off]' PASSING
1 AS off WITH CONDITIONAL WRAPPER) → 3
JSON_QUERY(jsonb '{"a": "[1, 2]"}', 'lax $.a' OMIT QUOTES)
→ [1, 2]
JSON_QUERY(jsonb '{"a": "[1, 2]"}', 'lax $.a' RETURNING
int[] OMIT QUOTES ERROR ON ERROR) →


ERROR:  malformed array literal: "[1, 2]"
DETAIL:  Missing "]" after array dimensions.
```

```
JSON_VALUE (
    context_item, path_expression
    [ PASSING { value AS varname } [, ...]]
    [ RETURNING data_type ]
    [ { ERROR | NULL | DEFAULT expression } ON EMPTY ]
    [ { ERROR | NULL | DEFAULT expression } ON ERROR ]) → text
```

- Returns the result of applying the SQL/JSON *path_expression* to the *context_item*.
- Only use `JSON_VALUE()` if the extracted value is expected to be a single SQL/JSON scalar item; getting multiple values will be treated as an error. If you expect that extracted value might be an object or an array, use the `JSON_QUERY` function instead.
- By default, the result, which must be a single scalar value, is returned as a value of type `text`, though the `RETURNING` clause can be used to return as some other type to which it can be successfully coerced.
- The `ON ERROR` and `ON EMPTY` clauses have similar semantics as mentioned in the description of `JSON_QUERY`, except the set of values returned in lieu of throwing an error is different.
- Note that scalar strings returned by `JSON_VALUE` always have their quotes removed, equivalent to specifying `OMIT QUOTES` in `JSON_QUERY`.

  Examples:

```
JSON_VALUE(jsonb '"123.45"', '$' RETURNING float) → 123.45
JSON_VALUE(jsonb '"03:04 2015-02-01"', '$.datetime("H-
H24:MI YYYY-MM-DD")' RETURNING date) → 2015-02-01
JSON_VALUE(jsonb '[1,2]', 'strict $[$off]' PASSING 1 as
off) → 2
JSON_VALUE(jsonb '[1,2]', 'strict $[*]' DEFAULT 9 ON ERROR)
→ 9
```

> **Note**
>
> The *context_item* expression is converted to `jsonb` by an implicit cast if the expression is not already of type `jsonb`. Note, however, that any parsing errors that occur during that conversion are thrown unconditionally, that is, are not handled according to the (specified or implicit) `ON ERROR` clause.

Note

`JSON_VALUE()` returns an SQL NULL if *`path_expression`* returns a JSON `null`, whereas `JSON_QUERY()` returns the JSON `null` as is.

9.16.4. JSON_TABLE

`JSON_TABLE` is an SQL/JSON function which queries JSON data and presents the results as a relational view, which can be accessed as a regular SQL table. You can use `JSON_TABLE` inside the `FROM` clause of a `SELECT`, `UPDATE`, or `DELETE` and as data source in a `MERGE` statement.

Taking JSON data as input, `JSON_TABLE` uses a JSON path expression to extract a part of the provided data to use as a *row pattern* for the constructed view. Each SQL/JSON value given by the row pattern serves as source for a separate row in the constructed view.

To split the row pattern into columns, `JSON_TABLE` provides the `COLUMNS` clause that defines the schema of the created view. For each column, a separate JSON path expression can be specified to be evaluated against the row pattern to get an SQL/JSON value that will become the value for the specified column in a given output row.

JSON data stored at a nested level of the row pattern can be extracted using the `NESTED PATH` clause. Each `NESTED PATH` clause can be used to generate one or more columns using the data from a nested level of the row pattern. Those columns can be specified using a `COLUMNS` clause that looks similar to the top-level COLUMNS clause. Rows constructed from NESTED COLUMNS are called *child rows* and are joined against the row constructed from the columns specified in the parent `COLUMNS` clause to get the row in the final view. Child columns themselves may contain a `NESTED PATH` specification thus allowing to extract data located at arbitrary nesting levels. Columns produced by multiple `NESTED PATH`s at the same level are considered to be *siblings* of each other and their rows after joining with the parent row are combined using UNION.

The rows produced by `JSON_TABLE` are laterally joined to the row that generated them, so you do not have to explicitly join the constructed view with the original table holding JSON data.

The syntax is:

```
JSON_TABLE (
    context_item, path_expression [ AS json_path_name ] [ PASSING
 { value AS varname } [, ...] ]
    COLUMNS ( json_table_column [, ...] )
    [ { ERROR | EMPTY [ARRAY]} ON ERROR ]
)


where json_table_column is:

  name FOR ORDINALITY
  | name type
        [ FORMAT JSON [ENCODING UTF8]]
        [ PATH path_expression ]
        [ { WITHOUT | WITH { CONDITIONAL | [UNCONDITIONAL] } }
 [ ARRAY ] WRAPPER ]
        [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
        [ { ERROR | NULL | EMPTY { [ARRAY] | OBJECT } |
 DEFAULT expression } ON EMPTY ]
        [ { ERROR | NULL | EMPTY { [ARRAY] | OBJECT } |
 DEFAULT expression } ON ERROR ]
```

```
  | name type EXISTS [ PATH path_expression ]
        [ { ERROR | TRUE | FALSE | UNKNOWN } ON ERROR ]
  | NESTED [ PATH ] path_expression [ AS json_path_name ] COLUMNS
 ( json_table_column [, ...] )
```

Each syntax element is described below in more detail.

context_item, *path_expression* [AS *json_path_name*] [PASSING { *value* AS *varname* } [, ...]]

The *context_item* specifies the input document to query, the *path_expression* is an SQL/JSON path expression defining the query, and *json_path_name* is an optional name for the *path_expression*. The optional PASSING clause provides data values for the variables mentioned in the *path_expression*. The result of the input data evaluation using the aforementioned elements is called the *row pattern*, which is used as the source for row values in the constructed view.

COLUMNS (*json_table_column* [, ...])

The COLUMNS clause defining the schema of the constructed view. In this clause, you can specify each column to be filled with an SQL/JSON value obtained by applying a JSON path expression against the row pattern. *json_table_column* has the following variants:

name FOR ORDINALITY

Adds an ordinality column that provides sequential row numbering starting from 1. Each NESTED PATH (see below) gets its own counter for any nested ordinality columns.

name type [FORMAT JSON [ENCODING UTF8]] [PATH *path_expression*]

Inserts an SQL/JSON value obtained by applying *path_expression* against the row pattern into the view's output row after coercing it to specified *type*.

Specifying FORMAT JSON makes it explicit that you expect the value to be a valid json object. It only makes sense to specify FORMAT JSON if *type* is one of bpchar, bytea, character varying, name, json, jsonb, text, or a domain over these types.

Optionally, you can specify WRAPPER and QUOTES clauses to format the output. Note that specifying OMIT QUOTES overrides FORMAT JSON if also specified, because unquoted literals do not constitute valid json values.

Optionally, you can use ON EMPTY and ON ERROR clauses to specify whether to throw the error or return the specified value when the result of JSON path evaluation is empty and when an error occurs during JSON path evaluation or when coercing the SQL/JSON value to the specified type, respectively. The default for both is to return a NULL value.

> **Note**
>
> This clause is internally turned into and has the same semantics as JSON_VALUE or JSON_QUERY. The latter if the specified type is not a scalar type or if either of FORMAT JSON, WRAPPER, or QUOTES clause is present.

name type EXISTS [PATH *path_expression*]

Inserts a boolean value obtained by applying *path_expression* against the row pattern into the view's output row after coercing it to specified *type*.

The value corresponds to whether applying the PATH expression to the row pattern yields any values.

The specified *type* should have a cast from the boolean type.

Optionally, you can use ON ERROR to specify whether to throw the error or return the specified value when an error occurs during JSON path evaluation or when coercing SQL/JSON value to the specified type. The default is to return a boolean value FALSE.

> **Note**
>
> This clause is internally turned into and has the same semantics as JSON_EXISTS.

NESTED [PATH] *path_expression* [AS *json_path_name*] COLUMNS (*json_table_column* [, ...])

Extracts SQL/JSON values from nested levels of the row pattern, generates one or more columns as defined by the COLUMNS subclause, and inserts the extracted SQL/JSON values into those columns. The *json_table_column* expression in the COLUMNS subclause uses the same syntax as in the parent COLUMNS clause.

The NESTED PATH syntax is recursive, so you can go down multiple nested levels by specifying several NESTED PATH subclauses within each other. It allows to unnest the hierarchy of JSON objects and arrays in a single function invocation rather than chaining several JSON_TABLE expressions in an SQL statement.

> **Note**
>
> In each variant of *json_table_column* described above, if the PATH clause is omitted, path expression $.*name* is used, where *name* is the provided column name.

AS *json_path_name*

The optional *json_path_name* serves as an identifier of the provided *path_expression*. The name must be unique and distinct from the column names.

{ ERROR | EMPTY } ON ERROR

The optional ON ERROR can be used to specify how to handle errors when evaluating the top-level *path_expression*. Use ERROR if you want the errors to be thrown and EMPTY to return an empty table, that is, a table containing 0 rows. Note that this clause does not affect the errors that occur when evaluating columns, for which the behavior depends on whether the ON ERROR clause is specified against a given column.

Examples

In the examples that follow, the following table containing JSON data will be used:

```
CREATE TABLE my_films ( js jsonb );

INSERT INTO my_films VALUES (
'{ "favorites" : [
   { "kind" : "comedy", "films" : [
     { "title" : "Bananas",
       "director" : "Woody Allen"},
     { "title" : "The Dinner Game",
       "director" : "Francis Veber" } ] },
   { "kind" : "horror", "films" : [
```

```
    { "title" : "Psycho",
      "director" : "Alfred Hitchcock" } ] },
  { "kind" : "thriller", "films" : [
    { "title" : "Vertigo",
      "director" : "Alfred Hitchcock" } ] },
  { "kind" : "drama", "films" : [
    { "title" : "Yojimbo",
      "director" : "Akira Kurosawa" } ] }
 ] }');
```

The following query shows how to use `JSON_TABLE` to turn the JSON objects in the `my_films` table to a view containing columns for the keys `kind`, `title`, and `director` contained in the original JSON along with an ordinality column:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE (js, '$.favorites[*]' COLUMNS (
   id FOR ORDINALITY,
   kind text PATH '$.kind',
   title text PATH '$.films[*].title' WITH WRAPPER,
   director text PATH '$.films[*].director' WITH WRAPPER)) AS jt;
```

```
 id |   kind   |            title             |
 director
----+----------+------------------------------
+----------------------------------
  1 | comedy   | ["Bananas", "The Dinner Game"] | ["Woody Allen",
 "Francis Veber"]
  2 | horror   | ["Psycho"]                     | ["Alfred
 Hitchcock"]
  3 | thriller | ["Vertigo"]                    | ["Alfred
 Hitchcock"]
  4 | drama    | ["Yojimbo"]                    | ["Akira
 Kurosawa"]
(4 rows)
```

The following is a modified version of the above query to show the usage of `PASSING` arguments in the filter specified in the top-level JSON path expression and the various options for the individual columns:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE (js, '$.favorites[*] ? (@.films[*].director ==
 $filter)'
   PASSING 'Alfred Hitchcock' AS filter, 'Vertigo' AS filter2
     COLUMNS (
     id FOR ORDINALITY,
     kind text PATH '$.kind',
     title text FORMAT JSON PATH '$.films[*].title' OMIT QUOTES,
     director text PATH '$.films[*].director' KEEP QUOTES)) AS jt;
```

```
 id |   kind   |  title  |      director
----+----------+---------+--------------------
  1 | horror   | Psycho  | "Alfred Hitchcock"
  2 | thriller | Vertigo | "Alfred Hitchcock"
```

```
(2 rows)
```

The following is a modified version of the above query to show the usage of `NESTED PATH` for populating title and director columns, illustrating how they are joined to the parent columns id and kind:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*] ? (@.films[*].director ==
 $filter)'
   PASSING 'Alfred Hitchcock' AS filter
   COLUMNS (
    id FOR ORDINALITY,
    kind text PATH '$.kind',
    NESTED PATH '$.films[*]' COLUMNS (
      title text FORMAT JSON PATH '$.title' OMIT QUOTES,
      director text PATH '$.director' KEEP QUOTES))) AS jt;

 id |   kind   |  title  |      director
----+----------+---------+--------------------
  1 | horror   | Psycho  | "Alfred Hitchcock"
  2 | thriller | Vertigo | "Alfred Hitchcock"
(2 rows)
```

The following is the same query but without the filter in the root path:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*]'
   COLUMNS (
    id FOR ORDINALITY,
    kind text PATH '$.kind',
    NESTED PATH '$.films[*]' COLUMNS (
      title text FORMAT JSON PATH '$.title' OMIT QUOTES,
      director text PATH '$.director' KEEP QUOTES))) AS jt;

 id |   kind   |      title      |      director
----+----------+-----------------+--------------------
  1 | comedy   | Bananas         | "Woody Allen"
  1 | comedy   | The Dinner Game | "Francis Veber"
  2 | horror   | Psycho          | "Alfred Hitchcock"
  3 | thriller | Vertigo         | "Alfred Hitchcock"
  4 | drama    | Yojimbo         | "Akira Kurosawa"
(5 rows)
```

The following shows another query using a different `JSON` object as input. It shows the UNION "sibling join" between `NESTED` paths `$.movies[*]` and `$.books[*]` and also the usage of `FOR ORDINALITY` column at `NESTED` levels (columns `movie_id`, `book_id`, and `author_id`):

```
SELECT * FROM JSON_TABLE (
'{"favorites":
    {"movies":
      [{"name": "One", "director": "John Doe"},
       {"name": "Two", "director": "Don Joe"}],
```

```
     "books":
      [{"name": "Mystery", "authors": [{"name": "Brown Dan"}]},
       {"name": "Wonder", "authors": [{"name": "Jun Murakami"},
 {"name":"Craig Doe"}]}]
}}'::json, '$.favorites[*]'
COLUMNS (
  user_id FOR ORDINALITY,
  NESTED '$.movies[*]'
    COLUMNS (
    movie_id FOR ORDINALITY,
    mname text PATH '$.name',
    director text),
  NESTED '$.books[*]'
    COLUMNS (
      book_id FOR ORDINALITY,
      bname text PATH '$.name',
      NESTED '$.authors[*]'
        COLUMNS (
          author_id FOR ORDINALITY,
          author_name text PATH '$.name'))));


 user_id | movie_id | mname | director | book_id |  bname  |
 author_id | author_name
---------+----------+-------+----------+---------+---------
+-----------+--------------
       1 |        1 | One   | John Doe |         |         |
    |
       1 |        2 | Two   | Don Joe  |         |         |
    |
       1 |          |       |          |       1 | Mystery |
  1 | Brown Dan
       1 |          |       |          |       2 | Wonder  |
  1 | Jun Murakami
       1 |          |       |          |       2 | Wonder  |
  2 | Craig Doe
(5 rows)
```

9.17. Sequence Manipulation Functions

This section describes functions for operating on *sequence objects*, also called sequence generators or just sequences. Sequence objects are special single-row tables created with CREATE SEQUENCE. Sequence objects are commonly used to generate unique identifiers for rows of a table. The sequence functions, listed in Table 9.53, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

Table 9.53. Sequence Functions

Function Description
`nextval(regclass)` → `bigint` Advances the sequence object to its next value and returns that value. This is done atomically: even if multiple sessions execute `nextval` concurrently, each will safely receive a distinct sequence value. If the sequence object has been created with default parameters, successive `nextval` calls will return successive values beginning with 1. Other behaviors can be obtained by using appropriate parameters in the CREATE SEQUENCE command.

Function Description	
This function requires USAGE or UPDATE privilege on the sequence.	

setval(regclass, bigint [, boolean]) → bigint
Sets the sequence object's current value, and optionally its is_called flag. The two-parameter form sets the sequence's last_value field to the specified value and sets its is_called field to true, meaning that the next nextval will advance the sequence before returning a value. The value that will be reported by currval is also set to the specified value. In the three-parameter form, is_called can be set to either true or false. true has the same effect as the two-parameter form. If it is set to false, the next nextval will return exactly the specified value, and sequence advancement commences with the following nextval. Furthermore, the value reported by currval is not changed in this case. For example,
`SELECT setval('myseq', 42);  Next nextval will return 43`
`SELECT setval('myseq', 42, true);  Same as above`
`SELECT setval('myseq', 42, false);  Next nextval will return 42`
The result returned by setval is just the value of its second argument.
This function requires UPDATE privilege on the sequence.	

currval(regclass) → bigint Returns the value most recently obtained by nextval for this sequence in the current session. (An error is reported if nextval has never been called for this sequence in this session.) Because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed nextval since the current session did. This function requires USAGE or SELECT privilege on the sequence.	
lastval() → bigint Returns the value most recently returned by nextval in the current session. This function is identical to currval, except that instead of taking the sequence name as an argument it refers to whichever sequence nextval was most recently applied to in the current session. It is an error to call lastval if nextval has not yet been called in the current session. This function requires USAGE or SELECT privilege on the last used sequence.	

Caution

To avoid blocking concurrent transactions that obtain numbers from the same sequence, the value obtained by nextval is not reclaimed for re-use if the calling transaction later aborts. This means that transaction aborts or database crashes can result in gaps in the sequence of assigned values. That can happen without a transaction abort, too. For example an INSERT with an ON CONFLICT clause will compute the to-be-inserted tuple, including doing any required nextval calls, before detecting any conflict that would cause it to follow the ON CONFLICT rule instead. Thus, PostgreSQL sequence objects *cannot be used to obtain "gapless" sequences*.

Likewise, sequence state changes made by setval are immediately visible to other transactions, and are not undone if the calling transaction rolls back.

If the database cluster crashes before committing a transaction containing a nextval or setval call, the sequence state change might not have made its way to persistent storage, so that it is uncertain whether the sequence will have its original or updated state after the cluster restarts. This is harmless for usage of the sequence within the database, since other effects of

uncommitted transactions will not be visible either. However, if you wish to use a sequence value for persistent outside-the-database purposes, make sure that the `nextval` call has been committed before doing so.

The sequence to be operated on by a sequence function is specified by a `regclass` argument, which is simply the OID of the sequence in the `pg_class` system catalog. You do not have to look up the OID by hand, however, since the `regclass` data type's input converter will do the work for you. See Section 8.19 for details.

9.18. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

Tip

If your needs go beyond the capabilities of these conditional expressions, you might want to consider writing a server-side function in a more expressive programming language.

Note

Although `COALESCE`, `GREATEST`, and `LEAST` are syntactically similar to functions, they are not ordinary functions, and thus cannot be used with explicit `VARIADIC` array arguments.

9.18.1. CASE

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other programming languages:

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

`CASE` clauses can be used wherever an expression is valid. Each *`condition`* is an expression that returns a `boolean` result. If the condition's result is true, the value of the `CASE` expression is the *`result`* that follows the condition, and the remainder of the `CASE` expression is not processed. If the condition's result is not true, any subsequent `WHEN` clauses are examined in the same manner. If no `WHEN` *`condition`* yields true, the value of the `CASE` expression is the *`result`* of the `ELSE` clause. If the `ELSE` clause is omitted and no condition is true, the result is null.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3
```

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

The data types of all the *result* expressions must be convertible to a single output type. See Section 10.5 for more details.

There is a "simple" form of CASE expression that is a variant of the general form above:

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

The first *expression* is computed, then compared to each of the *value* expressions in the WHEN clauses until one is found that is equal to it. If no match is found, the *result* of the ELSE clause (or a null value) is returned. This is similar to the switch statement in C.

The example above can be written using the simple CASE syntax:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **Note**
>
> As described in Section 4.2.14, there are various situations in which subexpressions of an expression are evaluated at different times, so that the principle that "CASE evaluates only necessary subexpressions" is not ironclad. For example a constant 1/0 subexpression will

usually result in a division-by-zero failure at planning time, even if it's within a CASE arm that would never be entered at run time.

9.18.2. COALESCE

```
COALESCE(value [, ...])
```

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

This returns description if it is not null, otherwise short_description if it is not null, otherwise (none).

The arguments must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details).

Like a CASE expression, COALESCE only evaluates the arguments that are needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

9.18.3. NULLIF

```
NULLIF(value1, value2)
```

The NULLIF function returns a null value if *value1* equals *value2*; otherwise it returns *value1*. This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value, '(none)') ...
```

In this example, if value is (none), null is returned, otherwise the value of value is returned.

The two arguments must be of comparable types. To be specific, they are compared exactly as if you had written *value1 = value2*, so there must be a suitable = operator available.

The result has the same type as the first argument — but there is a subtlety. What is actually returned is the first argument of the implied = operator, and in some cases that will have been promoted to match the second argument's type. For example, NULLIF(1, 2.2) yields numeric, because there is no integer = numeric operator, only numeric = numeric.

9.18.4. GREATEST and LEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions. The expressions must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details).

NULL values in the argument list are ignored. The result will be NULL only if all the expressions evaluate to NULL. (This is a deviation from the SQL standard. According to the standard, the return value is NULL if any argument is NULL. Some other databases behave this way.)

9.19. Array Functions and Operators

Table 9.54 shows the specialized operators available for array types. In addition to those, the usual comparison operators shown in Table 9.1 are available for arrays. The comparison operators compare the array contents element-by-element, using the default B-tree comparison function for the element data type, and sort based on the first difference. In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order.

Table 9.54. Array Operators

Operator **Description** **Example(s)**

`anyarray @> anyarray → boolean`
Does the first array contain the second, that is, does each element appearing in the second array equal some element of the first array? (Duplicates are not treated specially, thus `ARRAY[1]` and `ARRAY[1,1]` are each considered to contain the other.)
`ARRAY[1,4,3] @> ARRAY[3,1,3] → t`

`anyarray <@ anyarray → boolean`
Is the first array contained by the second?
`ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6] → t`

`anyarray && anyarray → boolean`
Do the arrays overlap, that is, have any elements in common?
`ARRAY[1,4,3] && ARRAY[2,1] → t`

`anycompatiblearray || anycompatiblearray → anycompatiblearray`
Concatenates the two arrays. Concatenating a null or empty array is a no-op; otherwise the arrays must have the same number of dimensions (as illustrated by the first example) or differ in number of dimensions by one (as illustrated by the second). If the arrays are not of identical element types, they will be coerced to a common type (see Section 10.5).
`ARRAY[1,2,3] || ARRAY[4,5,6,7] → {1,2,3,4,5,6,7}`
`ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9.9]] → {{1,2,3},{4,5,6},{7,8,9.9}}`

`anycompatible || anycompatiblearray → anycompatiblearray`
Concatenates an element onto the front of an array (which must be empty or one-dimensional).
`3 || ARRAY[4,5,6] → {3,4,5,6}`

`anycompatiblearray || anycompatible → anycompatiblearray`
Concatenates an element onto the end of an array (which must be empty or one-dimensional).
`ARRAY[4,5,6] || 7 → {4,5,6,7}`

See Section 8.15 for more details about array operator behavior. See Section 11.2 for more details about which operators support indexed operations.

Table 9.55 shows the functions available for use with array types. See Section 8.15 for more information and examples of the use of these functions.

Table 9.55. Array Functions

Function Description Example(s)

`array_append` (`anycompatiblearray`, `anycompatible`) → `anycompatiblearray`
Appends an element to the end of an array (same as the `anycompatiblearray || anycompatible` operator).
`array_append(ARRAY[1,2], 3)` → `{1,2,3}`

`array_cat` (`anycompatiblearray`, `anycompatiblearray`) → `anycompatiblearray`
Concatenates two arrays (same as the `anycompatiblearray || anycompatiblearray` operator).
`array_cat(ARRAY[1,2,3], ARRAY[4,5])` → `{1,2,3,4,5}`

`array_dims` (`anyarray`) → `text`
Returns a text representation of the array's dimensions.
`array_dims(ARRAY[[1,2,3], [4,5,6]])` → `[1:2][1:3]`

`array_fill` (`anyelement`, `integer[]` [, `integer[]`]) → `anyarray`
Returns an array filled with copies of the given value, having dimensions of the lengths specified by the second argument. The optional third argument supplies lower-bound values for each dimension (which default to all `1`).
`array_fill(11, ARRAY[2,3])` → `{{11,11,11},{11,11,11}}`
`array_fill(7, ARRAY[3], ARRAY[2])` → `[2:4]={7,7,7}`

`array_length` (`anyarray`, `integer`) → `integer`
Returns the length of the requested array dimension. (Produces NULL instead of 0 for empty or missing array dimensions.)
`array_length(array[1,2,3], 1)` → `3`
`array_length(array[]::int[], 1)` → `NULL`
`array_length(array['text'], 2)` → `NULL`

`array_lower` (`anyarray`, `integer`) → `integer`
Returns the lower bound of the requested array dimension.
`array_lower('[0:2]={1,2,3}'::integer[], 1)` → `0`

`array_ndims` (`anyarray`) → `integer`
Returns the number of dimensions of the array.
`array_ndims(ARRAY[[1,2,3], [4,5,6]])` → `2`

`array_position` (`anycompatiblearray`, `anycompatible` [, `integer`]) → `integer`
Returns the subscript of the first occurrence of the second argument in the array, or `NULL` if it's not present. If the third argument is given, the search begins at that subscript. The array must be one-dimensional. Comparisons are done using `IS NOT DISTINCT FROM` semantics, so it is possible to search for `NULL`.
`array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon')` → `2`

`array_positions` (`anycompatiblearray`, `anycompatible`) → `integer[]` Returns an array of the subscripts of all occurrences of the second argument in the array given as first argument. The array must be one-dimensional. Comparisons are done using `IS NOT DISTINCT FROM` semantics, so it is possible to search for `NULL`. `NULL`

Function **Description** **Example(s)**
is returned only if the array is `NULL`; if the value is not found in the array, an empty array is returned. `array_positions(ARRAY['A','A','B','A'], 'A')` → `{1,2,4}`

`array_prepend` (`anycompatible`, `anycompatiblearray`) → `anycompatiblearray`
Prepends an element to the beginning of an array (same as the `anycompatible || anycompatiblearray` operator).
`array_prepend(1, ARRAY[2,3])` → `{1,2,3}`

`array_remove` (`anycompatiblearray`, `anycompatible`) → `anycompatiblearray`
Removes all elements equal to the given value from the array. The array must be one-dimensional. Comparisons are done using `IS NOT DISTINCT FROM` semantics, so it is possible to remove `NULL`s.
`array_remove(ARRAY[1,2,3,2], 2)` → `{1,3}`

`array_replace` (`anycompatiblearray`, `anycompatible`, `anycompatible`) → `anycompatiblearray`
Replaces each array element equal to the second argument with the third argument.
`array_replace(ARRAY[1,2,5,4], 5, 3)` → `{1,2,3,4}`

`array_sample` (*`array`* `anyarray`, *`n`* `integer`) → `anyarray`
Returns an array of *`n`* items randomly selected from *`array`*. *`n`* may not exceed the length of *`array`*'s first dimension. If *`array`* is multi-dimensional, an "item" is a slice having a given first subscript.
`array_sample(ARRAY[1,2,3,4,5,6], 3)` → `{2,6,1}`
`array_sample(ARRAY[[1,2],[3,4],[5,6]], 2)` → `{{5,6},{1,2}}`

`array_shuffle` (`anyarray`) → `anyarray`
Randomly shuffles the first dimension of the array.
`array_shuffle(ARRAY[[1,2],[3,4],[5,6]])` → `{{5,6},{1,2},{3,4}}`

`array_to_string` (*`array`* `anyarray`, *`delimiter`* `text` [, *`null_string`* `text`]) → `text`
Converts each array element to its text representation, and concatenates those separated by the *`delimiter`* string. If *`null_string`* is given and is not `NULL`, then `NULL` array entries are represented by that string; otherwise, they are omitted. See also `string_to_array`.
`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')` → `1,2,3,*,5`

`array_upper` (`anyarray`, `integer`) → `integer`
Returns the upper bound of the requested array dimension.
`array_upper(ARRAY[1,8,3,7], 1)` → `4`

`cardinality` (`anyarray`) → `integer`
Returns the total number of elements in the array, or 0 if the array is empty.
`cardinality(ARRAY[[1,2],[3,4]])` → `4`

`trim_array` (*`array`* `anyarray`, *`n`* `integer`) → `anyarray` Trims an array by removing the last *`n`* elements. If the array is multidimensional, only the first dimension is trimmed.

Function Description Example(s)
`trim_array(ARRAY[1,2,3,4,5,6], 2)` → `{1,2,3,4}`

`unnest ( anyarray )` → `setof anyelement`
Expands an array into a set of rows. The array's elements are read out in storage order.
`unnest(ARRAY[1,2])` →
`1`
`2`
`unnest(ARRAY[['foo','bar'],['baz','quux']])` →
`foo`
`bar`
`baz`
`quux`

`unnest ( anyarray, anyarray [, ... ] )` → `setof anyelement, anyelement [, ... ]`
Expands multiple arrays (possibly of different data types) into a set of rows. If the arrays are not all the same length then the shorter ones are padded with `NULL`s. This form is only allowed in a query's FROM clause; see Section 7.2.1.4.
`select * from unnest(ARRAY[1,2], ARRAY['foo','bar','baz']) as x(a,b)` →
`a | b`
`---+-----`
`1 | foo`
`2 | bar`
`  | baz`

See also Section 9.21 about the aggregate function `array_agg` for use with arrays.

9.20. Range/Multirange Functions and Operators

See Section 8.17 for an overview of range types.

Table 9.56 shows the specialized operators available for range types. Table 9.57 shows the specialized operators available for multirange types. In addition to those, the usual comparison operators shown in Table 9.1 are available for range and multirange types. The comparison operators order first by the range lower bounds, and only if those are equal do they compare the upper bounds. The multirange operators compare each range until one is unequal. This does not usually result in a useful overall ordering, but the operators are provided to allow unique indexes to be constructed on ranges.

Table 9.56. Range Operators

Operator Description Example(s)
`anyrange @> anyrange` → `boolean`

Operator **Description** **Example(s)**
Does the first range contain the second? `int4range(2,4) @> int4range(2,3)` → `t`

`anyrange @> anyelement` → `boolean`
Does the range contain the element?
`'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp` → `t`

`anyrange <@ anyrange` → `boolean`
Is the first range contained by the second?
`int4range(2,4) <@ int4range(1,7)` → `t`

`anyelement <@ anyrange` → `boolean`
Is the element contained in the range?
`42 <@ int4range(1,7)` → `f`

`anyrange && anyrange` → `boolean`
Do the ranges overlap, that is, have any elements in common?
`int8range(3,7) && int8range(4,12)` → `t`

`anyrange << anyrange` → `boolean`
Is the first range strictly left of the second?
`int8range(1,10) << int8range(100,110)` → `t`

`anyrange >> anyrange` → `boolean`
Is the first range strictly right of the second?
`int8range(50,60) >> int8range(20,30)` → `t`

`anyrange &< anyrange` → `boolean`
Does the first range not extend to the right of the second?
`int8range(1,20) &< int8range(18,20)` → `t`

`anyrange &> anyrange` → `boolean`
Does the first range not extend to the left of the second?
`int8range(7,20) &> int8range(5,10)` → `t`

`anyrange -|- anyrange` → `boolean`
Are the ranges adjacent?
`numrange(1.1,2.2) -|- numrange(2.2,3.3)` → `t`

`anyrange + anyrange` → `anyrange`
Computes the union of the ranges. The ranges must overlap or be adjacent, so that the union is a single range (but see `range_merge()`).
`numrange(5,15) + numrange(10,20)` → `[5,20)`

`anyrange * anyrange` → `anyrange`
Computes the intersection of the ranges.
`int8range(5,15) * int8range(10,20)` → `[10,15)`

`anyrange - anyrange` → `anyrange`
Computes the difference of the ranges. The second range must not be contained in the first in such a way that the difference would not be a single range.
`int8range(5,15) - int8range(10,20)` → `[5,10)`

Table 9.57. Multirange Operators

Operator Description Example(s)

`anymultirange @> anymultirange → boolean`
Does the first multirange contain the second?
`'{[2,4)}'::int4multirange @> '{[2,3)}'::int4multirange → t`

`anymultirange @> anyrange → boolean`
Does the multirange contain the range?
`'{[2,4)}'::int4multirange @> int4range(2,3) → t`

`anymultirange @> anyelement → boolean`
Does the multirange contain the element?
`'{[2011-01-01,2011-03-01)}'::tsmultirange @> '2011-01-10'::timestamp → t`

`anyrange @> anymultirange → boolean`
Does the range contain the multirange?
`'[2,4)'::int4range @> '{[2,3)}'::int4multirange → t`

`anymultirange <@ anymultirange → boolean`
Is the first multirange contained by the second?
`'{[2,4)}'::int4multirange <@ '{[1,7)}'::int4multirange → t`

`anymultirange <@ anyrange → boolean`
Is the multirange contained by the range?
`'{[2,4)}'::int4multirange <@ int4range(1,7) → t`

`anyrange <@ anymultirange → boolean`
Is the range contained by the multirange?
`int4range(2,4) <@ '{[1,7)}'::int4multirange → t`

`anyelement <@ anymultirange → boolean`
Is the element contained by the multirange?
`4 <@ '{[1,7)}'::int4multirange → t`

`anymultirange && anymultirange → boolean`
Do the multiranges overlap, that is, have any elements in common?
`'{[3,7)}'::int8multirange && '{[4,12)}'::int8multirange → t`

`anymultirange && anyrange → boolean`
Does the multirange overlap the range?
`'{[3,7)}'::int8multirange && int8range(4,12) → t`

`anyrange && anymultirange → boolean`
Does the range overlap the multirange?
`int8range(3,7) && '{[4,12)}'::int8multirange → t`

`anymultirange << anymultirange → boolean`
Is the first multirange strictly left of the second?
`'{[1,10)}'::int8multirange << '{[100,110)}'::int8multirange → t`

`anymultirange << anyrange → boolean` Is the multirange strictly left of the range?

Operator **Description** **Example(s)**
`'{[1,10)}'::int8multirange << int8range(100,110)` → `t`

`anyrange << anymultirange` → `boolean`
Is the range strictly left of the multirange?
`int8range(1,10) << '{[100,110)}'::int8multirange` → `t`

`anymultirange >> anymultirange` → `boolean`
Is the first multirange strictly right of the second?
`'{[50,60)}'::int8multirange >> '{[20,30)}'::int8multirange` → `t`

`anymultirange >> anyrange` → `boolean`
Is the multirange strictly right of the range?
`'{[50,60)}'::int8multirange >> int8range(20,30)` → `t`

`anyrange >> anymultirange` → `boolean`
Is the range strictly right of the multirange?
`int8range(50,60) >> '{[20,30)}'::int8multirange` → `t`

`anymultirange &< anymultirange` → `boolean`
Does the first multirange not extend to the right of the second?
`'{[1,20)}'::int8multirange &< '{[18,20)}'::int8multirange` → `t`

`anymultirange &< anyrange` → `boolean`
Does the multirange not extend to the right of the range?
`'{[1,20)}'::int8multirange &< int8range(18,20)` → `t`

`anyrange &< anymultirange` → `boolean`
Does the range not extend to the right of the multirange?
`int8range(1,20) &< '{[18,20)}'::int8multirange` → `t`

`anymultirange &> anymultirange` → `boolean`
Does the first multirange not extend to the left of the second?
`'{[7,20)}'::int8multirange &> '{[5,10)}'::int8multirange` → `t`

`anymultirange &> anyrange` → `boolean`
Does the multirange not extend to the left of the range?
`'{[7,20)}'::int8multirange &> int8range(5,10)` → `t`

`anyrange &> anymultirange` → `boolean`
Does the range not extend to the left of the multirange?
`int8range(7,20) &> '{[5,10)}'::int8multirange` → `t`

`anymultirange -|- anymultirange` → `boolean`
Are the multiranges adjacent?
`'{[1.1,2.2)}'::nummultirange -|- '{[2.2,3.3)}'::nummultirange` → `t`

`anymultirange -|- anyrange` → `boolean`
Is the multirange adjacent to the range?
`'{[1.1,2.2)}'::nummultirange -|- numrange(2.2,3.3)` → `t`

`anyrange -|- anymultirange` → `boolean`

Operator Description Example(s)
Is the range adjacent to the multirange? `numrange(1.1,2.2) -|- '{[2.2,3.3)}'::nummultirange` → `t`

`anymultirange + anymultirange → anymultirange`
Computes the union of the multiranges. The multiranges need not overlap or be adjacent.
`'{[5,10)}'::nummultirange + '{[15,20)}'::nummultirange` → `{[5,10), [15,20)}`

`anymultirange * anymultirange → anymultirange`
Computes the intersection of the multiranges.
`'{[5,15)}'::int8multirange * '{[10,20)}'::int8multirange` → `{[10,15)}`

`anymultirange - anymultirange → anymultirange`
Computes the difference of the multiranges.
`'{[5,20)}'::int8multirange - '{[10,15)}'::int8multirange` → `{[5,10), [15,20)}`

The left-of/right-of/adjacent operators always return false when an empty range or multirange is involved; that is, an empty range is not considered to be either before or after any other range.

Elsewhere empty ranges and multiranges are treated as the additive identity: anything unioned with an empty value is itself. Anything minus an empty value is itself. An empty multirange has exactly the same points as an empty range. Every range contains the empty range. Every multirange contains as many empty ranges as you like.

The range union and difference operators will fail if the resulting range would need to contain two disjoint sub-ranges, as such a range cannot be represented. There are separate operators for union and difference that take multirange parameters and return a multirange, and they do not fail even if their arguments are disjoint. So if you need a union or difference operation for ranges that may be disjoint, you can avoid errors by first casting your ranges to multiranges.

Table 9.58 shows the functions available for use with range types. Table 9.59 shows the functions available for use with multirange types.

Table 9.58. Range Functions

Function Description Example(s)

`lower ( anyrange ) → anyelement`
Extracts the lower bound of the range (`NULL` if the range is empty or has no lower bound).
`lower(numrange(1.1,2.2))` → `1.1`

`upper ( anyrange ) → anyelement`
Extracts the upper bound of the range (`NULL` if the range is empty or has no upper bound).
`upper(numrange(1.1,2.2))` → `2.2`

`isempty ( anyrange ) → boolean`
Is the range empty?
`isempty(numrange(1.1,2.2))` → `f`

Function Description Example(s)

`lower_inc` (`anyrange`) → `boolean`
Is the range's lower bound inclusive?
`lower_inc(numrange(1.1,2.2))` → `t`

`upper_inc` (`anyrange`) → `boolean`
Is the range's upper bound inclusive?
`upper_inc(numrange(1.1,2.2))` → `f`

`lower_inf` (`anyrange`) → `boolean`
Does the range have no lower bound? (A lower bound of `-Infinity` returns false.)
`lower_inf('(,)'::daterange)` → `t`

`upper_inf` (`anyrange`) → `boolean`
Does the range have no upper bound? (An upper bound of `Infinity` returns false.)
`upper_inf('(,)'::daterange)` → `t`

`range_merge` (`anyrange`, `anyrange`) → `anyrange`
Computes the smallest range that includes both of the given ranges.
`range_merge('[1,2)'::int4range, '[3,4)'::int4range)` → `[1,4)`

Table 9.59. Multirange Functions

Function Description Example(s)

`lower` (`anymultirange`) → `anyelement`
Extracts the lower bound of the multirange (`NULL` if the multirange is empty has no lower bound).
`lower('{[1.1,2.2)}'::nummultirange)` → `1.1`

`upper` (`anymultirange`) → `anyelement`
Extracts the upper bound of the multirange (`NULL` if the multirange is empty or has no upper bound).
`upper('{[1.1,2.2)}'::nummultirange)` → `2.2`

`isempty` (`anymultirange`) → `boolean`
Is the multirange empty?
`isempty('{[1.1,2.2)}'::nummultirange)` → `f`

`lower_inc` (`anymultirange`) → `boolean`
Is the multirange's lower bound inclusive?
`lower_inc('{[1.1,2.2)}'::nummultirange)` → `t`

`upper_inc` (`anymultirange`) → `boolean`
Is the multirange's upper bound inclusive?
`upper_inc('{[1.1,2.2)}'::nummultirange)` → `f`

`lower_inf` (`anymultirange`) → `boolean`
Does the multirange have no lower bound? (A lower bound of `-Infinity` returns false.)
`lower_inf('{(,)}'::datemultirange)` → `t`

Function Description Example(s)

`upper_inf ( anymultirange ) → boolean`
Does the multirange have no upper bound? (An upper bound of `Infinity` returns false.)
`upper_inf('{(,)}'::datemultirange) → t`

`range_merge ( anymultirange ) → anyrange`
Computes the smallest range that includes the entire multirange.
`range_merge('{[1,2), [3,4)}'::int4multirange) → [1,4)`

`multirange ( anyrange ) → anymultirange`
Returns a multirange containing just the given range.
`multirange('[1,2)'::int4range) → {[1,2)}`

`unnest ( anymultirange ) → setof anyrange`
Expands a multirange into a set of ranges in ascending order.
`unnest('{[1,2), [3,4)}'::int4multirange) →`
 	
`[1,2)`
`[3,4)`

The `lower_inc`, `upper_inc`, `lower_inf`, and `upper_inf` functions all return false for an empty range or multirange.

9.21. Aggregate Functions

Aggregate functions compute a single result from a set of input values. The built-in general-purpose aggregate functions are listed in Table 9.60 while statistical aggregates are in Table 9.61. The built-in within-group ordered-set aggregate functions are listed in Table 9.62 while the built-in within-group hypothetical-set ones are in Table 9.63. Grouping operations, which are closely related to aggregate functions, are listed in Table 9.64. The special syntax considerations for aggregate functions are explained in Section 4.2.7. Consult Section 2.7 for additional introductory information.

Aggregate functions that support *Partial Mode* are eligible to participate in various optimizations, such as parallel aggregation.

While all aggregates below accept an optional `ORDER BY` clause (as outlined in Section 4.2.7), the clause has only been added to aggregates whose output is affected by ordering.

Table 9.60. General-Purpose Aggregate Functions

Function Description	Partial Mode
`any_value ( anyelement ) → *same as input type*` Returns an arbitrary value from the non-null input values.	Yes
`array_agg ( anynonarray ORDER BY input_sort_columns ) → anyarray` Collects all the input values, including nulls, into an array.	Yes
`array_agg ( anyarray ORDER BY input_sort_columns ) → anyarray` Concatenates all the input arrays into an array of one higher dimension. (The inputs must all have the same dimensionality, and cannot be empty or null.)	Yes

Function Description	Partial Mode
avg (smallint) → numeric avg (integer) → numeric avg (bigint) → numeric avg (numeric) → numeric avg (real) → double precision avg (double precision) → double precision avg (interval) → interval Computes the average (arithmetic mean) of all the non-null input values.	Yes
bit_and (smallint) → smallint bit_and (integer) → integer bit_and (bigint) → bigint bit_and (bit) → bit Computes the bitwise AND of all non-null input values.	Yes
bit_or (smallint) → smallint bit_or (integer) → integer bit_or (bigint) → bigint bit_or (bit) → bit Computes the bitwise OR of all non-null input values.	Yes
bit_xor (smallint) → smallint bit_xor (integer) → integer bit_xor (bigint) → bigint bit_xor (bit) → bit Computes the bitwise exclusive OR of all non-null input values. Can be useful as a checksum for an unordered set of values.	Yes
bool_and (boolean) → boolean Returns true if all non-null input values are true, otherwise false.	Yes
bool_or (boolean) → boolean Returns true if any non-null input value is true, otherwise false.	Yes
count (*) → bigint Computes the number of input rows.	Yes
count ("any") → bigint Computes the number of input rows in which the input value is not null.	Yes
every (boolean) → boolean This is the SQL standard's equivalent to bool_and.	Yes
json_agg (anyelement ORDER BY input_sort_columns) → json jsonb_agg (anyelement ORDER BY input_sort_columns) → jsonb Collects all the input values, including nulls, into a JSON array. Values are converted to JSON as per to_json or to_jsonb.	No
json_agg_strict (anyelement) → json jsonb_agg_strict (anyelement) → jsonb Collects all the input values, skipping nulls, into a JSON array. Values are converted to JSON as per to_json or to_jsonb.	No

Function Description	Partial Mode
`json_arrayagg` ([*value_expression*] [`ORDER BY` *sort_expression*] [{ `NULL` | `ABSENT` } `ON NULL`] [`RETURNING` *data_type* [`FORMAT JSON` [`ENCODING UTF8`]]]) Behaves in the same way as `json_array` but as an aggregate function so it only takes one *value_expression* parameter. If `ABSENT ON NULL` is specified, any NULL values are omitted. If `ORDER BY` is specified, the elements will appear in the array in that order rather than in the input order. `SELECT json_arrayagg(v) FROM (VALUES(2),(1)) t(v)` → `[2, 1]`	No
`json_objectagg` ([{ *key_expression* { `VALUE` | ':' } *value_expression* }] [{ `NULL` | `ABSENT` } `ON NULL`] [{ `WITH` | `WITHOUT` } `UNIQUE` [`KEYS`]] [`RETURNING` *data_type* [`FORMAT JSON` [`ENCODING UTF8`]]]) Behaves like `json_object`, but as an aggregate function, so it only takes one *key_expression* and one *value_expression* parameter. `SELECT json_objectagg(k:v) FROM (VALUES ('a'::text,current_date),('b',current_date + 1)) AS t(k,v)` → `{ "a" : "2022-05-10", "b" : "2022-05-11" }`	No

`json_object_agg` (*key* `"any"`, *value* `"any"` `ORDER BY` *input_sort_columns*) → `json`
`jsonb_object_agg` (*key* `"any"`, *value* `"any"` `ORDER BY` *input_sort_columns*) → `jsonb`
Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. Values can be null, but keys cannot.
	No

`json_object_agg_strict` (*key* `"any"`, *value* `"any"`) → `json`
`jsonb_object_agg_strict` (*key* `"any"`, *value* `"any"`) → `jsonb`
Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. The *key* can not be null. If the *value* is null then the entry is skipped,
	No

`json_object_agg_unique` (*key* `"any"`, *value* `"any"`) → `json`
`jsonb_object_agg_unique` (*key* `"any"`, *value* `"any"`) → `jsonb`
Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. Values can be null, but keys cannot. If there is a duplicate key an error is thrown.
	No

`json_object_agg_unique_strict` (*key* `"any"`, *value* `"any"`) → `json`
`jsonb_object_agg_unique_strict` (*key* `"any"`, *value* `"any"`) → `jsonb`
Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. The *key* can not be null. If the *value* is null then the entry is skipped. If there is a duplicate key an error is thrown.
	No

`max` (*see text*) → *same as input type* Computes the maximum of the non-null input values. Available for any numeric, string, date/time, or enum type, as well as `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8`, and arrays of any of these types.	Yes
`min` (*see text*) → *same as input type*	Yes

Function Description	Partial Mode
Computes the minimum of the non-null input values. Available for any numeric, string, date/time, or enum type, as well as `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8`, and arrays of any of these types.	

`range_agg ( value anyrange ) → anymultirange`
`range_agg ( value anymultirange ) → anymultirange`
Computes the union of the non-null input values.
	No

`range_intersect_agg ( value anyrange ) → anyrange`
`range_intersect_agg ( value anymultirange ) → anymultirange`
Computes the intersection of the non-null input values.
	No

`string_agg ( value text, delimiter text ) → text`
`string_agg ( value bytea, delimiter bytea ORDER BY input_sort_columns ) → bytea`
Concatenates the non-null input values into a string. Each value after the first is preceded by the corresponding *delimiter* (if it's not null).
	Yes

`sum ( smallint ) → bigint`
`sum ( integer ) → bigint`
`sum ( bigint ) → numeric`
`sum ( numeric ) → numeric`
`sum ( real ) → real`
`sum ( double precision ) → double precision`
`sum ( interval ) → interval`
`sum ( money ) → money`
Computes the sum of the non-null input values.
	Yes

`xmlagg ( xml ORDER BY input_sort_columns ) → xml` Concatenates the non-null XML input values (see Section 9.15.1.8).	No

It should be noted that except for `count`, these functions return a null value when no rows are selected. In particular, `sum` of no rows returns null, not zero as one might expect, and `array_agg` returns null rather than an empty array when there are no input rows. The `coalesce` function can be used to substitute zero or an empty array for null when necessary.

The aggregate functions `array_agg`, `json_agg`, `jsonb_agg`, `json_agg_strict`, `jsonb_agg_strict`, `json_object_agg`, `jsonb_object_agg`, `json_object_agg_strict`, `jsonb_object_agg_strict`, `json_object_agg_unique`, `jsonb_object_agg_unique`, `json_object_agg_unique_strict`, `jsonb_object_agg_unique_strict`, `string_agg`, and `xmlagg`, as well as similar user-defined aggregate functions, produce meaningfully different result values depending on the order of the input values. This ordering is unspecified by default, but can be controlled by writing an `ORDER BY` clause within the aggregate call, as shown in Section 4.2.7. Alternatively, supplying the input values from a sorted subquery will usually work. For example:

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Beware that this approach can fail if the outer query level contains additional processing, such as a join, because that might cause the subquery's output to be reordered before the aggregate is computed.

Note

The boolean aggregates `bool_and` and `bool_or` correspond to the standard SQL aggregates `every` and `any` or `some`. PostgreSQL supports `every`, but not `any` or `some`, because there is an ambiguity built into the standard syntax:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Here `ANY` can be considered either as introducing a subquery, or as being an aggregate function, if the subquery returns one row with a Boolean value. Thus the standard name cannot be given to these aggregates.

Note

Users accustomed to working with other SQL database management systems might be disappointed by the performance of the `count` aggregate when it is applied to the entire table. A query like:

```
SELECT count(*) FROM sometable;
```

will require effort proportional to the size of the table: PostgreSQL will need to scan either the entire table or the entirety of an index that includes all rows in the table.

Table 9.61 shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Functions shown as accepting *`numeric_type`* are available for all the types `smallint`, `integer`, `bigint`, `numeric`, `real`, and `double precision`. Where the description mentions *`N`*, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when *`N`* is zero.

Table 9.61. Aggregate Functions for Statistics

Function Description	Partial Mode
`corr(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Computes the correlation coefficient.	Yes
`covar_pop(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Computes the population covariance.	Yes
`covar_samp(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Computes the sample covariance.	Yes
`regr_avgx(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Computes the average of the independent variable, `sum(`*`X`*`)/`*`N`*.	Yes
`regr_avgy(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Computes the average of the dependent variable, `sum(`*`Y`*`)/`*`N`*.	Yes
`regr_count(`*`Y`* `double precision,` *`X`* `double precision)` → `bigint`	Yes

Function Description	Partial Mode
Computes the number of rows in which both inputs are non-null.	
`regr_intercept` (*Y* `double precision`, *X* `double precision`) → `double precision` Computes the y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs.	Yes
`regr_r2` (*Y* `double precision`, *X* `double precision`) → `double precision` Computes the square of the correlation coefficient.	Yes
`regr_slope` (*Y* `double precision`, *X* `double precision`) → `double precision` Computes the slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs.	Yes
`regr_sxx` (*Y* `double precision`, *X* `double precision`) → `double precision` Computes the “sum of squares” of the independent variable, `sum(`*X*`^2) - sum(`*X*`)^2/`*N*.	Yes
`regr_sxy` (*Y* `double precision`, *X* `double precision`) → `double precision` Computes the “sum of products” of independent times dependent variables, `sum(`*X*`*`*Y*`) - sum(`*X*`) * sum(`*Y*`)/`*N*.	Yes
`regr_syy` (*Y* `double precision`, *X* `double precision`) → `double precision` Computes the “sum of squares” of the dependent variable, `sum(`*Y*`^2) - sum(`*Y*`)^2/`*N*.	Yes
`stddev` (*numeric_type*) → `double precision` for `real` or `double precision`, otherwise `numeric` This is a historical alias for `stddev_samp`.	Yes
`stddev_pop` (*numeric_type*) → `double precision` for `real` or `double precision`, otherwise `numeric` Computes the population standard deviation of the input values.	Yes
`stddev_samp` (*numeric_type*) → `double precision` for `real` or `double precision`, otherwise `numeric` Computes the sample standard deviation of the input values.	Yes
`variance` (*numeric_type*) → `double precision` for `real` or `double precision`, otherwise `numeric` This is a historical alias for `var_samp`.	Yes
`var_pop` (*numeric_type*) → `double precision` for `real` or `double precision`, otherwise `numeric` Computes the population variance of the input values (square of the population standard deviation).	Yes
`var_samp` (*numeric_type*) → `double precision` for `real` or `double precision`, otherwise `numeric` Computes the sample variance of the input values (square of the sample standard deviation).	Yes

Table 9.62 shows some aggregate functions that use the *ordered-set aggregate* syntax. These functions are sometimes referred to as “inverse distribution” functions. Their aggregated input is introduced by

ORDER BY, and they may also take a *direct argument* that is not aggregated, but is computed only once. All these functions ignore null values in their aggregated input. For those that take a *fraction* parameter, the fraction value must be between 0 and 1; an error is thrown if not. However, a null *fraction* value simply produces a null result.

Table 9.62. Ordered-Set Aggregate Functions

Function Description	Partial Mode
mode () WITHIN GROUP (ORDER BY anyelement) → anyelement Computes the *mode*, the most frequent value of the aggregated argument (arbitrarily choosing the first one if there are multiple equally-frequent values). The aggregated argument must be of a sortable type.	No
percentile_cont (*fraction* double precision) WITHIN GROUP (ORDER BY double precision) → double precision percentile_cont (*fraction* double precision) WITHIN GROUP (ORDER BY interval) → interval Computes the *continuous percentile*, a value corresponding to the specified *fraction* within the ordered set of aggregated argument values. This will interpolate between adjacent input items if needed.	No
percentile_cont (*fractions* double precision[]) WITHIN GROUP (ORDER BY double precision) → double precision[] percentile_cont (*fractions* double precision[]) WITHIN GROUP (ORDER BY interval) → interval[] Computes multiple continuous percentiles. The result is an array of the same dimensions as the *fractions* parameter, with each non-null element replaced by the (possibly interpolated) value corresponding to that percentile.	No
percentile_disc (*fraction* double precision) WITHIN GROUP (ORDER BY anyelement) → anyelement Computes the *discrete percentile*, the first value within the ordered set of aggregated argument values whose position in the ordering equals or exceeds the specified *fraction*. The aggregated argument must be of a sortable type.	No
percentile_disc (*fractions* double precision[]) WITHIN GROUP (ORDER BY anyelement) → anyarray Computes multiple discrete percentiles. The result is an array of the same dimensions as the *fractions* parameter, with each non-null element replaced by the input value corresponding to that percentile. The aggregated argument must be of a sortable type.	No

Each of the "hypothetical-set" aggregates listed in Table 9.63 is associated with a window function of the same name defined in Section 9.22. In each case, the aggregate's result is the value that the associated window function would have returned for the "hypothetical" row constructed from *args*, if such a row had been added to the sorted group of rows represented by the *sorted_args*. For each of these functions, the list of direct arguments given in *args* must match the number and types of the aggregated arguments given in *sorted_args*. Unlike most built-in aggregates, these aggregates are not strict, that is they do not drop input rows containing nulls. Null values sort according to the rule specified in the ORDER BY clause.

Table 9.63. Hypothetical-Set Aggregate Functions

Function Description	Partial Mode
rank (*args*) WITHIN GROUP (ORDER BY *sorted_args*) → bigint	No

Function Description	Partial Mode
Computes the rank of the hypothetical row, with gaps; that is, the row number of the first row in its peer group.	
`dense_rank(` *`args`* `) WITHIN GROUP (ORDER BY` *`sorted_args`* `) → bigint` Computes the rank of the hypothetical row, without gaps; this function effectively counts peer groups.	No
`percent_rank(` *`args`* `) WITHIN GROUP (ORDER BY` *`sorted_args`* `) → double precision` Computes the relative rank of the hypothetical row, that is (`rank` - 1) / (total rows - 1). The value thus ranges from 0 to 1 inclusive.	No
`cume_dist(` *`args`* `) WITHIN GROUP (ORDER BY` *`sorted_args`* `) → double precision` Computes the cumulative distribution, that is (number of rows preceding or peers with hypothetical row) / (total rows). The value thus ranges from 1/*N* to 1.	No

Table 9.64. Grouping Operations

Function Description
`GROUPING(` *`group_by_expression(s)`* `) → integer` Returns a bit mask indicating which `GROUP BY` expressions are not included in the current grouping set. Bits are assigned with the rightmost argument corresponding to the least-significant bit; each bit is 0 if the corresponding expression is included in the grouping criteria of the grouping set generating the current result row, and 1 if it is not included.

The grouping operations shown in Table 9.64 are used in conjunction with grouping sets (see Section 7.2.4) to distinguish result rows. The arguments to the `GROUPING` function are not actually evaluated, but they must exactly match expressions given in the `GROUP BY` clause of the associated query level. For example:

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM
 items_sold GROUP BY ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

Here, the `grouping` value `0` in the first four rows shows that those have been grouped normally, over both the grouping columns. The value `1` indicates that `model` was not grouped by in the next-to-last two rows, and the value `3` indicates that neither `make` nor `model` was grouped by in the last row (which therefore is an aggregate over all the input rows).

9.22. Window Functions

Window functions provide the ability to perform calculations across sets of rows that are related to the current query row. See Section 3.5 for an introduction to this feature, and Section 4.2.8 for syntax details.

The built-in window functions are listed in Table 9.65. Note that these functions *must* be invoked using window function syntax, i.e., an `OVER` clause is required.

In addition to these functions, any built-in or user-defined ordinary aggregate (i.e., not ordered-set or hypothetical-set aggregates) can be used as a window function; see Section 9.21 for a list of the built-in aggregates. Aggregate functions act as window functions only when an `OVER` clause follows the call; otherwise they act as plain aggregates and return a single row for the entire set.

Table 9.65. General-Purpose Window Functions

Function Description
`row_number` () → `bigint` Returns the number of the current row within its partition, counting from 1.
`rank` () → `bigint` Returns the rank of the current row, with gaps; that is, the `row_number` of the first row in its peer group.
`dense_rank` () → `bigint` Returns the rank of the current row, without gaps; this function effectively counts peer groups.
`percent_rank` () → `double precision` Returns the relative rank of the current row, that is (`rank` - 1) / (total partition rows - 1). The value thus ranges from 0 to 1 inclusive.
`cume_dist` () → `double precision` Returns the cumulative distribution, that is (number of partition rows preceding or peers with current row) / (total partition rows). The value thus ranges from 1/*N* to 1.
`ntile` (*num_buckets* `integer`) → `integer` Returns an integer ranging from 1 to the argument value, dividing the partition as equally as possible.
`lag` (*value* `anycompatible` [, *offset* `integer` [, *default* `anycompatible`]]) → `anycompatible` Returns *value* evaluated at the row that is *offset* rows before the current row within the partition; if there is no such row, instead returns *default* (which must be of a type compatible with *value*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to 1 and *default* to `NULL`.
`lead` (*value* `anycompatible` [, *offset* `integer` [, *default* `anycompatible`]]) → `anycompatible` Returns *value* evaluated at the row that is *offset* rows after the current row within the partition; if there is no such row, instead returns *default* (which must be of a type compatible with *value*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to 1 and *default* to `NULL`.

Function Description
first_value (*value* anyelement) → anyelement Returns *value* evaluated at the row that is the first row of the window frame.
last_value (*value* anyelement) → anyelement Returns *value* evaluated at the row that is the last row of the window frame.
nth_value (*value* anyelement, *n* integer) → anyelement Returns *value* evaluated at the row that is the *n*'th row of the window frame (counting from 1); returns NULL if there is no such row.

All of the functions listed in Table 9.65 depend on the sort ordering specified by the ORDER BY clause of the associated window definition. Rows that are not distinct when considering only the ORDER BY columns are said to be *peers*. The four ranking functions (including cume_dist) are defined so that they give the same answer for all rows of a peer group.

Note that first_value, last_value, and nth_value consider only the rows within the "window frame", which by default contains the rows from the start of the partition through the last peer of the current row. This is likely to give unhelpful results for last_value and sometimes also nth_value. You can redefine the frame by adding a suitable frame specification (RANGE, ROWS or GROUPS) to the OVER clause. See Section 4.2.8 for more information about frame specifications.

When an aggregate function is used as a window function, it aggregates over the rows within the current row's window frame. An aggregate used with ORDER BY and the default window frame definition produces a "running sum" type of behavior, which may or may not be what's wanted. To obtain aggregation over the whole partition, omit ORDER BY or use ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Other frame specifications can be used to obtain other effects.

> **Note**
>
> The SQL standard defines a RESPECT NULLS or IGNORE NULLS option for lead, lag, first_value, last_value, and nth_value. This is not implemented in PostgreSQL: the behavior is always the same as the standard's default, namely RESPECT NULLS. Likewise, the standard's FROM FIRST or FROM LAST option for nth_value is not implemented: only the default FROM FIRST behavior is supported. (You can achieve the result of FROM LAST by reversing the ORDER BY ordering.)

9.23. Merge Support Functions

PostgreSQL includes one merge support function that may be used in the RETURNING list of a MERGE command to identify the action taken for each row; see Table 9.66.

Table 9.66. Merge Support Functions

Function Description
merge_action() → text Returns the merge action command executed for the current row. This will be 'INSERT', 'UPDATE', or 'DELETE'.

Example:

```
MERGE INTO products p
```

```
  USING stock s ON p.product_id = s.product_id
  WHEN MATCHED AND s.quantity > 0 THEN
    UPDATE SET in_stock = true, quantity = s.quantity
  WHEN MATCHED THEN
    UPDATE SET in_stock = false, quantity = 0
  WHEN NOT MATCHED THEN
    INSERT (product_id, in_stock, quantity)
      VALUES (s.product_id, true, s.quantity)
  RETURNING merge_action(), p.*;

 merge_action | product_id | in_stock | quantity
--------------+------------+----------+----------
 UPDATE       |       1001 | t        |       50
 UPDATE       |       1002 | f        |        0
 INSERT       |       1003 | t        |       10
```

Note that this function can only be used in the `RETURNING` list of a `MERGE` command. It is an error to use it in any other part of a query.

9.24. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

9.24.1. EXISTS

`EXISTS (`*`subquery`*`)`

The argument of `EXISTS` is an arbitrary `SELECT` statement, or *subquery*. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of `EXISTS` is “true”; if the subquery returns no rows, the result of `EXISTS` is “false”.

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed long enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has side effects (such as calling sequence functions); whether the side effects occur might be unpredictable.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally unimportant. A common coding convention is to write all `EXISTS` tests in the form `EXISTS(SELECT 1 WHERE ...)`. There are exceptions to this rule however, such as subqueries that use `INTERSECT`.

This simple example is like an inner join on `col2`, but it produces at most one output row for each `tab1` row, even if there are several matching `tab2` rows:

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.24.2. IN

`expression` `IN (`*`subquery`*`)`

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `IN` is “true”

if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the `IN` construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

row_constructor `IN (`*subquery*`)`

The left-hand side of this form of `IN` is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of `IN` is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of `IN` is null.

9.24.3. NOT IN

expression `NOT IN (`*subquery*`)`

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `NOT IN` is “true” if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is “false” if any equal row is found.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the `NOT IN` construct will be null, not true. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

row_constructor `NOT IN (`*subquery*`)`

The left-hand side of this form of `NOT IN` is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of `NOT IN` is “true” if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is “false” if any equal row is found.

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of `NOT IN` is null.

9.24.4. ANY/SOME

expression operator `ANY (`*subquery*`)`
expression operator `SOME (`*subquery*`)`

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ANY is "true" if any true result is obtained. The result is "false" if no true result is found (including the case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the ANY construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor operator ANY (*subquery*)
row_constructor operator SOME (*subquery*)

The left-hand side of this form of ANY is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. The result of ANY is "true" if the comparison returns true for any subquery row. The result is "false" if the comparison returns false for every subquery row (including the case where the subquery returns no rows). The result is NULL if no comparison with a subquery row returns true, and at least one comparison returns NULL.

See Section 9.25.5 for details about the meaning of a row constructor comparison.

9.24.5. ALL

expression operator ALL (*subquery*)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ALL is "true" if all rows yield true (including the case where the subquery returns no rows). The result is "false" if any false result is found. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

NOT IN is equivalent to <> ALL.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor operator ALL (*subquery*)

The left-hand side of this form of ALL is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. The result of ALL is "true" if the comparison returns true for all subquery rows (including the case where the subquery returns no rows). The result is "false" if the comparison returns false for any subquery row. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

See Section 9.25.5 for details about the meaning of a row constructor comparison.

9.24.6. Single-Row Comparison

row_constructor operator (*subquery*)

The left-hand side is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-

hand row. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be null.) The left-hand side is evaluated and compared row-wise to the single subquery result row.

See Section 9.25.5 for details about the meaning of a row constructor comparison.

9.25. Row and Array Comparisons

This section describes several specialized constructs for making multiple comparisons between groups of values. These forms are syntactically related to the subquery forms of the previous section, but do not involve subqueries. The forms involving array subexpressions are PostgreSQL extensions; the rest are SQL-compliant. All of the expression forms documented in this section return Boolean (true/false) results.

9.25.1. IN

```
expression IN (value [, ...])
```

The right-hand side is a parenthesized list of expressions. The result is “true” if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

```
expression = value1
OR
expression = value2
OR
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the `IN` construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

9.25.2. NOT IN

```
expression NOT IN (value [, ...])
```

The right-hand side is a parenthesized list of expressions. The result is “true” if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

```
expression <> value1
AND
expression <> value2
AND
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the `NOT IN` construct will be null, not true as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of null values.

> **Tip**
>
> `x NOT IN y` is equivalent to `NOT (x IN y)` in all cases. However, null values are much more likely to trip up the novice when working with `NOT IN` than when working with `IN`. It is best to express your condition positively if possible.

9.25.3. ANY/SOME (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given *operator*, which must yield a Boolean result. The result of ANY is “true” if any true result is obtained. The result is “false” if no true result is found (including the case where the array has zero elements).

If the array expression yields a null array, the result of ANY will be null. If the left-hand expression yields null, the result of ANY is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no true comparison result is obtained, the result of ANY will be null, not false (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

SOME is a synonym for ANY.

9.25.4. ALL (array)

```
expression operator ALL (array expression)
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given *operator*, which must yield a Boolean result. The result of ALL is “true” if all comparisons yield true (including the case where the array has zero elements). The result is “false” if any false result is found.

If the array expression yields a null array, the result of ALL will be null. If the left-hand expression yields null, the result of ALL is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of ALL will be null, not true (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

9.25.5. Row Constructor Comparison

```
row_constructor operator row_constructor
```

Each side is a row constructor, as described in Section 4.2.13. The two row constructors must have the same number of fields. The given *operator* is applied to each pair of corresponding fields. (Since the fields could be of different types, this means that a different specific operator could be selected for each pair.) All the selected operators must be members of some B-tree operator class, or be the negator of an = member of a B-tree operator class, meaning that row constructor comparison is only possible when the *operator* is =, <>, <, <=, >, or >=, or has semantics similar to one of these.

The = and <> cases work slightly differently from the others. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (null).

For the <, <=, > and >= cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found. If either of this pair of elements is null, the result of the row comparison is unknown (null); otherwise comparison of this pair of elements determines the result. For example, `ROW(1,2,NULL) < ROW(1,3,0)` yields true, not null, because the third pair of elements are not considered.

```
row_constructor IS DISTINCT FROM row_constructor
```

This construct is similar to a <> row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will either be true or false, never null.

row_constructor IS NOT DISTINCT FROM *row_constructor*

This construct is similar to a = row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

9.25.6. Composite Type Comparison

record operator record

The SQL specification requires row-wise comparison to return NULL if the result depends on comparing two NULL values or a NULL and a non-NULL. PostgreSQL does this only when comparing the results of two row constructors (as in Section 9.25.5) or comparing a row constructor to the output of a subquery (as in Section 9.24). In other contexts where two composite-type values are compared, two NULL field values are considered equal, and a NULL is considered larger than a non-NULL. This is necessary in order to have consistent sorting and indexing behavior for composite types.

Each side is evaluated and they are compared row-wise. Composite type comparisons are allowed when the *operator* is =, <>, <, <=, > or >=, or has semantics similar to one of these. (To be specific, an operator can be a row comparison operator if it is a member of a B-tree operator class, or is the negator of the = member of a B-tree operator class.) The default behavior of the above operators is the same as for IS [NOT] DISTINCT FROM for row constructors (see Section 9.25.5).

To support matching of rows which include elements without a default B-tree operator class, the following operators are defined for composite type comparison: *=, *<>, *<, *<=, *>, and *>=. These operators compare the internal binary representation of the two rows. Two rows might have a different binary representation even though comparisons of the two rows with the equality operator is true. The ordering of rows under these comparison operators is deterministic but not otherwise meaningful. These operators are used internally for materialized views and might be useful for other specialized purposes such as replication and B-Tree deduplication (see Section 64.1.4.3). They are not intended to be generally useful for writing queries, though.

9.26. Set Returning Functions

This section describes functions that possibly return more than one row. The most widely used functions in this class are series generating functions, as detailed in Table 9.67 and Table 9.68. Other, more specialized set-returning functions are described elsewhere in this manual. See Section 7.2.1.4 for ways to combine multiple set-returning functions.

Table 9.67. Series Generating Functions

Function Description
generate_series (*start* integer, *stop* integer [, *step* integer]) → setof integer generate_series (*start* bigint, *stop* bigint [, *step* bigint]) → setof bigint generate_series (*start* numeric, *stop* numeric [, *step* numeric]) → setof numeric Generates a series of values from *start* to *stop*, with a step size of *step*. *step* defaults to 1.

Function Description
generate_series(*start* timestamp, *stop* timestamp, *step* interval) → setof timestamp generate_series(*start* timestamp with time zone, *stop* timestamp with time zone, *step* interval [, *timezone* text]) → setof timestamp with time zone Generates a series of values from *start* to *stop*, with a step size of *step*. In the time-zone-aware form, times of day and daylight-savings adjustments are computed according to the time zone named by the *timezone* argument, or the current TimeZone setting if that is omitted.

When *step* is positive, zero rows are returned if *start* is greater than *stop*. Conversely, when *step* is negative, zero rows are returned if *start* is less than *stop*. Zero rows are also returned if any input is NULL. It is an error for *step* to be zero. Some examples follow:

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- this example relies on the date-plus-integer operator:
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS
 s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
```

```
                                '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)

-- this example assumes that TimeZone is set to UTC; note the DST
 transition:
SELECT * FROM generate_series('2001-10-22 00:00
 -04:00'::timestamptz,
                                '2001-11-01 00:00
 -05:00'::timestamptz,
                                '1 day'::interval, 'America/
New_York');
    generate_series
------------------------
 2001-10-22 04:00:00+00
 2001-10-23 04:00:00+00
 2001-10-24 04:00:00+00
 2001-10-25 04:00:00+00
 2001-10-26 04:00:00+00
 2001-10-27 04:00:00+00
 2001-10-28 04:00:00+00
 2001-10-29 05:00:00+00
 2001-10-30 05:00:00+00
 2001-10-31 05:00:00+00
 2001-11-01 05:00:00+00
(11 rows)
```

Table 9.68. Subscript Generating Functions

Function Description
`generate_subscripts` (*`array`* `anyarray`, *`dim`* `integer`) → `setof integer` Generates a series comprising the valid subscripts of the *`dim`*'th dimension of the given array.
`generate_subscripts` (*`array`* `anyarray`, *`dim`* `integer`, *`reverse`* `boolean`) → `setof integer` Generates a series comprising the valid subscripts of the *`dim`*'th dimension of the given array. When *`reverse`* is true, returns the series in reverse order.

`generate_subscripts` is a convenience function that generates the set of valid subscripts for the specified dimension of the given array. Zero rows are returned for arrays that do not have the requested dimension, or if any input is `NULL`. Some examples follow:

```
-- basic usage:
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
```

```
---
 1
 2
 3
 4
(4 rows)

-- presenting an array, the subscript and the subscripted
-- value requires a subquery:
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- unnest a 2D array:
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

When a function in the `FROM` clause is suffixed by `WITH ORDINALITY`, a `bigint` column is appended to the function's output column(s), which starts from 1 and increments by 1 for each row of the function's output. This is most useful in the case of set returning functions such as `unnest()`.

```
-- set returning function WITH ORDINALITY:
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
-----------------+----
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
```

```
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
 pg_ident.conf   | 10
 global          | 11
 pg_xact         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
 pg_wal          | 16
 pg_hba.conf     | 17
 pg_stat_tmp     | 18
 pg_subtrans     | 19
(19 rows)
```

9.27. System Information Functions and Operators

The functions described in this section are used to obtain various information about a PostgreSQL installation.

9.27.1. Session Information Functions

Table 9.69 shows several functions that extract session and system information.

In addition to the functions listed in this section, there are a number of functions related to the statistics system that also provide system information. See Section 27.2.26 for more information.

Table 9.69. Session Information Functions

Function **Description**

`current_catalog` → `name`
`current_database` () → `name`
Returns the name of the current database. (Databases are called “catalogs” in the SQL standard, so `current_catalog` is the standard's spelling.)

`current_query` () → `text` Returns the text of the currently executing query, as submitted by the client (which might contain more than one statement).
`current_role` → `name` This is equivalent to `current_user`.

`current_schema` → `name`
`current_schema` () → `name`
Returns the name of the schema that is first in the search path (or a null value if the search path is empty). This is the schema that will be used for any tables or other named objects that are created without specifying a target schema.

`current_schemas` (*`include_implicit`* `boolean`) → `name[]` Returns an array of the names of all schemas presently in the effective search path, in their priority order. (Items in the current search_path setting that do not correspond to existing, searchable schemas are omitted.) If the Boolean argument is `true`, then implicitly-searched system schemas such as `pg_catalog` are included in the result.

Function Description
current_user → name Returns the user name of the current execution context.
inet_client_addr () → inet Returns the IP address of the current client, or NULL if the current connection is via a Unix-domain socket.
inet_client_port () → integer Returns the IP port number of the current client, or NULL if the current connection is via a Unix-domain socket.
inet_server_addr () → inet Returns the IP address on which the server accepted the current connection, or NULL if the current connection is via a Unix-domain socket.
inet_server_port () → integer Returns the IP port number on which the server accepted the current connection, or NULL if the current connection is via a Unix-domain socket.
pg_backend_pid () → integer Returns the process ID of the server process attached to the current session.
pg_blocking_pids (integer) → integer[] Returns an array of the process ID(s) of the sessions that are blocking the server process with the specified process ID from acquiring a lock, or an empty array if there is no such server process or it is not blocked. One server process blocks another if it either holds a lock that conflicts with the blocked process's lock request (hard block), or is waiting for a lock that would conflict with the blocked process's lock request and is ahead of it in the wait queue (soft block). When using parallel queries the result always lists client-visible process IDs (that is, pg_backend_pid results) even if the actual lock is held or awaited by a child worker process. As a result of that, there may be duplicated PIDs in the result. Also note that when a prepared transaction holds a conflicting lock, it will be represented by a zero process ID. Frequent calls to this function could have some impact on database performance, because it needs exclusive access to the lock manager's shared state for a short time.
pg_conf_load_time () → timestamp with time zone Returns the time when the server configuration files were last loaded. If the current session was alive at the time, this will be the time when the session itself re-read the configuration files (so the reading will vary a little in different sessions). Otherwise it is the time when the postmaster process re-read the configuration files.
pg_current_logfile ([text]) → text Returns the path name of the log file currently in use by the logging collector. The path includes the log_directory directory and the individual log file name. The result is NULL if the logging collector is disabled. When multiple log files exist, each in a different format, pg_current_logfile without an argument returns the path of the file having the first format found in the ordered list: stderr, csvlog, jsonlog. NULL is returned if no log file has any of these formats. To request information about a specific log file format, supply either csvlog, jsonlog or stderr as the value of the optional parameter. The result is NULL if the log format requested is not configured in log_destination. The result reflects the contents of the current_logfiles file. This function is restricted to superusers and roles with privileges of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_my_temp_schema () → oid

Function Description
Returns the OID of the current session's temporary schema, or zero if it has none (because it has not created any temporary tables).
`pg_is_other_temp_schema` (`oid`) → `boolean` Returns true if the given OID is the OID of another session's temporary schema. (This can be useful, for example, to exclude other sessions' temporary tables from a catalog display.)
`pg_jit_available` () → `boolean` Returns true if a JIT compiler extension is available (see Chapter 30) and the jit configuration parameter is set to `on`.
`pg_listening_channels` () → `setof text` Returns the set of names of asynchronous notification channels that the current session is listening to.
`pg_notification_queue_usage` () → `double precision` Returns the fraction (0–1) of the asynchronous notification queue's maximum size that is currently occupied by notifications that are waiting to be processed. See LISTEN and NOTIFY for more information.
`pg_postmaster_start_time` () → `timestamp with time zone` Returns the time when the server started.
`pg_safe_snapshot_blocking_pids` (`integer`) → `integer[]` Returns an array of the process ID(s) of the sessions that are blocking the server process with the specified process ID from acquiring a safe snapshot, or an empty array if there is no such server process or it is not blocked. A session running a `SERIALIZABLE` transaction blocks a `SERIALIZABLE READ ONLY DEFERRABLE` transaction from acquiring a snapshot until the latter determines that it is safe to avoid taking any predicate locks. See Section 13.2.3 for more information about serializable and deferrable transactions. Frequent calls to this function could have some impact on database performance, because it needs access to the predicate lock manager's shared state for a short time.
`pg_trigger_depth` () → `integer` Returns the current nesting level of PostgreSQL triggers (0 if not called, directly or indirectly, from inside a trigger).
`session_user` → `name` Returns the session user's name.
`system_user` → `text` Returns the authentication method and the identity (if any) that the user presented during the authentication cycle before they were assigned a database role. It is represented as `auth_method:identity` or `NULL` if the user has not been authenticated (for example if Trust authentication has been used).
`user` → `name` This is equivalent to `current_user`.

Note

`current_catalog`, `current_role`, `current_schema`, `current_user`, `session_user`, and `user` have special syntactic status in SQL: they must be called with-

out trailing parentheses. In PostgreSQL, parentheses can optionally be used with `current_schema`, but not with the others.

The `session_user` is normally the user who initiated the current database connection; but superusers can change this setting with SET SESSION AUTHORIZATION. The `current_user` is the user identifier that is applicable for permission checking. Normally it is equal to the session user, but it can be changed with SET ROLE. It also changes during the execution of functions with the attribute `SECURITY DEFINER`. In Unix parlance, the session user is the "real user" and the current user is the "effective user". `current_role` and `user` are synonyms for `current_user`. (The SQL standard draws a distinction between `current_role` and `current_user`, but PostgreSQL does not, since it unifies users and roles into a single kind of entity.)

9.27.2. Access Privilege Inquiry Functions

Table 9.70 lists functions that allow querying object access privileges programmatically. (See Section 5.8 for more information about privileges.) In these functions, the user whose privileges are being inquired about can be specified by name or by OID (`pg_authid.oid`), or if the name is given as `public` then the privileges of the PUBLIC pseudo-role are checked. Also, the *`user`* argument can be omitted entirely, in which case the `current_user` is assumed. The object that is being inquired about can be specified either by name or by OID, too. When specifying by name, a schema name can be included if relevant. The access privilege of interest is specified by a text string, which must evaluate to one of the appropriate privilege keywords for the object's type (e.g., `SELECT`). Optionally, `WITH GRANT OPTION` can be added to a privilege type to test whether the privilege is held with grant option. Also, multiple privilege types can be listed separated by commas, in which case the result will be true if any of the listed privileges is held. (Case of the privilege string is not significant, and extra whitespace is allowed between but not within privilege names.) Some examples:

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH
 GRANT OPTION');
```

Table 9.70. Access Privilege Inquiry Functions

Function Description
`has_any_column_privilege` ([*`user`* `name` or `oid`,] *`table`* `text` or `oid`, *`privilege`* `text`) → `boolean` Does user have privilege for any column of table? This succeeds either if the privilege is held for the whole table, or if there is a column-level grant of the privilege for at least one column. Allowable privilege types are `SELECT`, `INSERT`, `UPDATE`, and `REFERENCES`.
`has_column_privilege` ([*`user`* `name` or `oid`,] *`table`* `text` or `oid`, *`column`* `text` or `smallint`, *`privilege`* `text`) → `boolean` Does user have privilege for the specified table column? This succeeds either if the privilege is held for the whole table, or if there is a column-level grant of the privilege for the column. The column can be specified by name or by attribute number (`pg_attribute.attnum`). Allowable privilege types are `SELECT`, `INSERT`, `UPDATE`, and `REFERENCES`.
`has_database_privilege` ([*`user`* `name` or `oid`,] *`database`* `text` or `oid`, *`privilege`* `text`) → `boolean` Does user have privilege for database? Allowable privilege types are `CREATE`, `CONNECT`, `TEMPORARY`, and `TEMP` (which is equivalent to `TEMPORARY`).
`has_foreign_data_wrapper_privilege` ([*`user`* `name` or `oid`,] *`fdw`* `text` or `oid`, *`privilege`* `text`) → `boolean`

Function Description
Does user have privilege for foreign-data wrapper? The only allowable privilege type is USAGE.
has_function_privilege ([*user* name or oid,] *function* text or oid, *privilege* text) → boolean Does user have privilege for function? The only allowable privilege type is EXECUTE. When specifying a function by name rather than by OID, the allowed input is the same as for the regprocedure data type (see Section 8.19). An example is: `SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');`
has_language_privilege ([*user* name or oid,] *language* text or oid, *privilege* text) → boolean Does user have privilege for language? The only allowable privilege type is USAGE.
has_parameter_privilege ([*user* name or oid,] *parameter* text, *privilege* text) → boolean Does user have privilege for configuration parameter? The parameter name is case-insensitive. Allowable privilege types are SET and ALTER SYSTEM.
has_schema_privilege ([*user* name or oid,] *schema* text or oid, *privilege* text) → boolean Does user have privilege for schema? Allowable privilege types are CREATE and USAGE.
has_sequence_privilege ([*user* name or oid,] *sequence* text or oid, *privilege* text) → boolean Does user have privilege for sequence? Allowable privilege types are USAGE, SELECT, and UPDATE.
has_server_privilege ([*user* name or oid,] *server* text or oid, *privilege* text) → boolean Does user have privilege for foreign server? The only allowable privilege type is USAGE.
has_table_privilege ([*user* name or oid,] *table* text or oid, *privilege* text) → boolean Does user have privilege for table? Allowable privilege types are SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, and MAINTAIN.
has_tablespace_privilege ([*user* name or oid,] *tablespace* text or oid, *privilege* text) → boolean Does user have privilege for tablespace? The only allowable privilege type is CREATE.
has_type_privilege ([*user* name or oid,] *type* text or oid, *privilege* text) → boolean Does user have privilege for data type? The only allowable privilege type is USAGE. When specifying a type by name rather than by OID, the allowed input is the same as for the regtype data type (see Section 8.19).
pg_has_role ([*user* name or oid,] *role* text or oid, *privilege* text) → boolean Does user have privilege for role? Allowable privilege types are MEMBER, USAGE, and SET. MEMBER denotes direct or indirect membership in the role without regard to what specific privileges may be conferred. USAGE denotes whether the privileges of the role are immediately available without doing SET ROLE, while SET denotes whether it is possible to change to the role using the SET ROLE command. WITH ADMIN OPTION

Function Description
or `WITH GRANT OPTION` can be added to any of these privilege types to test whether the `ADMIN` privilege is held (all six spellings test the same thing). This function does not allow the special case of setting *`user`* to `public`, because the PUBLIC pseudo-role can never be a member of real roles.
`row_security_active` (*`table`* `text` or `oid`) → `boolean` Is row-level security active for the specified table in the context of the current user and current environment?

Table 9.71 shows the operators available for the `aclitem` type, which is the catalog representation of access privileges. See Section 5.8 for information about how to read access privilege values.

Table 9.71. `aclitem` Operators

Operator Description Example(s)

`aclitem` = `aclitem` → `boolean`
Are `aclitem`s equal? (Notice that type `aclitem` lacks the usual set of comparison operators; it has only equality. In turn, `aclitem` arrays can only be compared for equality.)
`'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem` → `f`

`aclitem[]` @> `aclitem` → `boolean`
Does array contain the specified privileges? (This is true if there is an array entry that matches the `aclitem`'s grantee and grantor, and has at least the specified set of privileges.)
`'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] @> 'calvin=r*/hobbes'::aclitem` → `t`

`aclitem[]` ~ `aclitem` → `boolean`
This is a deprecated alias for @>.
`'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] ~ 'calvin=r*/hobbes'::aclitem` → `t`

Table 9.72 shows some additional functions to manage the `aclitem` type.

Table 9.72. `aclitem` Functions

Function Description
`acldefault` (*`type`* `"char"`, *`ownerId`* `oid`) → `aclitem[]` Constructs an `aclitem` array holding the default access privileges for an object of type *`type`* belonging to the role with OID *`ownerId`*. This represents the access privileges that will be assumed when an object's ACL entry is null. (The default access privileges are described in Section 5.8.) The *`type`* parameter must be one of 'c' for `COLUMN`, 'r' for `TABLE` and table-like objects, 's' for `SEQUENCE`, 'd' for `DATABASE`, 'f' for `FUNCTION` or `PROCEDURE`, 'l' for `LANGUAGE`, 'L' for `LARGE OBJECT`, 'n' for `SCHEMA`, 'p' for `PARAMETER`, 't' for `TABLESPACE`, 'F' for `FOREIGN DATA WRAPPER`, 'S' for `FOREIGN SERVER`, or 'T' for `TYPE` or `DOMAIN`.
`aclexplode` (`aclitem[]`) → `setof record` (*`grantor`* `oid`, *`grantee`* `oid`, *`privilege_type`* `text`, *`is_grantable`* `boolean`)

Function Description
Returns the `aclitem` array as a set of rows. If the grantee is the pseudo-role PUBLIC, it is represented by zero in the *`grantee`* column. Each granted privilege is represented as `SELECT`, `INSERT`, etc (see Table 5.1 for a full list). Note that each privilege is broken out as a separate row, so only one keyword appears in the *`privilege_type`* column.
`makeaclitem` (*`grantee`* `oid`, *`grantor`* `oid`, *`privileges`* `text`, *`is_grantable`* `boolean`) → `aclitem` Constructs an `aclitem` with the given properties. *`privileges`* is a comma-separated list of privilege names such as `SELECT`, `INSERT`, etc, all of which are set in the result. (Case of the privilege string is not significant, and extra whitespace is allowed between but not within privilege names.)

9.27.3. Schema Visibility Inquiry Functions

Table 9.73 shows functions that determine whether a certain object is *visible* in the current schema search path. For example, a table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement that the table can be referenced by name without explicit schema qualification. Thus, to list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

For functions and operators, an object in the search path is said to be visible if there is no object of the same name *and argument data type(s)* earlier in the path. For operator classes and families, both the name and the associated index access method are considered.

Table 9.73. Schema Visibility Inquiry Functions

Function Description
`pg_collation_is_visible` (*`collation`* `oid`) → `boolean` Is collation visible in search path?
`pg_conversion_is_visible` (*`conversion`* `oid`) → `boolean` Is conversion visible in search path?
`pg_function_is_visible` (*`function`* `oid`) → `boolean` Is function visible in search path? (This also works for procedures and aggregates.)
`pg_opclass_is_visible` (*`opclass`* `oid`) → `boolean` Is operator class visible in search path?
`pg_operator_is_visible` (*`operator`* `oid`) → `boolean` Is operator visible in search path?
`pg_opfamily_is_visible` (*`opclass`* `oid`) → `boolean` Is operator family visible in search path?
`pg_statistics_obj_is_visible` (*`stat`* `oid`) → `boolean` Is statistics object visible in search path?
`pg_table_is_visible` (*`table`* `oid`) → `boolean` Is table visible in search path? (This works for all types of relations, including views, materialized views, indexes, sequences and foreign tables.)
`pg_ts_config_is_visible` (*`config`* `oid`) → `boolean`

Function Description
Is text search configuration visible in search path?
`pg_ts_dict_is_visible` (*`dict`* `oid`) → `boolean` Is text search dictionary visible in search path?
`pg_ts_parser_is_visible` (*`parser`* `oid`) → `boolean` Is text search parser visible in search path?
`pg_ts_template_is_visible` (*`template`* `oid`) → `boolean` Is text search template visible in search path?
`pg_type_is_visible` (*`type`* `oid`) → `boolean` Is type (or domain) visible in search path?

All these functions require object OIDs to identify the object to be checked. If you want to test an object by name, it is convenient to use the OID alias types (`regclass`, `regtype`, `regprocedure`, `regoperator`, `regconfig`, or `regdictionary`), for example:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Note that it would not make much sense to test a non-schema-qualified type name in this way — if the name can be recognized at all, it must be visible.

9.27.4. System Catalog Information Functions

Table 9.74 lists functions that extract information from the system catalogs.

Table 9.74. System Catalog Information Functions

Function Description
`format_type` (*`type`* `oid`, *`typemod`* `integer`) → `text` Returns the SQL name for a data type that is identified by its type OID and possibly a type modifier. Pass NULL for the type modifier if no specific modifier is known.

`pg_basetype` (`regtype`) → `regtype`
Returns the OID of the base type of a domain identified by its type OID. If the argument is the OID of a non-domain type, returns the argument as-is. Returns NULL if the argument is not a valid type OID. If there's a chain of domain dependencies, it will recurse until finding the base type.
Assuming `CREATE DOMAIN mytext AS text`:
`pg_basetype('mytext'::regtype)` → `text`

`pg_char_to_encoding` (*`encoding`* `name`) → `integer` Converts the supplied encoding name into an integer representing the internal identifier used in some system catalog tables. Returns `-1` if an unknown encoding name is provided.
`pg_encoding_to_char` (*`encoding`* `integer`) → `name` Converts the integer used as the internal identifier of an encoding in some system catalog tables into a human-readable string. Returns an empty string if an invalid encoding number is provided.
`pg_get_catalog_foreign_keys` () → `setof record` (*`fktable`* `regclass`, *`fkcols`* `text[]`, *`pktable`* `regclass`, *`pkcols`* `text[]`, *`is_array`* `boolean`, *`is_opt`* `boolean`)

Function Description
Returns a set of records describing the foreign key relationships that exist within the PostgreSQL system catalogs. The *fktable* column contains the name of the referencing catalog, and the *fkcols* column contains the name(s) of the referencing column(s). Similarly, the *pktable* column contains the name of the referenced catalog, and the *pkcols* column contains the name(s) of the referenced column(s). If *is_array* is true, the last referencing column is an array, each of whose elements should match some entry in the referenced catalog. If *is_opt* is true, the referencing column(s) are allowed to contain zeroes instead of a valid reference.
pg_get_constraintdef (*constraint* oid [, *pretty* boolean]) → text Reconstructs the creating command for a constraint. (This is a decompiled reconstruction, not the original text of the command.)
pg_get_expr (*expr* pg_node_tree, *relation* oid [, *pretty* boolean]) → text Decompiles the internal form of an expression stored in the system catalogs, such as the default value for a column. If the expression might contain Vars, specify the OID of the relation they refer to as the second parameter; if no Vars are expected, passing zero is sufficient.
pg_get_functiondef (*func* oid) → text Reconstructs the creating command for a function or procedure. (This is a decompiled reconstruction, not the original text of the command.) The result is a complete CREATE OR REPLACE FUNCTION or CREATE OR REPLACE PROCEDURE statement.
pg_get_function_arguments (*func* oid) → text Reconstructs the argument list of a function or procedure, in the form it would need to appear in within CREATE FUNCTION (including default values).
pg_get_function_identity_arguments (*func* oid) → text Reconstructs the argument list necessary to identify a function or procedure, in the form it would need to appear in within commands such as ALTER FUNCTION. This form omits default values.
pg_get_function_result (*func* oid) → text Reconstructs the RETURNS clause of a function, in the form it would need to appear in within CREATE FUNCTION. Returns NULL for a procedure.
pg_get_indexdef (*index* oid [, *column* integer, *pretty* boolean]) → text Reconstructs the creating command for an index. (This is a decompiled reconstruction, not the original text of the command.) If *column* is supplied and is not zero, only the definition of that column is reconstructed.
pg_get_keywords () → setof record (*word* text, *catcode* "char", *barelabel* boolean, *catdesc* text, *baredesc* text) Returns a set of records describing the SQL keywords recognized by the server. The *word* column contains the keyword. The *catcode* column contains a category code: U for an unreserved keyword, C for a keyword that can be a column name, T for a keyword that can be a type or function name, or R for a fully reserved keyword. The *barelabel* column contains true if the keyword can be used as a “bare” column label in SELECT lists, or false if it can only be used after AS. The *catdesc* column contains a possibly-localized string describing the keyword's category. The *baredesc* column contains a possibly-localized string describing the keyword's column label status.
pg_get_partkeydef (*table* oid) → text Reconstructs the definition of a partitioned table's partition key, in the form it would have in the PARTITION BY clause of CREATE TABLE. (This is a decompiled reconstruction, not the original text of the command.)

Function Description
`pg_get_ruledef` (*`rule`* `oid` [, *`pretty`* `boolean`]) → `text` Reconstructs the creating command for a rule. (This is a decompiled reconstruction, not the original text of the command.)

`pg_get_serial_sequence` (*`table`* `text`, *`column`* `text`) → `text`
Returns the name of the sequence associated with a column, or NULL if no sequence is associated with the column. If the column is an identity column, the associated sequence is the sequence internally created for that column. For columns created using one of the serial types (`serial`, `smallserial`, `bigserial`), it is the sequence created for that serial column definition. In the latter case, the association can be modified or removed with `ALTER SEQUENCE OWNED BY`. (This function probably should have been called `pg_get_owned_sequence`; its current name reflects the fact that it has historically been used with serial-type columns.) The first parameter is a table name with optional schema, and the second parameter is a column name. Because the first parameter potentially contains both schema and table names, it is parsed per usual SQL rules, meaning it is lower-cased by default. The second parameter, being just a column name, is treated literally and so has its case preserved. The result is suitably formatted for passing to the sequence functions (see Section 9.17).
A typical use is in reading the current value of the sequence for an identity or serial column, for example:
`SELECT currval(pg_get_serial_sequence('sometable', 'id'));`

`pg_get_statisticsobjdef` (*`statobj`* `oid`) → `text` Reconstructs the creating command for an extended statistics object. (This is a decompiled reconstruction, not the original text of the command.)
`pg_get_triggerdef` (*`trigger`* `oid` [, *`pretty`* `boolean`]) → `text` Reconstructs the creating command for a trigger. (This is a decompiled reconstruction, not the original text of the command.)
`pg_get_userbyid` (*`role`* `oid`) → `name` Returns a role's name given its OID.
`pg_get_viewdef` (*`view`* `oid` [, *`pretty`* `boolean`]) → `text` Reconstructs the underlying `SELECT` command for a view or materialized view. (This is a decompiled reconstruction, not the original text of the command.)
`pg_get_viewdef` (*`view`* `oid`, *`wrap_column`* `integer`) → `text` Reconstructs the underlying `SELECT` command for a view or materialized view. (This is a decompiled reconstruction, not the original text of the command.) In this form of the function, pretty-printing is always enabled, and long lines are wrapped to try to keep them shorter than the specified number of columns.
`pg_get_viewdef` (*`view`* `text` [, *`pretty`* `boolean`]) → `text` Reconstructs the underlying `SELECT` command for a view or materialized view, working from a textual name for the view rather than its OID. (This is deprecated; use the OID variant instead.)
`pg_index_column_has_property` (*`index`* `regclass`, *`column`* `integer`, *`property`* `text`) → `boolean` Tests whether an index column has the named property. Common index column properties are listed in Table 9.75. (Note that extension access methods can define additional property names for their indexes.) `NULL` is returned if the property name is not known or does not apply to the particular object, or if the OID or column number does not identify a valid object.

Function Description
`pg_index_has_property` (*`index`* `regclass`, *`property`* `text`) → `boolean` Tests whether an index has the named property. Common index properties are listed in Table 9.76. (Note that extension access methods can define additional property names for their indexes.) `NULL` is returned if the property name is not known or does not apply to the particular object, or if the OID does not identify a valid object.
`pg_indexam_has_property` (*`am`* `oid`, *`property`* `text`) → `boolean` Tests whether an index access method has the named property. Access method properties are listed in Table 9.77. `NULL` is returned if the property name is not known or does not apply to the particular object, or if the OID does not identify a valid object.
`pg_options_to_table` (*`options_array`* `text[]`) → `setof record` (*`option_name`* `text`, *`option_value`* `text`) Returns the set of storage options represented by a value from `pg_class.reloptions` or `pg_attribute.attoptions`.
`pg_settings_get_flags` (*`guc`* `text`) → `text[]` Returns an array of the flags associated with the given GUC, or `NULL` if it does not exist. The result is an empty array if the GUC exists but there are no flags to show. Only the most useful flags listed in Table 9.78 are exposed.
`pg_tablespace_databases` (*`tablespace`* `oid`) → `setof oid` Returns the set of OIDs of databases that have objects stored in the specified tablespace. If this function returns any rows, the tablespace is not empty and cannot be dropped. To identify the specific objects populating the tablespace, you will need to connect to the database(s) identified by `pg_tablespace_databases` and query their `pg_class` catalogs.
`pg_tablespace_location` (*`tablespace`* `oid`) → `text` Returns the file system path that this tablespace is located in.

`pg_typeof` (`"any"`) → `regtype`
Returns the OID of the data type of the value that is passed to it. This can be helpful for troubleshooting or dynamically constructing SQL queries. The function is declared as returning `regtype`, which is an OID alias type (see Section 8.19); this means that it is the same as an OID for comparison purposes but displays as a type name.
`pg_typeof(33)` → `integer`

`COLLATION FOR` (`"any"`) → `text`
Returns the name of the collation of the value that is passed to it. The value is quoted and schema-qualified if necessary. If no collation was derived for the argument expression, then `NULL` is returned. If the argument is not of a collatable data type, then an error is raised.
`collation for ('foo'::text)` → `"default"`
`collation for ('foo' COLLATE "de_DE")` → `"de_DE"`

`to_regclass` (`text`) → `regclass` Translates a textual relation name to its OID. A similar result is obtained by casting the string to type `regclass` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found.
`to_regcollation` (`text`) → `regcollation` Translates a textual collation name to its OID. A similar result is obtained by casting the string to type `regcollation` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found.
`to_regnamespace` (`text`) → `regnamespace`

Function Description
Translates a textual schema name to its OID. A similar result is obtained by casting the string to type `regnamespace` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found.
`to_regoper ( text ) → regoper` Translates a textual operator name to its OID. A similar result is obtained by casting the string to type `regoper` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found or is ambiguous.
`to_regoperator ( text ) → regoperator` Translates a textual operator name (with parameter types) to its OID. A similar result is obtained by casting the string to type `regoperator` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found.
`to_regproc ( text ) → regproc` Translates a textual function or procedure name to its OID. A similar result is obtained by casting the string to type `regproc` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found or is ambiguous.
`to_regprocedure ( text ) → regprocedure` Translates a textual function or procedure name (with argument types) to its OID. A similar result is obtained by casting the string to type `regprocedure` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found.
`to_regrole ( text ) → regrole` Translates a textual role name to its OID. A similar result is obtained by casting the string to type `regrole` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found.
`to_regtype ( text ) → regtype` Parses a string of text, extracts a potential type name from it, and translates that name into a type OID. A syntax error in the string will result in an error; but if the string is a syntactically valid type name that happens not to be found in the catalogs, the result is `NULL`. A similar result is obtained by casting the string to type `regtype` (see Section 8.19), except that that will throw error for name not found.

`to_regtypemod ( text ) → integer`
Parses a string of text, extracts a potential type name from it, and translates its type modifier, if any. A syntax error in the string will result in an error; but if the string is a syntactically valid type name that happens not to be found in the catalogs, the result is `NULL`. The result is `-1` if no type modifier is present.
`to_regtypemod` can be combined with to_regtype to produce appropriate inputs for format_type, allowing a string representing a type name to be canonicalized.
`format_type(to_regtype('varchar(32)'), to_regtypemod('varchar(32)')) → character varying(32)`

Most of the functions that reconstruct (decompile) database objects have an optional *`pretty`* flag, which if `true` causes the result to be “pretty-printed”. Pretty-printing suppresses unnecessary parentheses and adds whitespace for legibility. The pretty-printed format is more readable, but the default format is more likely to be interpreted the same way by future versions of PostgreSQL; so avoid using pretty-printed output for dump purposes. Passing `false` for the *`pretty`* parameter yields the same result as omitting the parameter.

Table 9.75. Index Column Properties

Name	Description
`asc`	Does the column sort in ascending order on a forward scan?
`desc`	Does the column sort in descending order on a forward scan?
`nulls_first`	Does the column sort with nulls first on a forward scan?
`nulls_last`	Does the column sort with nulls last on a forward scan?
`orderable`	Does the column possess any defined sort ordering?
`distance_orderable`	Can the column be scanned in order by a "distance" operator, for example `ORDER BY col <-> constant` ?
`returnable`	Can the column value be returned by an index-only scan?
`search_array`	Does the column natively support `col = ANY(array)` searches?
`search_nulls`	Does the column support `IS NULL` and `IS NOT NULL` searches?

Table 9.76. Index Properties

Name	Description
`clusterable`	Can the index be used in a `CLUSTER` command?
`index_scan`	Does the index support plain (non-bitmap) scans?
`bitmap_scan`	Does the index support bitmap scans?
`backward_scan`	Can the scan direction be changed in mid-scan (to support `FETCH BACKWARD` on a cursor without needing materialization)?

Table 9.77. Index Access Method Properties

Name	Description
`can_order`	Does the access method support `ASC`, `DESC` and related keywords in `CREATE INDEX`?
`can_unique`	Does the access method support unique indexes?
`can_multi_col`	Does the access method support indexes with multiple columns?
`can_exclude`	Does the access method support exclusion constraints?
`can_include`	Does the access method support the `INCLUDE` clause of `CREATE INDEX`?

Table 9.78. GUC Flags

Flag	Description
EXPLAIN	Parameters with this flag are included in EXPLAIN (SETTINGS) commands.
NO_SHOW_ALL	Parameters with this flag are excluded from SHOW ALL commands.
NO_RESET	Parameters with this flag do not support RESET commands.
NO_RESET_ALL	Parameters with this flag are excluded from RESET ALL commands.
NOT_IN_SAMPLE	Parameters with this flag are not included in postgresql.conf by default.
RUNTIME_COMPUTED	Parameters with this flag are runtime-computed ones.

9.27.5. Object Information and Addressing Functions

Table 9.79 lists functions related to database object identification and addressing.

Table 9.79. Object Information and Addressing Functions

Function Description
pg_describe_object (*classid* oid, *objid* oid, *objsubid* integer) → text Returns a textual description of a database object identified by catalog OID, object OID, and sub-object ID (such as a column number within a table; the sub-object ID is zero when referring to a whole object). This description is intended to be human-readable, and might be translated, depending on server configuration. This is especially useful to determine the identity of an object referenced in the pg_depend catalog. This function returns NULL values for undefined objects.
pg_identify_object (*classid* oid, *objid* oid, *objsubid* integer) → record (*type* text, *schema* text, *name* text, *identity* text) Returns a row containing enough information to uniquely identify the database object specified by catalog OID, object OID and sub-object ID. This information is intended to be machine-readable, and is never translated. *type* identifies the type of database object; *schema* is the schema name that the object belongs in, or NULL for object types that do not belong to schemas; *name* is the name of the object, quoted if necessary, if the name (along with schema name, if pertinent) is sufficient to uniquely identify the object, otherwise NULL; *identity* is the complete object identity, with the precise format depending on object type, and each name within the format being schema-qualified and quoted as necessary. Undefined objects are identified with NULL values.
pg_identify_object_as_address (*classid* oid, *objid* oid, *objsubid* integer) → record (*type* text, *object_names* text[], *object_args* text[]) Returns a row containing enough information to uniquely identify the database object specified by catalog OID, object OID and sub-object ID. The returned information is independent of the current server, that is, it could be used to identify an identically named object in another server. *type* identifies the type of database object; *object_names* and *object_args* are text arrays that together form a reference to the object. These three values can be passed to pg_get_object_address to obtain the internal address of the object.

Function Description
`pg_get_object_address` (*`type`* `text`, *`object_names`* `text[]`, *`object_args`* `text[]`) → `record` (*`classid`* `oid`, *`objid`* `oid`, *`objsubid`* `integer`) Returns a row containing enough information to uniquely identify the database object specified by a type code and object name and argument arrays. The returned values are the ones that would be used in system catalogs such as `pg_depend`; they can be passed to other system functions such as `pg_describe_object` or `pg_identify_object`. *`classid`* is the OID of the system catalog containing the object; *`objid`* is the OID of the object itself, and *`objsubid`* is the sub-object ID, or zero if none. This function is the inverse of `pg_identify_object_as_address`. Undefined objects are identified with `NULL` values.

9.27.6. Comment Information Functions

The functions shown in Table 9.80 extract comments previously stored with the COMMENT command. A null value is returned if no comment could be found for the specified parameters.

Table 9.80. Comment Information Functions

Function Description
`col_description` (*`table`* `oid`, *`column`* `integer`) → `text` Returns the comment for a table column, which is specified by the OID of its table and its column number. (`obj_description` cannot be used for table columns, since columns do not have OIDs of their own.)
`obj_description` (*`object`* `oid`, *`catalog`* `name`) → `text` Returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, `obj_description(123456, 'pg_class')` would retrieve the comment for the table with OID 123456.
`obj_description` (*`object`* `oid`) → `text` Returns the comment for a database object specified by its OID alone. This is *deprecated* since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment might be returned.
`shobj_description` (*`object`* `oid`, *`catalog`* `name`) → `text` Returns the comment for a shared database object specified by its OID and the name of the containing system catalog. This is just like `obj_description` except that it is used for retrieving comments on shared objects (that is, databases, roles, and tablespaces). Some system catalogs are global to all databases within each cluster, and the descriptions for objects in them are stored globally as well.

9.27.7. Data Validity Checking Functions

The functions shown in Table 9.81 can be helpful for checking validity of proposed input data.

Table 9.81. Data Validity Checking Functions

Function Description Example(s)
`pg_input_is_valid` (*`string`* `text`, *`type`* `text`) → `boolean` Tests whether the given *`string`* is valid input for the specified data type, returning true or false.

Function Description Example(s)

This function will only work as desired if the data type's input function has been updated to report invalid input as a “soft” error. Otherwise, invalid input will abort the transaction, just as if the string had been cast to the type directly.
`pg_input_is_valid('42', 'integer')` → `t`
`pg_input_is_valid('42000000000', 'integer')` → `f`
`pg_input_is_valid('1234.567', 'numeric(7,4)')` → `f`

`pg_input_error_info(` *`string`* `text,` *`type`* `text ) → record (` *`message`* `text,` *`detail`* `text,` *`hint`* `text,` *`sql_error_code`* `text )`
Tests whether the given *`string`* is valid input for the specified data type; if not, return the details of the error that would have been thrown. If the input is valid, the results are NULL. The inputs are the same as for `pg_input_is_valid`.
This function will only work as desired if the data type's input function has been updated to report invalid input as a “soft” error. Otherwise, invalid input will abort the transaction, just as if the string had been cast to the type directly.
`SELECT * FROM pg_input_error_info('42000000000', 'integer')`
→
`message | detail | hint | sql_error_code`
`---+---+---`
`value "42000000000" is out of range for type integer |  |  | 22003`

9.27.8. Transaction ID and Snapshot Information Functions

The functions shown in Table 9.82 provide server transaction information in an exportable form. The main use of these functions is to determine which transactions were committed between two snapshots.

Table 9.82. Transaction ID and Snapshot Information Functions

Function Description
`age ( xid ) → integer` Returns the number of transactions between the supplied transaction id and the current transaction counter.
`mxid_age ( xid ) → integer` Returns the number of multixacts IDs between the supplied multixact ID and the current multixacts counter.
`pg_current_xact_id () → xid8` Returns the current transaction's ID. It will assign a new one if the current transaction does not have one already (because it has not performed any database updates); see Section 66.1 for details. If executed in a subtransaction, this will return the top-level transaction ID; see Section 66.3 for details.
`pg_current_xact_id_if_assigned () → xid8` Returns the current transaction's ID, or `NULL` if no ID is assigned yet. (It's best to use this variant if the transaction might otherwise be read-only, to avoid unnecessary consump-

Function Description
tion of an XID.) If executed in a subtransaction, this will return the top-level transaction ID.
`pg_xact_status ( xid8 ) → text` Reports the commit status of a recent transaction. The result is one of `in progress`, `committed`, or `aborted`, provided that the transaction is recent enough that the system retains the commit status of that transaction. If it is old enough that no references to the transaction survive in the system and the commit status information has been discarded, the result is `NULL`. Applications might use this function, for example, to determine whether their transaction committed or aborted after the application and database server become disconnected while a `COMMIT` is in progress. Note that prepared transactions are reported as `in progress`; applications must check `pg_prepared_xacts` if they need to determine whether a transaction ID belongs to a prepared transaction.
`pg_current_snapshot () → pg_snapshot` Returns a current *snapshot*, a data structure showing which transaction IDs are now in-progress. Only top-level transaction IDs are included in the snapshot; subtransaction IDs are not shown; see Section 66.3 for details.
`pg_snapshot_xip ( pg_snapshot ) → setof xid8` Returns the set of in-progress transaction IDs contained in a snapshot.
`pg_snapshot_xmax ( pg_snapshot ) → xid8` Returns the `xmax` of a snapshot.
`pg_snapshot_xmin ( pg_snapshot ) → xid8` Returns the `xmin` of a snapshot.
`pg_visible_in_snapshot ( xid8, pg_snapshot ) → boolean` Is the given transaction ID *visible* according to this snapshot (that is, was it completed before the snapshot was taken)? Note that this function will not give the correct answer for a subtransaction ID (subxid); see Section 66.3 for details.

The internal transaction ID type `xid` is 32 bits wide and wraps around every 4 billion transactions. However, the functions shown in Table 9.82, except `age` and `mxid_age`, use a 64-bit type `xid8` that does not wrap around during the life of an installation and can be converted to `xid` by casting if required; see Section 66.1 for details. The data type `pg_snapshot` stores information about transaction ID visibility at a particular moment in time. Its components are described in Table 9.83. `pg_snapshot`'s textual representation is *`xmin`*`:`*`xmax`*`:`*`xip_list`*. For example `10:20:10,14,15` means `xmin=10, xmax=20, xip_list=10, 14, 15`.

Table 9.83. Snapshot Components

Name	Description
`xmin`	Lowest transaction ID that was still active. All transaction IDs less than `xmin` are either committed and visible, or rolled back and dead.
`xmax`	One past the highest completed transaction ID. All transaction IDs greater than or equal to `xmax` had not yet completed as of the time of the snapshot, and thus are invisible.
`xip_list`	Transactions in progress at the time of the snapshot. A transaction ID that is `xmin <= X < xmax` and not in this list was already completed at the time of the snapshot, and thus is either visible or dead according to its commit status. This

Name	Description
	list does not include the transaction IDs of sub-transactions (subxids).

In releases of PostgreSQL before 13 there was no `xid8` type, so variants of these functions were provided that used `bigint` to represent a 64-bit XID, with a correspondingly distinct snapshot data type `txid_snapshot`. These older functions have `txid` in their names. They are still supported for backward compatibility, but may be removed from a future release. See Table 9.84.

Table 9.84. Deprecated Transaction ID and Snapshot Information Functions

Function Description
`txid_current` () → `bigint` See `pg_current_xact_id()`.
`txid_current_if_assigned` () → `bigint` See `pg_current_xact_id_if_assigned()`.
`txid_current_snapshot` () → `txid_snapshot` See `pg_current_snapshot()`.
`txid_snapshot_xip` (`txid_snapshot`) → `setof bigint` See `pg_snapshot_xip()`.
`txid_snapshot_xmax` (`txid_snapshot`) → `bigint` See `pg_snapshot_xmax()`.
`txid_snapshot_xmin` (`txid_snapshot`) → `bigint` See `pg_snapshot_xmin()`.
`txid_visible_in_snapshot` (`bigint`, `txid_snapshot`) → `boolean` See `pg_visible_in_snapshot()`.
`txid_status` (`bigint`) → `text` See `pg_xact_status()`.

9.27.9. Committed Transaction Information Functions

The functions shown in Table 9.85 provide information about when past transactions were committed. They only provide useful data when the track_commit_timestamp configuration option is enabled, and only for transactions that were committed after it was enabled. Commit timestamp information is routinely removed during vacuum.

Table 9.85. Committed Transaction Information Functions

Function Description
`pg_xact_commit_timestamp` (`xid`) → `timestamp with time zone` Returns the commit timestamp of a transaction.
`pg_xact_commit_timestamp_origin` (`xid`) → `record` (*`timestamp`* `timestamp with time zone`, *`roident`* `oid`) Returns the commit timestamp and replication origin of a transaction.
`pg_last_committed_xact` () → `record` (*`xid`* `xid`, *`timestamp`* `timestamp with time zone`, *`roident`* `oid`) Returns the transaction ID, commit timestamp and replication origin of the latest committed transaction.

9.27.10. Control Data Functions

The functions shown in Table 9.86 print information initialized during `initdb`, such as the catalog version. They also show information about write-ahead logging and checkpoint processing. This information is cluster-wide, not specific to any one database. These functions provide most of the same information, from the same source, as the pg_controldata application.

Table 9.86. Control Data Functions

Function Description
`pg_control_checkpoint` () → `record` Returns information about current checkpoint state, as shown in Table 9.87.
`pg_control_system` () → `record` Returns information about current control file state, as shown in Table 9.88.
`pg_control_init` () → `record` Returns information about cluster initialization state, as shown in Table 9.89.
`pg_control_recovery` () → `record` Returns information about recovery state, as shown in Table 9.90.

Table 9.87. `pg_control_checkpoint` Output Columns

Column Name	Data Type
`checkpoint_lsn`	`pg_lsn`
`redo_lsn`	`pg_lsn`
`redo_wal_file`	`text`
`timeline_id`	`integer`
`prev_timeline_id`	`integer`
`full_page_writes`	`boolean`
`next_xid`	`text`
`next_oid`	`oid`
`next_multixact_id`	`xid`
`next_multi_offset`	`xid`
`oldest_xid`	`xid`
`oldest_xid_dbid`	`oid`
`oldest_active_xid`	`xid`
`oldest_multi_xid`	`xid`
`oldest_multi_dbid`	`oid`
`oldest_commit_ts_xid`	`xid`
`newest_commit_ts_xid`	`xid`
`checkpoint_time`	`timestamp with time zone`

Table 9.88. `pg_control_system` Output Columns

Column Name	Data Type
`pg_control_version`	`integer`

Column Name	Data Type
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone

Table 9.89. `pg_control_init` Output Columns

Column Name	Data Type
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float8_pass_by_value	boolean
data_page_checksum_version	integer

Table 9.90. `pg_control_recovery` Output Columns

Column Name	Data Type
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

9.27.11. Version Information Functions

The functions shown in Table 9.91 print version information.

Table 9.91. Version Information Functions

Function Description
`version` () → `text` Returns a string describing the PostgreSQL server's version. You can also get this information from server_version, or for a machine-readable version use server_version_num. Software developers should use `server_version_num` (available since 8.2) or `PQserverVersion` instead of parsing the text version.
`unicode_version` () → `text` Returns a string representing the version of Unicode used by PostgreSQL.
`icu_unicode_version` () → `text` Returns a string representing the version of Unicode used by ICU, if the server was built with ICU support; otherwise returns `NULL`

9.27.12. WAL Summarization Information Functions

The functions shown in Table 9.92 print information about the status of WAL summarization. See summarize_wal.

Table 9.92. WAL Summarization Information Functions

Function Description
`pg_available_wal_summaries` () → `setof record` (*`tli`* `bigint`, *`start_lsn`* `pg_lsn`, *`end_lsn`* `pg_lsn`) Returns information about the WAL summary files present in the data directory, under `pg_wal/summaries`. One row will be returned per WAL summary file. Each file summarizes WAL on the indicated TLI within the indicated LSN range. This function might be useful to determine whether enough WAL summaries are present on the server to take an incremental backup based on some prior backup whose start LSN is known.
`pg_wal_summary_contents` (*`tli`* `bigint`, *`start_lsn`* `pg_lsn`, *`end_lsn`* `pg_lsn`) → `setof record` (*`relfilenode`* `oid`, *`reltablespace`* `oid`, *`reldatabase`* `oid`, *`relforknumber`* `smallint`, *`relblocknumber`* `bigint`, *`is_limit_block`* `boolean`) Returns one information about the contents of a single WAL summary file identified by TLI and starting and ending LSNs. Each row with `is_limit_block` false indicates that the block identified by the remaining output columns was modified by at least one WAL record within the range of records summarized by this file. Each row with `is_limit_block` true indicates either that (a) the relation fork was truncated to the length given by `relblocknumber` within the relevant range of WAL records or (b) that the relation fork was created or dropped within the relevant range of WAL records; in such cases, `relblocknumber` will be zero.
`pg_get_wal_summarizer_state` () → `record` (*`summarized_tli`* `bigint`, *`summarized_lsn`* `pg_lsn`, *`pending_lsn`* `pg_lsn`, *`summarizer_pid`* `int`) Returns information about the progress of the WAL summarizer. If the WAL summarizer has never run since the instance was started, then `summarized_tli` and `summarized_lsn` will be `0` and `0/0` respectively; otherwise, they will be the TLI and ending LSN of the last WAL summary file written to disk. If the WAL summarizer is currently running, `pending_lsn` will be the ending LSN of the last record that it has consumed, which must always be greater than or equal to `summarized_lsn`; if the WAL summarizer is not running, it will be equal to `summarized_lsn`. `summarizer_pid` is the PID of the WAL summarizer process, if it is running, and otherwise NULL. As a special exception, the WAL summarizer will refuse to generate WAL summary files if run on WAL generated under `wal_level=minimal`, since such summaries would be unsafe to use as the basis for an incremental backup. In this case, the fields above will continue to advance as if summaries were being generated, but nothing will be written to disk. Once the summarizer reaches WAL generated while `wal_level` was set to `replica` or higher, it will resume writing summaries to disk.

9.28. System Administration Functions

The functions described in this section are used to control and monitor a PostgreSQL installation.

9.28.1. Configuration Settings Functions

Table 9.93 shows the functions available to query and alter run-time configuration parameters.

Table 9.93. Configuration Settings Functions

Function Description Example(s)

`current_setting` (*`setting_name`* `text` [, *`missing_ok`* `boolean`]) → `text`
Returns the current value of the setting *`setting_name`*. If there is no such setting, `current_setting` throws an error unless *`missing_ok`* is supplied and is `true` (in which case NULL is returned). This function corresponds to the SQL command SHOW.
`current_setting('datestyle')` → `ISO, MDY`

`set_config` (*`setting_name`* `text`, *`new_value`* `text`, *`is_local`* `boolean`) → `text`
Sets the parameter *`setting_name`* to *`new_value`*, and returns that value. If *`is_local`* is `true`, the new value will only apply during the current transaction. If you want the new value to apply for the rest of the current session, use `false` instead. This function corresponds to the SQL command SET.
`set_config('log_statement_stats', 'off', false)` → `off`

9.28.2. Server Signaling Functions

The functions shown in Table 9.94 send control signals to other server processes. Use of these functions is restricted to superusers by default but access may be granted to others using `GRANT`, with noted exceptions.

Each of these functions returns `true` if the signal was successfully sent and `false` if sending the signal failed.

Table 9.94. Server Signaling Functions

Function Description
`pg_cancel_backend` (*`pid`* `integer`) → `boolean` Cancels the current query of the session whose backend process has the specified process ID. This is also allowed if the calling role is a member of the role whose backend is being canceled or the calling role has privileges of `pg_signal_backend`, however only superusers can cancel superuser backends.
`pg_log_backend_memory_contexts` (*`pid`* `integer`) → `boolean` Requests to log the memory contexts of the backend with the specified process ID. This function can send the request to backends and auxiliary processes except logger. These memory contexts will be logged at `LOG` message level. They will appear in the server log based on the log configuration set (see Section 19.8 for more information), but will not be sent to the client regardless of client_min_messages.
`pg_reload_conf` () → `boolean` Causes all processes of the PostgreSQL server to reload their configuration files. (This is initiated by sending a SIGHUP signal to the postmaster process, which in turn sends SIGHUP to each of its children.) You can use the `pg_file_settings`, `pg_hba_file_rules` and `pg_ident_file_mappings` views to check the configuration files for possible errors, before reloading.
`pg_rotate_logfile` () → `boolean` Signals the log-file manager to switch to a new output file immediately. This works only when the built-in log collector is running, since otherwise there is no log-file manager subprocess.

Function Description
`pg_terminate_backend ( pid integer, timeout bigint DEFAULT 0 ) → boolean` Terminates the session whose backend process has the specified process ID. This is also allowed if the calling role is a member of the role whose backend is being terminated or the calling role has privileges of `pg_signal_backend`, however only superusers can terminate superuser backends. If *`timeout`* is not specified or zero, this function returns `true` whether the process actually terminates or not, indicating only that the sending of the signal was successful. If the *`timeout`* is specified (in milliseconds) and greater than zero, the function waits until the process is actually terminated or until the given time has passed. If the process is terminated, the function returns `true`. On timeout, a warning is emitted and `false` is returned.

`pg_cancel_backend` and `pg_terminate_backend` send signals (SIGINT or SIGTERM respectively) to backend processes identified by process ID. The process ID of an active backend can be found from the `pid` column of the `pg_stat_activity` view, or by listing the `postgres` processes on the server (using ps on Unix or the Task Manager on Windows). The role of an active backend can be found from the `usename` column of the `pg_stat_activity` view.

`pg_log_backend_memory_contexts` can be used to log the memory contexts of a backend process. For example:

```
postgres=# SELECT pg_log_backend_memory_contexts(pg_backend_pid());
 pg_log_backend_memory_contexts
--------------------------------
 t
(1 row)
```

One message for each memory context will be logged. For example:

```
LOG:  logging memory contexts of PID 10377
STATEMENT:  SELECT
 pg_log_backend_memory_contexts(pg_backend_pid());
LOG:  level: 0; TopMemoryContext: 80800 total in 6 blocks; 14432
 free (5 chunks); 66368 used
LOG:  level: 1; pgstat TabStatusArray lookup hash table: 8192 total
 in 1 blocks; 1408 free (0 chunks); 6784 used
LOG:  level: 1; TopTransactionContext: 8192 total in 1 blocks; 7720
 free (1 chunks); 472 used
LOG:  level: 1; RowDescriptionContext: 8192 total in 1 blocks; 6880
 free (0 chunks); 1312 used
LOG:  level: 1; MessageContext: 16384 total in 2 blocks; 5152 free
 (0 chunks); 11232 used
LOG:  level: 1; Operator class cache: 8192 total in 1 blocks; 512
 free (0 chunks); 7680 used
LOG:  level: 1; smgr relation table: 16384 total in 2 blocks; 4544
 free (3 chunks); 11840 used
LOG:  level: 1; TransactionAbortContext: 32768 total in 1 blocks;
 32504 free (0 chunks); 264 used
...
LOG:  level: 1; ErrorContext: 8192 total in 1 blocks; 7928 free (3
 chunks); 264 used
LOG:  Grand total: 1651920 bytes in 201 blocks; 622360 free (88
 chunks); 1029560 used
```

If there are more than 100 child contexts under the same parent, the first 100 child contexts are logged, along with a summary of the remaining contexts. Note that frequent calls to this function could incur significant overhead, because it may generate a large number of log messages.

9.28.3. Backup Control Functions

The functions shown in Table 9.95 assist in making on-line backups. These functions cannot be executed during recovery (except `pg_backup_start`, `pg_backup_stop`, and `pg_wal_lsn_diff`).

For details about proper usage of these functions, see Section 25.3.

Table 9.95. Backup Control Functions

Function Description
`pg_create_restore_point ( name text ) → pg_lsn` Creates a named marker record in the write-ahead log that can later be used as a recovery target, and returns the corresponding write-ahead log location. The given name can then be used with recovery_target_name to specify the point up to which recovery will proceed. Avoid creating multiple restore points with the same name, since recovery will stop at the first one whose name matches the recovery target. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_current_wal_flush_lsn () → pg_lsn` Returns the current write-ahead log flush location (see notes below).
`pg_current_wal_insert_lsn () → pg_lsn` Returns the current write-ahead log insert location (see notes below).
`pg_current_wal_lsn () → pg_lsn` Returns the current write-ahead log write location (see notes below).
`pg_backup_start ( label text [, fast boolean ] ) → pg_lsn` Prepares the server to begin an on-line backup. The only required parameter is an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) If the optional second parameter is given as `true`, it specifies executing `pg_backup_start` as quickly as possible. This forces an immediate checkpoint which will cause a spike in I/O operations, slowing any concurrently executing queries. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_backup_stop ( [wait_for_archive boolean ] ) → record ( lsn pg_lsn, labelfile text, spcmapfile text )` Finishes performing an on-line backup. The desired contents of the backup label file and the tablespace map file are returned as part of the result of the function and must be written to files in the backup area. These files must not be written to the live data directory (doing so will cause PostgreSQL to fail to restart in the event of a crash). There is an optional parameter of type `boolean`. If false, the function will return immediately after the backup is completed, without waiting for WAL to be archived. This behavior is only useful with backup software that independently monitors WAL archiving. Otherwise, WAL required to make the backup consistent might be missing and make the backup useless. By default or when this parameter is true, `pg_backup_stop` will wait for WAL to be archived when archiving is enabled. (On a standby, this means that it will wait only when `archive_mode` = `always`. If write activity on the primary is low, it may be useful to run `pg_switch_wal` on the primary in order to trigger an immediate segment switch.)

Function Description
When executed on a primary, this function also creates a backup history file in the write-ahead log archive area. The history file includes the label given to `pg_backup_start`, the starting and ending write-ahead log locations for the backup, and the starting and ending times of the backup. After recording the ending location, the current write-ahead log insertion point is automatically advanced to the next write-ahead log file, so that the ending write-ahead log file can be archived immediately to complete the backup. The result of the function is a single record. The *`lsn`* column holds the backup's ending write-ahead log location (which again can be ignored). The second column returns the contents of the backup label file, and the third column returns the contents of the tablespace map file. These must be stored as part of the backup and are required as part of the restore process. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_switch_wal` () → `pg_lsn` Forces the server to switch to a new write-ahead log file, which allows the current file to be archived (assuming you are using continuous archiving). The result is the ending write-ahead log location plus 1 within the just-completed write-ahead log file. If there has been no write-ahead log activity since the last write-ahead log switch, `pg_switch_wal` does nothing and returns the start location of the write-ahead log file currently in use. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_walfile_name` (*`lsn`* `pg_lsn`) → `text` Converts a write-ahead log location to the name of the WAL file holding that location.
`pg_walfile_name_offset` (*`lsn`* `pg_lsn`) → `record` (*`file_name`* `text`, *`file_offset`* `integer`) Converts a write-ahead log location to a WAL file name and byte offset within that file.
`pg_split_walfile_name` (*`file_name`* `text`) → `record` (*`segment_number`* `numeric`, *`timeline_id`* `bigint`) Extracts the sequence number and timeline ID from a WAL file name.
`pg_wal_lsn_diff` (*`lsn1`* `pg_lsn`, *`lsn2`* `pg_lsn`) → `numeric` Calculates the difference in bytes (*`lsn1`* - *`lsn2`*) between two write-ahead log locations. This can be used with `pg_stat_replication` or some of the functions shown in Table 9.95 to get the replication lag.

`pg_current_wal_lsn` displays the current write-ahead log write location in the same format used by the above functions. Similarly, `pg_current_wal_insert_lsn` displays the current write-ahead log insertion location and `pg_current_wal_flush_lsn` displays the current write-ahead log flush location. The insertion location is the "logical" end of the write-ahead log at any instant, while the write location is the end of what has actually been written out from the server's internal buffers, and the flush location is the last location known to be written to durable storage. The write location is the end of what can be examined from outside the server, and is usually what you want if you are interested in archiving partially-complete write-ahead log files. The insertion and flush locations are made available primarily for server debugging purposes. These are all read-only operations and do not require superuser permissions.

You can use `pg_walfile_name_offset` to extract the corresponding write-ahead log file name and byte offset from a `pg_lsn` value. For example:

```
postgres=# SELECT * FROM
 pg_walfile_name_offset((pg_backup_stop()).lsn);
       file_name         | file_offset
```

```
--------------------------+-------------
 000000010000000000000000D |     4039624
(1 row)
```

Similarly, `pg_walfile_name` extracts just the write-ahead log file name.

`pg_split_walfile_name` is useful to compute a LSN from a file offset and WAL file name, for example:

```
postgres=# \set file_name '000000010000000100C000AB'
postgres=# \set offset 256
postgres=# SELECT '0/0'::pg_lsn + pd.segment_number *
 ps.setting::int + :offset AS lsn
  FROM pg_split_walfile_name(:'file_name') pd,
       pg_show_all_settings() ps
  WHERE ps.name = 'wal_segment_size';
      lsn
---------------
 C001/AB000100
(1 row)
```

9.28.4. Recovery Control Functions

The functions shown in Table 9.96 provide information about the current status of a standby server. These functions may be executed both during recovery and in normal running.

Table 9.96. Recovery Information Functions

Function Description
`pg_is_in_recovery` () → `boolean` Returns true if recovery is still in progress.
`pg_last_wal_receive_lsn` () → `pg_lsn` Returns the last write-ahead log location that has been received and synced to disk by streaming replication. While streaming replication is in progress this will increase monotonically. If recovery has completed then this will remain static at the location of the last WAL record received and synced to disk during recovery. If streaming replication is disabled, or if it has not yet started, the function returns `NULL`.
`pg_last_wal_replay_lsn` () → `pg_lsn` Returns the last write-ahead log location that has been replayed during recovery. If recovery is still in progress this will increase monotonically. If recovery has completed then this will remain static at the location of the last WAL record applied during recovery. When the server has been started normally without recovery, the function returns `NULL`.
`pg_last_xact_replay_timestamp` () → `timestamp with time zone` Returns the time stamp of the last transaction replayed during recovery. This is the time at which the commit or abort WAL record for that transaction was generated on the primary. If no transactions have been replayed during recovery, the function returns `NULL`. Otherwise, if recovery is still in progress this will increase monotonically. If recovery has completed then this will remain static at the time of the last transaction applied during recovery. When the server has been started normally without recovery, the function returns `NULL`.
`pg_get_wal_resource_managers` () → `setof record` (*`rm_id`* `integer`, *`rm_name`* `text`, *`rm_builtin`* `boolean`)

Function Description
Returns the currently-loaded WAL resource managers in the system. The column *rm_builtin* indicates whether it's a built-in resource manager, or a custom resource manager loaded by an extension.

The functions shown in Table 9.97 control the progress of recovery. These functions may be executed only during recovery.

Table 9.97. Recovery Control Functions

Function Description
pg_is_wal_replay_paused () → boolean Returns true if recovery pause is requested.
pg_get_wal_replay_pause_state () → text Returns recovery pause state. The return values are not paused if pause is not requested, pause requested if pause is requested but recovery is not yet paused, and paused if the recovery is actually paused.
pg_promote (*wait* boolean DEFAULT true, *wait_seconds* integer DEFAULT 60) → boolean Promotes a standby server to primary status. With *wait* set to true (the default), the function waits until promotion is completed or *wait_seconds* seconds have passed, and returns true if promotion is successful and false otherwise. If *wait* is set to false, the function returns true immediately after sending a SIGUSR1 signal to the postmaster to trigger promotion. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
pg_wal_replay_pause () → void Request to pause recovery. A request doesn't mean that recovery stops right away. If you want a guarantee that recovery is actually paused, you need to check for the recovery pause state returned by pg_get_wal_replay_pause_state(). Note that pg_is_wal_replay_paused() returns whether a request is made. While recovery is paused, no further database changes are applied. If hot standby is active, all new queries will see the same consistent snapshot of the database, and no further query conflicts will be generated until recovery is resumed. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
pg_wal_replay_resume () → void Restarts recovery if it was paused. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

pg_wal_replay_pause and pg_wal_replay_resume cannot be executed while a promotion is ongoing. If a promotion is triggered while recovery is paused, the paused state ends and promotion continues.

If streaming replication is disabled, the paused state may continue indefinitely without a problem. If streaming replication is in progress then WAL records will continue to be received, which will eventually fill available disk space, depending upon the duration of the pause, the rate of WAL generation and available disk space.

9.28.5. Snapshot Synchronization Functions

PostgreSQL allows database sessions to synchronize their snapshots. A *snapshot* determines which data is visible to the transaction that is using the snapshot. Synchronized snapshots are necessary when two or more sessions need to see identical content in the database. If two sessions just start their transactions independently, there is always a possibility that some third transaction commits between the executions of the two `START TRANSACTION` commands, so that one session sees the effects of that transaction and the other does not.

To solve this problem, PostgreSQL allows a transaction to *export* the snapshot it is using. As long as the exporting transaction remains open, other transactions can *import* its snapshot, and thereby be guaranteed that they see exactly the same view of the database that the first transaction sees. But note that any database changes made by any one of these transactions remain invisible to the other transactions, as is usual for changes made by uncommitted transactions. So the transactions are synchronized with respect to pre-existing data, but act normally for changes they make themselves.

Snapshots are exported with the `pg_export_snapshot` function, shown in Table 9.98, and imported with the SET TRANSACTION command.

Table 9.98. Snapshot Synchronization Functions

Function Description
`pg_export_snapshot` () → `text` Saves the transaction's current snapshot and returns a `text` string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it. A transaction can export more than one snapshot, if needed. Note that doing so is only useful in `READ COMMITTED` transactions, since in `REPEATABLE READ` and higher isolation levels, transactions use the same snapshot throughout their lifetime. Once a transaction has exported any snapshots, it cannot be prepared with PREPARE TRANSACTION.
`pg_log_standby_snapshot` () → `pg_lsn` Take a snapshot of running transactions and write it to WAL, without having to wait for bgwriter or checkpointer to log one. This is useful for logical decoding on standby, as logical slot creation has to wait until such a record is replayed on the standby.

9.28.6. Replication Management Functions

The functions shown in Table 9.99 are for controlling and interacting with replication features. See Section 26.2.5, Section 26.2.6, and Chapter 48 for information about the underlying features. Use of functions for replication origin is only allowed to the superuser by default, but may be allowed to other users by using the `GRANT` command. Use of functions for replication slots is restricted to superusers and users having `REPLICATION` privilege.

Many of these functions have equivalent commands in the replication protocol; see Section 53.4.

The functions described in Section 9.28.3, Section 9.28.4, and Section 9.28.5 are also relevant for replication.

Table 9.99. Replication Management Functions

Function Description
`pg_create_physical_replication_slot ( slot_name name [, immediately_reserve boolean, temporary boolean ] ) → record ( slot_name name, lsn pg_lsn )` Creates a new physical replication slot named *slot_name*. The optional second parameter, when `true`, specifies that the LSN for this replication slot be reserved immediately; otherwise the LSN is reserved on first connection from a streaming replication client. Streaming changes from a physical slot is only possible with the streaming-replication protocol — see Section 53.4. The optional third parameter, *temporary*, when set to true, specifies that the slot should not be permanently stored to disk and is only meant for use by the current session. Temporary slots are also released upon any error. This function corresponds to the replication protocol command `CREATE_REPLICATION_SLOT ... PHYSICAL`.
`pg_drop_replication_slot ( slot_name name ) → void` Drops the physical or logical replication slot named *slot_name*. Same as replication protocol command `DROP_REPLICATION_SLOT`. For logical slots, this must be called while connected to the same database the slot was created on.
`pg_create_logical_replication_slot ( slot_name name, plugin name [, temporary boolean, twophase boolean, failover boolean ] ) → record ( slot_name name, lsn pg_lsn )` Creates a new logical (decoding) replication slot named *slot_name* using the output plugin *plugin*. The optional third parameter, *temporary*, when set to true, specifies that the slot should not be permanently stored to disk and is only meant for use by the current session. Temporary slots are also released upon any error. The optional fourth parameter, *twophase*, when set to true, specifies that the decoding of prepared transactions is enabled for this slot. The optional fifth parameter, *failover*, when set to true, specifies that this slot is enabled to be synced to the standbys so that logical replication can be resumed after failover. A call to this function has the same effect as the replication protocol command `CREATE_REPLICATION_SLOT ... LOGICAL`.
`pg_copy_physical_replication_slot ( src_slot_name name, dst_slot_name name [, temporary boolean ] ) → record ( slot_name name, lsn pg_lsn )` Copies an existing physical replication slot named *src_slot_name* to a physical replication slot named *dst_slot_name*. The copied physical slot starts to reserve WAL from the same LSN as the source slot. *temporary* is optional. If *temporary* is omitted, the same value as the source slot is used.
`pg_copy_logical_replication_slot ( src_slot_name name, dst_slot_name name [, temporary boolean [, plugin name ]] ) → record ( slot_name name, lsn pg_lsn )` Copies an existing logical replication slot named *src_slot_name* to a logical replication slot named *dst_slot_name*, optionally changing the output plugin and persistence. The copied logical slot starts from the same LSN as the source logical slot. Both *temporary* and *plugin* are optional; if they are omitted, the values of the source slot are used.
`pg_logical_slot_get_changes ( slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[] ) → setof record ( lsn pg_lsn, xid xid, data text )` Returns changes in the slot *slot_name*, starting from the point from which changes have been consumed last. If *upto_lsn* and *upto_nchanges* are NULL, logical decoding will continue until end of WAL. If *upto_lsn* is non-NULL, decoding will include only those transactions which commit prior to the specified LSN. If *upto_n*

Function Description
changes is non-NULL, decoding will stop when the number of rows produced by decoding exceeds the specified value. Note, however, that the actual number of rows returned may be larger, since this limit is only checked after adding the rows produced when decoding each new transaction commit. If the specified slot is a logical failover slot then the function will not return until all physical slots specified in synchronized_standby_slots have confirmed WAL receipt.
pg_logical_slot_peek_changes (*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn, *xid* xid, *data* text) Behaves just like the pg_logical_slot_get_changes() function, except that changes are not consumed; that is, they will be returned again on future calls.
pg_logical_slot_get_binary_changes (*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn, *xid* xid, *data* bytea) Behaves just like the pg_logical_slot_get_changes() function, except that changes are returned as bytea.
pg_logical_slot_peek_binary_changes (*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn, *xid* xid, *data* bytea) Behaves just like the pg_logical_slot_peek_changes() function, except that changes are returned as bytea.
pg_replication_slot_advance (*slot_name* name, *upto_lsn* pg_lsn) → record (*slot_name* name, *end_lsn* pg_lsn) Advances the current confirmed position of a replication slot named *slot_name*. The slot will not be moved backwards, and it will not be moved beyond the current insert location. Returns the name of the slot and the actual position that it was advanced to. The updated slot position information is written out at the next checkpoint if any advancing is done. So in the event of a crash, the slot may return to an earlier position. If the specified slot is a logical failover slot then the function will not return until all physical slots specified in synchronized_standby_slots have confirmed WAL receipt.
pg_replication_origin_create (*node_name* text) → oid Creates a replication origin with the given external name, and returns the internal ID assigned to it.
pg_replication_origin_drop (*node_name* text) → void Deletes a previously-created replication origin, including any associated replay progress.
pg_replication_origin_oid (*node_name* text) → oid Looks up a replication origin by name and returns the internal ID. If no such replication origin is found, NULL is returned.
pg_replication_origin_session_setup (*node_name* text) → void Marks the current session as replaying from the given origin, allowing replay progress to be tracked. Can only be used if no origin is currently selected. Use pg_replication_origin_session_reset to undo.
pg_replication_origin_session_reset () → void Cancels the effects of pg_replication_origin_session_setup().
pg_replication_origin_session_is_setup () → boolean Returns true if a replication origin has been selected in the current session.
pg_replication_origin_session_progress (*flush* boolean) → pg_lsn

Function Description
Returns the replay location for the replication origin selected in the current session. The parameter *flush* determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not.
pg_replication_origin_xact_setup (*origin_lsn* pg_lsn, *origin_timestamp* timestamp with time zone) → void Marks the current transaction as replaying a transaction that has committed at the given LSN and timestamp. Can only be called when a replication origin has been selected using pg_replication_origin_session_setup.
pg_replication_origin_xact_reset () → void Cancels the effects of pg_replication_origin_xact_setup().
pg_replication_origin_advance (*node_name* text, *lsn* pg_lsn) → void Sets replication progress for the given node to the given location. This is primarily useful for setting up the initial location, or setting a new location after configuration changes and similar. Be aware that careless use of this function can lead to inconsistently replicated data.
pg_replication_origin_progress (*node_name* text, *flush* boolean) → pg_lsn Returns the replay location for the given replication origin. The parameter *flush* determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not.
pg_logical_emit_message (*transactional* boolean, *prefix* text, *content* text [, *flush* boolean DEFAULT false]) → pg_lsn pg_logical_emit_message (*transactional* boolean, *prefix* text, *content* bytea [, *flush* boolean DEFAULT false]) → pg_lsn Emits a logical decoding message. This can be used to pass generic messages to logical decoding plugins through WAL. The *transactional* parameter specifies if the message should be part of the current transaction, or if it should be written immediately and decoded as soon as the logical decoder reads the record. The *prefix* parameter is a textual prefix that can be used by logical decoding plugins to easily recognize messages that are interesting for them. The *content* parameter is the content of the message, given either in text or binary form. The *flush* parameter (default set to false) controls if the message is immediately flushed to WAL or not. *flush* has no effect with *transactional*, as the message's WAL record is flushed along with its transaction.
pg_sync_replication_slots () → void Synchronize the logical failover replication slots from the primary server to the standby server. This function can only be executed on the standby server. Temporary synced slots, if any, cannot be used for logical decoding and must be dropped after promotion. See Section 47.2.3 for details. Note that this function cannot be executed if sync_replication_slots is enabled and the slotsync worker is already running to perform the synchronization of slots. **Caution** If, after executing the function, hot_standby_feedback is disabled on the standby or the physical slot configured in primary_slot_name is removed, then it is possible that the necessary rows of the synchronized slot will be removed by the VACUUM process on the primary server, resulting in the synchronized slot becoming invalidated.

9.28.7. Database Object Management Functions

The functions shown in Table 9.100 calculate the disk space usage of database objects, or assist in presentation or understanding of usage results. `bigint` results are measured in bytes. If an OID that does not represent an existing object is passed to one of these functions, `NULL` is returned.

Table 9.100. Database Object Size Functions

Function Description
`pg_column_size` (`"any"`) → `integer` Shows the number of bytes used to store any individual data value. If applied directly to a table column value, this reflects any compression that was done.
`pg_column_compression` (`"any"`) → `text` Shows the compression algorithm that was used to compress an individual variable-length value. Returns `NULL` if the value is not compressed.
`pg_column_toast_chunk_id` (`"any"`) → `oid` Shows the `chunk_id` of an on-disk TOASTed value. Returns `NULL` if the value is un-TOASTed or not on-disk. See Section 65.2 for more information about TOAST.

`pg_database_size` (`name`) → `bigint`
`pg_database_size` (`oid`) → `bigint`
Computes the total disk space used by the database with the specified name or OID. To use this function, you must have `CONNECT` privilege on the specified database (which is granted by default) or have privileges of the `pg_read_all_stats` role.

`pg_indexes_size` (`regclass`) → `bigint` Computes the total disk space used by indexes attached to the specified table.
`pg_relation_size` (*`relation`* `regclass` [, *`fork`* `text`]) → `bigint` Computes the disk space used by one “fork” of the specified relation. (Note that for most purposes it is more convenient to use the higher-level functions `pg_total_relation_size` or `pg_table_size`, which sum the sizes of all forks.) With one argument, this returns the size of the main data fork of the relation. The second argument can be provided to specify which fork to examine: • `main` returns the size of the main data fork of the relation. • `fsm` returns the size of the Free Space Map (see Section 65.3) associated with the relation. • `vm` returns the size of the Visibility Map (see Section 65.4) associated with the relation. • `init` returns the size of the initialization fork, if any, associated with the relation.
`pg_size_bytes` (`text`) → `bigint` Converts a size in human-readable format (as returned by `pg_size_pretty`) into bytes. Valid units are `bytes`, `B`, `kB`, `MB`, `GB`, `TB`, and `PB`.

`pg_size_pretty` (`bigint`) → `text`
`pg_size_pretty` (`numeric`) → `text`
Converts a size in bytes into a more easily human-readable format with size units (bytes, kB, MB, GB, TB, or PB as appropriate). Note that the units are powers of 2 rather than powers of 10, so 1kB is 1024 bytes, 1MB is 1024^2 = 1048576 bytes, and so on.

`pg_table_size` (`regclass`) → `bigint` Computes the disk space used by the specified table, excluding indexes (but including its TOAST table if any, free space map, and visibility map).

Function Description

`pg_tablespace_size ( name ) → bigint`
`pg_tablespace_size ( oid ) → bigint`
Computes the total disk space used in the tablespace with the specified name or OID. To use this function, you must have `CREATE` privilege on the specified tablespace or have privileges of the `pg_read_all_stats` role, unless it is the default tablespace for the current database.

`pg_total_relation_size ( regclass ) → bigint` Computes the total disk space used by the specified table, including all indexes and TOAST data. The result is equivalent to `pg_table_size` + `pg_indexes_size`.

The functions above that operate on tables or indexes accept a `regclass` argument, which is simply the OID of the table or index in the `pg_class` system catalog. You do not have to look up the OID by hand, however, since the `regclass` data type's input converter will do the work for you. See Section 8.19 for details.

The functions shown in Table 9.101 assist in identifying the specific disk files associated with database objects.

Table 9.101. Database Object Location Functions

Function Description
`pg_relation_filenode ( relation regclass ) → oid` Returns the “filenode” number currently assigned to the specified relation. The filenode is the base component of the file name(s) used for the relation (see Section 65.1 for more information). For most relations the result is the same as `pg_class.relfilenode`, but for certain system catalogs `relfilenode` is zero and this function must be used to get the correct value. The function returns NULL if passed a relation that does not have storage, such as a view.
`pg_relation_filepath ( relation regclass ) → text` Returns the entire file path name (relative to the database cluster's data directory, `PGDATA`) of the relation.
`pg_filenode_relation ( tablespace oid, filenode oid ) → regclass` Returns a relation's OID given the tablespace OID and filenode it is stored under. This is essentially the inverse mapping of `pg_relation_filepath`. For a relation in the database's default tablespace, the tablespace can be specified as zero. Returns `NULL` if no relation in the current database is associated with the given values.

Table 9.102 lists functions used to manage collations.

Table 9.102. Collation Management Functions

Function Description
`pg_collation_actual_version ( oid ) → text` Returns the actual version of the collation object as it is currently installed in the operating system. If this is different from the value in `pg_collation.collversion`, then objects depending on the collation might need to be rebuilt. See also ALTER COLLATION.
`pg_database_collation_actual_version ( oid ) → text` Returns the actual version of the database's collation as it is currently installed in the operating system. If this is different from the value in `pg_database.datcollver-`

Function Description
sion, then objects depending on the collation might need to be rebuilt. See also ALTER DATABASE.
pg_import_system_collations (*schema* regnamespace) → integer Adds collations to the system catalog pg_collation based on all the locales it finds in the operating system. This is what initdb uses; see Section 23.2.2 for more details. If additional locales are installed into the operating system later on, this function can be run again to add collations for the new locales. Locales that match existing entries in pg_collation will be skipped. (But collation objects based on locales that are no longer present in the operating system are not removed by this function.) The *schema* parameter would typically be pg_catalog, but that is not a requirement; the collations could be installed into some other schema as well. The function returns the number of new collation objects it created. Use of this function is restricted to superusers.

Table 9.103 lists functions that provide information about the structure of partitioned tables.

Table 9.103. Partitioning Information Functions

Function Description
pg_partition_tree (regclass) → setof record (*relid* regclass, *parentrelid* regclass, *isleaf* boolean, *level* integer) Lists the tables or indexes in the partition tree of the given partitioned table or partitioned index, with one row for each partition. Information provided includes the OID of the partition, the OID of its immediate parent, a boolean value telling if the partition is a leaf, and an integer telling its level in the hierarchy. The level value is 0 for the input table or index, 1 for its immediate child partitions, 2 for their partitions, and so on. Returns no rows if the relation does not exist or is not a partition or partitioned table.
pg_partition_ancestors (regclass) → setof regclass Lists the ancestor relations of the given partition, including the relation itself. Returns no rows if the relation does not exist or is not a partition or partitioned table.
pg_partition_root (regclass) → regclass Returns the top-most parent of the partition tree to which the given relation belongs. Returns NULL if the relation does not exist or is not a partition or partitioned table.

For example, to check the total size of the data contained in a partitioned table measurement, one could use the following query:

```
SELECT pg_size_pretty(sum(pg_relation_size(relid))) AS total_size
  FROM pg_partition_tree('measurement');
```

9.28.8. Index Maintenance Functions

Table 9.104 shows the functions available for index maintenance tasks. (Note that these maintenance tasks are normally done automatically by autovacuum; use of these functions is only required in special cases.) These functions cannot be executed during recovery. Use of these functions is restricted to superusers and the owner of the given index.

Table 9.104. Index Maintenance Functions

Function Description
brin_summarize_new_values (*index* regclass) → integer

Function Description
Scans the specified BRIN index to find page ranges in the base table that are not currently summarized by the index; for any such range it creates a new summary index tuple by scanning those table pages. Returns the number of new page range summaries that were inserted into the index.
`brin_summarize_range` (*`index`* `regclass`, *`blockNumber`* `bigint`) → `integer` Summarizes the page range covering the given block, if not already summarized. This is like `brin_summarize_new_values` except that it only processes the page range that covers the given table block number.
`brin_desummarize_range` (*`index`* `regclass`, *`blockNumber`* `bigint`) → `void` Removes the BRIN index tuple that summarizes the page range covering the given table block, if there is one.
`gin_clean_pending_list` (*`index`* `regclass`) → `bigint` Cleans up the “pending” list of the specified GIN index by moving entries in it, in bulk, to the main GIN data structure. Returns the number of pages removed from the pending list. If the argument is a GIN index built with the `fastupdate` option disabled, no cleanup happens and the result is zero, because the index doesn't have a pending list. See Section 64.4.4.1 and Section 64.4.5 for details about the pending list and `fastupdate` option.

9.28.9. Generic File Access Functions

The functions shown in Table 9.105 provide native access to files on the machine hosting the server. Only files within the database cluster directory and the `log_directory` can be accessed, unless the user is a superuser or is granted the role `pg_read_server_files`. Use a relative path for files in the cluster directory, and a path matching the `log_directory` configuration setting for log files.

Note that granting users the EXECUTE privilege on `pg_read_file()`, or related functions, allows them the ability to read any file on the server that the database server process can read; these functions bypass all in-database privilege checks. This means that, for example, a user with such access is able to read the contents of the `pg_authid` table where authentication information is stored, as well as read any table data in the database. Therefore, granting access to these functions should be carefully considered.

When granting privilege on these functions, note that the table entries showing optional parameters are mostly implemented as several physical functions with different parameter lists. Privilege must be granted separately on each such function, if it is to be used. psql's `\df` command can be useful to check what the actual function signatures are.

Some of these functions take an optional *`missing_ok`* parameter, which specifies the behavior when the file or directory does not exist. If `true`, the function returns `NULL` or an empty result set, as appropriate. If `false`, an error is raised. (Failure conditions other than “file not found” are reported as errors in any case.) The default is `false`.

Table 9.105. Generic File Access Functions

Function Description
`pg_ls_dir` (*`dirname`* `text` [, *`missing_ok`* `boolean`, *`include_dot_dirs`* `boolean`]) → `setof text` Returns the names of all files (and directories and other special files) in the specified directory. The *`include_dot_dirs`* parameter indicates whether “.” and “..” are to be included in the result set; the default is to exclude them. Including them can be useful when *`missing_ok`* is `true`, to distinguish an empty directory from a non-existent directory.

Function Description
This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
pg_ls_logdir () → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the server's log directory. Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and roles with privileges of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_ls_waldir () → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the server's write-ahead log (WAL) directory. Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and roles with privileges of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_ls_logicalmapdir () → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the server's pg_logical/mappings directory. Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and members of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_ls_logicalsnapdir () → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the server's pg_logical/snapshots directory. Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and members of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_ls_replslotdir (*slot_name* text) → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the server's pg_replslot/slot_name directory, where *slot_name* is the name of the replication slot provided as input of the function. Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and members of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_ls_archive_statusdir () → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the server's WAL archive status directory (pg_wal/archive_status). Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and members of the pg_monitor role by default, but other users can be granted EXECUTE to run the function.
pg_ls_tmpdir ([*tablespace* oid]) → setof record (*name* text, *size* bigint, *modification* timestamp with time zone) Returns the name, size, and last modification time (mtime) of each ordinary file in the temporary file directory for the specified *tablespace*. If *tablespace* is not provid-

Function Description
ed, the `pg_default` tablespace is examined. Filenames beginning with a dot, directories, and other special files are excluded. This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function.
`pg_read_file ( filename text [, offset bigint, length bigint ] [, missing_ok boolean ] ) → text` Returns all or part of a text file, starting at the given byte *`offset`*, returning at most *`length`* bytes (less if the end of file is reached first). If *`offset`* is negative, it is relative to the end of the file. If *`offset`* and *`length`* are omitted, the entire file is returned. The bytes read from the file are interpreted as a string in the database's encoding; an error is thrown if they are not valid in that encoding. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

`pg_read_binary_file ( filename text [, offset bigint, length bigint ] [, missing_ok boolean ] ) → bytea`
Returns all or part of a file. This function is identical to `pg_read_file` except that it can read arbitrary binary data, returning the result as `bytea` not `text`; accordingly, no encoding checks are performed.
This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
In combination with the `convert_from` function, this function can be used to read a text file in a specified encoding and convert to the database's encoding:
`SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');`

`pg_stat_file ( filename text [, missing_ok boolean ] ) → record ( size bigint, access timestamp with time zone, modification timestamp with time zone, change timestamp with time zone, creation timestamp with time zone, isdir boolean )` Returns a record containing the file's size, last access time stamp, last modification time stamp, last file status change time stamp (Unix platforms only), file creation time stamp (Windows only), and a flag indicating if it is a directory. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

9.28.10. Advisory Lock Functions

The functions shown in Table 9.106 manage advisory locks. For details about proper use of these functions, see Section 13.3.5.

All these functions are intended to be used to lock application-defined resources, which can be identified either by a single 64-bit key value or two 32-bit key values (note that these two key spaces do not overlap). If another session already holds a conflicting lock on the same resource identifier, the functions will either wait until the resource becomes available, or return a `false` result, as appropriate for the function. Locks can be either shared or exclusive: a shared lock does not conflict with other shared locks on the same resource, only with exclusive locks. Locks can be taken at session level (so that they are held until released or the session ends) or at transaction level (so that they are held until the current transaction ends; there is no provision for manual release). Multiple session-level lock requests stack, so that if the same resource identifier is locked three times there must then be three unlock requests to release the resource in advance of session end.

Table 9.106. Advisory Lock Functions

Function Description

`pg_advisory_lock` (*key* `bigint`) → `void`
`pg_advisory_lock` (*key1* `integer`, *key2* `integer`) → `void`
Obtains an exclusive session-level advisory lock, waiting if necessary.

`pg_advisory_lock_shared` (*key* `bigint`) → `void`
`pg_advisory_lock_shared` (*key1* `integer`, *key2* `integer`) → `void`
Obtains a shared session-level advisory lock, waiting if necessary.

`pg_advisory_unlock` (*key* `bigint`) → `boolean`
`pg_advisory_unlock` (*key1* `integer`, *key2* `integer`) → `boolean`
Releases a previously-acquired exclusive session-level advisory lock. Returns `true` if the lock is successfully released. If the lock was not held, `false` is returned, and in addition, an SQL warning will be reported by the server.

`pg_advisory_unlock_all` () → `void` Releases all session-level advisory locks held by the current session. (This function is implicitly invoked at session end, even if the client disconnects ungracefully.)

`pg_advisory_unlock_shared` (*key* `bigint`) → `boolean`
`pg_advisory_unlock_shared` (*key1* `integer`, *key2* `integer`) → `boolean`
Releases a previously-acquired shared session-level advisory lock. Returns `true` if the lock is successfully released. If the lock was not held, `false` is returned, and in addition, an SQL warning will be reported by the server.

`pg_advisory_xact_lock` (*key* `bigint`) → `void`
`pg_advisory_xact_lock` (*key1* `integer`, *key2* `integer`) → `void`
Obtains an exclusive transaction-level advisory lock, waiting if necessary.

`pg_advisory_xact_lock_shared` (*key* `bigint`) → `void`
`pg_advisory_xact_lock_shared` (*key1* `integer`, *key2* `integer`) → `void`
Obtains a shared transaction-level advisory lock, waiting if necessary.

`pg_try_advisory_lock` (*key* `bigint`) → `boolean`
`pg_try_advisory_lock` (*key1* `integer`, *key2* `integer`) → `boolean`
Obtains an exclusive session-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately.

`pg_try_advisory_lock_shared` (*key* `bigint`) → `boolean`
`pg_try_advisory_lock_shared` (*key1* `integer`, *key2* `integer`) → `boolean`
Obtains a shared session-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately.

`pg_try_advisory_xact_lock` (*key* `bigint`) → `boolean`
`pg_try_advisory_xact_lock` (*key1* `integer`, *key2* `integer`) → `boolean`
Obtains an exclusive transaction-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately.

`pg_try_advisory_xact_lock_shared` (*key* `bigint`) → `boolean` `pg_try_advisory_xact_lock_shared` (*key1* `integer`, *key2* `integer`) → `boolean`

Function Description
Obtains a shared transaction-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately.

9.29. Trigger Functions

While many uses of triggers involve user-written trigger functions, PostgreSQL provides a few built-in trigger functions that can be used directly in user-defined triggers. These are summarized in Table 9.107. (Additional built-in trigger functions exist, which implement foreign key constraints and deferred index constraints. Those are not documented here since users need not use them directly.)

For more information about creating triggers, see CREATE TRIGGER.

Table 9.107. Built-In Trigger Functions

Function Description Example Usage

`suppress_redundant_updates_trigger ( ) → trigger`
Suppresses do-nothing update operations. See below for details.
`CREATE TRIGGER ... suppress_redundant_updates_trigger()`

`tsvector_update_trigger ( ) → trigger`
Automatically updates a `tsvector` column from associated plain-text document column(s). The text search configuration to use is specified by name as a trigger argument. See Section 12.4.3 for details.
`CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)`

`tsvector_update_trigger_column ( ) → trigger`
Automatically updates a `tsvector` column from associated plain-text document column(s). The text search configuration to use is taken from a `regconfig` column of the table. See Section 12.4.3 for details.
`CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, tsconfigcol, title, body)`

The `suppress_redundant_updates_trigger` function, when applied as a row-level `BEFORE UPDATE` trigger, will prevent any update that does not actually change the data in the row from taking place. This overrides the normal behavior which always performs a physical row update regardless of whether or not the data has changed. (This normal behavior makes updates run faster, since no checking is required, and is also useful in certain cases.)

Ideally, you should avoid running updates that don't actually change the data in the record. Redundant updates can cost considerable unnecessary time, especially if there are lots of indexes to alter, and space in dead rows that will eventually have to be vacuumed. However, detecting such situations in client code is not always easy, or even possible, and writing expressions to detect them can be error-prone. An alternative is to use `suppress_redundant_updates_trigger`, which will skip updates that don't change the data. You should use this with care, however. The trigger takes a small but non-trivial time for each record, so if most of the records affected by updates do actually change, use of this trigger will make updates run slower on average.

The `suppress_redundant_updates_trigger` function can be added to a table like this:

```
CREATE TRIGGER z_min_update
```

```
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

In most cases, you need to fire this trigger last for each row, so that it does not override other triggers that might wish to alter the row. Bearing in mind that triggers fire in name order, you would therefore choose a trigger name that comes after the name of any other trigger you might have on the table. (Hence the “z” prefix in the example.)

9.30. Event Trigger Functions

PostgreSQL provides these helper functions to retrieve information from event triggers.

For more information about event triggers, see Chapter 38.

9.30.1. Capturing Changes at Command End

`pg_event_trigger_ddl_commands () → setof record`

`pg_event_trigger_ddl_commands` returns a list of DDL commands executed by each user action, when invoked in a function attached to a `ddl_command_end` event trigger. If called in any other context, an error is raised. `pg_event_trigger_ddl_commands` returns one row for each base command executed; some commands that are a single SQL sentence may return more than one row. This function returns the following columns:

Name	Type	Description
`classid`	`oid`	OID of catalog the object belongs in
`objid`	`oid`	OID of the object itself
`objsubid`	`integer`	Sub-object ID (e.g., attribute number for a column)
`command_tag`	`text`	Command tag
`object_type`	`text`	Type of the object
`schema_name`	`text`	Name of the schema the object belongs in, if any; otherwise `NULL`. No quoting is applied.
`object_identity`	`text`	Text rendering of the object identity, schema-qualified. Each identifier included in the identity is quoted if necessary.
`in_extension`	`boolean`	True if the command is part of an extension script
`command`	`pg_ddl_command`	A complete representation of the command, in internal format. This cannot be output directly, but it can be passed to other functions to obtain different pieces of information about the command.

9.30.2. Processing Objects Dropped by a DDL Command

`pg_event_trigger_dropped_objects () → setof record`

`pg_event_trigger_dropped_objects` returns a list of all objects dropped by the command in whose `sql_drop` event it is called. If called in any other context, an error is raised. This function returns the following columns:

Name	Type	Description
`classid`	`oid`	OID of catalog the object belonged in
`objid`	`oid`	OID of the object itself
`objsubid`	`integer`	Sub-object ID (e.g., attribute number for a column)
`original`	`boolean`	True if this was one of the root object(s) of the deletion
`normal`	`boolean`	True if there was a normal dependency relationship in the dependency graph leading to this object
`is_temporary`	`boolean`	True if this was a temporary object
`object_type`	`text`	Type of the object
`schema_name`	`text`	Name of the schema the object belonged in, if any; otherwise `NULL`. No quoting is applied.
`object_name`	`text`	Name of the object, if the combination of schema and name can be used as a unique identifier for the object; otherwise `NULL`. No quoting is applied, and name is never schema-qualified.
`object_identity`	`text`	Text rendering of the object identity, schema-qualified. Each identifier included in the identity is quoted if necessary.
`address_names`	`text[]`	An array that, together with `object_type` and `address_args`, can be used by the `pg_get_object_address` function to recreate the object address in a remote server containing an identically named object of the same kind.
`address_args`	`text[]`	Complement for `address_names`

The `pg_event_trigger_dropped_objects` function can be used in an event trigger like this:

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
```

```
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END;
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE FUNCTION test_event_trigger_for_drops();
```

9.30.3. Handling a Table Rewrite Event

The functions shown in Table 9.108 provide information about a table for which a `table_rewrite` event has just been called. If called in any other context, an error is raised.

Table 9.108. Table Rewrite Information Functions

Function Description
`pg_event_trigger_table_rewrite_oid` () → `oid` Returns the OID of the table about to be rewritten.
`pg_event_trigger_table_rewrite_reason` () → `integer` Returns a code explaining the reason(s) for rewriting. The value is a bitmap built from the following values: `1` (the table has changed its persistence), `2` (default value of a column has changed), `4` (a column has a new data type) and `8` (the table access method has changed).

These functions can be used in an event trigger like this:

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

9.31. Statistics Information Functions

PostgreSQL provides a function to inspect complex statistics defined using the `CREATE STATISTICS` command.

9.31.1. Inspecting MCV Lists

`pg_mcv_list_items ( pg_mcv_list ) → setof record`

`pg_mcv_list_items` returns a set of records describing all items stored in a multi-column MCV list. It returns the following columns:

Name	Type	Description
`index`	`integer`	index of the item in the MCV list
`values`	`text[]`	values stored in the MCV item
`nulls`	`boolean[]`	flags identifying `NULL` values
`frequency`	`double precision`	frequency of this MCV item
`base_frequency`	`double precision`	base frequency of this MCV item

The `pg_mcv_list_items` function can be used like this:

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid
 = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname =
 'stts';
```

Values of the `pg_mcv_list` type can be obtained only from the `pg_statistic_ext_data.stxdmcv` column.

Chapter 10. Type Conversion

SQL statements can, intentionally or not, require the mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user does not need to understand the details of the type conversion mechanism. However, implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by using *explicit* type conversion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 8 and Chapter 9 for more information on specific data types and allowed functions and operators.

10.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is more general and flexible than other SQL implementations. Hence, most type conversion behavior in PostgreSQL is governed by general rules rather than by ad hoc heuristics. This allows the use of mixed-type expressions even with user-defined types.

The PostgreSQL scanner/parser divides lexical elements into five fundamental categories: integers, non-integer numbers, strings, identifiers, and key words. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

has two literal constants, of type `text` and `point`. If a type is not specified for a string literal, then the placeholder type `unknown` is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Function calls

> Much of the PostgreSQL type system is built around a rich set of functions. Functions can have one or more arguments. Since PostgreSQL permits function overloading, the function name alone does not uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Operators

> PostgreSQL allows expressions with prefix (one-argument) operators, as well as infix (two-argument) operators. Like functions, operators can be overloaded, so the same problem of selecting the right operator exists.

Value Storage

> SQL `INSERT` and `UPDATE` statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

`UNION`, `CASE`, and related constructs

Since all query results from a unionized `SELECT` statement must appear in a single set of columns, the types of the results of each `SELECT` clause must be matched up and converted to a uniform set. Similarly, the result expressions of a `CASE` construct must be converted to a common type so that the `CASE` expression as a whole has a known output type. Some other constructs, such as `ARRAY[]` and the `GREATEST` and `LEAST` functions, likewise require determination of a common type for several subexpressions.

The system catalogs store information about which conversions, or *casts*, exist between which data types, and how to perform those conversions. Additional casts can be added by the user with the CREATE CAST command. (This is usually done in conjunction with defining new data types. The set of casts between built-in types has been carefully crafted and is best not altered.)

An additional heuristic provided by the parser allows improved determination of the proper casting behavior among groups of types that have implicit casts. Data types are divided into several basic *type categories*, including `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network`, and user-defined. (For a list see Table 51.65; but note it is also possible to create custom type categories.) Within each category there can be one or more *preferred types*, which are preferred when there is a choice of possible types. With careful selection of preferred types and available implicit casts, it is possible to ensure that ambiguous expressions (those with multiple candidate parsing solutions) can be resolved in a useful way.

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- There should be no extra overhead in the parser or executor if a query does not need implicit type conversion. That is, if a query is well-formed and the types already match, then the query should execute without spending extra time in the parser and without introducing unnecessary implicit conversion calls in the query.
- Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function and no longer do implicit conversion to use the old function.

10.2. Operators

The specific operator that is referenced by an operator expression is determined using the following procedure. Note that this procedure is indirectly affected by the precedence of the operators involved, since that will determine which sub-expressions are taken to be the inputs of which operators. See Section 4.1.6 for more information.

Operator Type Resolution

1. Select the operators to be considered from the `pg_operator` system catalog. If a non-schema-qualified operator name was used (the usual case), the operators considered are those with the matching name and argument count that are visible in the current search path (see Section 5.10.3). If a qualified operator name was given, only operators in the specified schema are considered.
    - (Optional) If the search path finds multiple operators with identical argument types, only the one appearing earliest in the path is considered. Operators with different argument types are considered on an equal footing regardless of search path position.
2. Check for an operator accepting exactly the input argument types. If one exists (there can be only one exact match in the set of operators considered), use it. Lack of an exact match creates a

security hazard when calling, via qualified name [1] (not typical), any operator found in a schema that permits untrusted users to create objects. In such situations, cast arguments to force an exact match.

a. (Optional) If one argument of a binary operator invocation is of the `unknown` type, then assume it is the same type as the other argument for this check. Invocations involving two `unknown` inputs, or a prefix operator with an `unknown` input, will never find a match at this step.

b. (Optional) If one argument of a binary operator invocation is of the `unknown` type and the other is of a domain type, next check to see if there is an operator accepting exactly the domain's base type on both sides; if so, use it.

3. Look for the best match.

a. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. `unknown` literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

b. If any input argument is of a domain type, treat it as being of the domain's base type for all subsequent steps. This ensures that domains act like their base types for purposes of ambiguous-operator resolution.

c. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have exact matches. If only one candidate remains, use it; else continue to the next step.

d. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

e. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.

f. If there are both `unknown` and known-type arguments, and all the known-type arguments have the same type, assume that the `unknown` arguments are also of that type, and check which candidates can accept that type at the `unknown`-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Some examples follow.

Example 10.1. Square Root Operator Type Resolution

There is only one square root operator (prefix `|/`) defined in the standard catalog, and it takes an argument of type `double precision`. The scanner assigns an initial type of `integer` to the argument in this query expression:

```
SELECT |/ 40 AS "square root of 40";
```

[1] The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a secure schema usage pattern.

```
 square root of 40
------------------
 6.324555320336759
(1 row)
```

So the parser does a type conversion on the operand and the query is equivalent to:

```
SELECT |/ CAST(40 AS double precision) AS "square root of 40";
```

Example 10.2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types and for working with complex extension types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking `text` for both arguments. Since there is, it assumes that the second argument should be interpreted as type `text`.

Here is a concatenation of two values of unspecified types:

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the preferred type for strings, `text`, is used as the specific type to resolve the unknown-type literals as.

Example 10.3. Absolute-Value and Negation Operator Type Resolution

The PostgreSQL operator catalog has several entries for the prefix operator `@`, all of which implement absolute-value operations for various numeric data types. One of these entries is for type `float8`, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with an `unknown` input:

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Here the system has implicitly resolved the unknown-type literal as type `float8` before applying the chosen operator. We can verify that `float8` and not some other type was used:

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

On the other hand, the prefix operator ~ (bitwise negation) is defined only for integer data types, not for `float8`. So, if we try a similar case with ~, we get:

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need
 to add
explicit type casts.
```

This happens because the system cannot decide which of the several possible ~ operators should be preferred. We can help it out with an explicit cast:

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

Example 10.4. Array Inclusion Operator Type Resolution

Here is another example of resolving an operator with one known and one unknown input:

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

The PostgreSQL operator catalog has several entries for the infix operator <@, but the only two that could possibly accept an integer array on the left-hand side are array inclusion (`anyarray <@ anyarray`) and range inclusion (`anyelement <@ anyrange`). Since none of these polymorphic pseudo-types (see Section 8.21) are considered preferred, the parser cannot resolve the ambiguity on that basis. However, Step 3.f tells it to assume that the unknown-type literal is of the same type as the other input, that is, integer array. Now only one of the two operators can match, so array inclusion is selected. (Had range inclusion been selected, we would have gotten an error, because the string does not have the right format to be a range literal.)

Example 10.5. Custom Operator on a Domain Type

Users sometimes try to declare operators applying just to a domain type. This is possible but is not nearly as useful as it might seem, because the operator resolution rules are designed to select operators applying to the domain's base type. As an example consider

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean
 AS ...;
```

```
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext,
 rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

This query will not use the custom operator. The parser will first see if there is a `mytext = mytext` operator (Step 2.a), which there is not; then it will consider the domain's base type `text`, and see if there is a `text = text` operator (Step 2.b), which there is; so it resolves the `unknown`-type literal as `text` and uses the `text = text` operator. The only way to get the custom operator to be used is to explicitly cast the literal:

```
SELECT * FROM mytable WHERE val = text 'foo';
```

so that the `mytext = text` operator is found immediately according to the exact-match rule. If the best-match rules are reached, they actively discriminate against operators on domain types. If they did not, such an operator would create too many ambiguous-operator failures, because the casting rules always consider a domain as castable to or from its base type, and so the domain operator would be considered usable in all the same cases as a similarly-named operator on the base type.

10.3. Functions

The specific function that is referenced by a function call is determined using the following procedure.

Function Type Resolution

1. Select the functions to be considered from the `pg_proc` system catalog. If a non-schema-qualified function name was used, the functions considered are those with the matching name and argument count that are visible in the current search path (see Section 5.10.3). If a qualified function name was given, only functions in the specified schema are considered.

    a. (Optional) If the search path finds multiple functions of identical argument types, only the one appearing earliest in the path is considered. Functions of different argument types are considered on an equal footing regardless of search path position.

    b. (Optional) If a function is declared with a `VARIADIC` array parameter, and the call does not use the `VARIADIC` keyword, then the function is treated as if the array parameter were replaced by one or more occurrences of its element type, as needed to match the call. After such expansion the function might have effective argument types identical to some non-variadic function. In that case the function appearing earlier in the search path is used, or if the two functions are in the same schema, the non-variadic one is preferred.

        This creates a security hazard when calling, via qualified name [2], a variadic function found in a schema that permits untrusted users to create objects. A malicious user can take control and execute arbitrary SQL functions as though you executed them. Substitute a call bearing the `VARIADIC` keyword, which bypasses this hazard. Calls populating `VARIADIC "any"` parameters often have no equivalent formulation containing the `VARIADIC` keyword. To issue those calls safely, the function's schema must permit only trusted users to create objects.

    c. (Optional) Functions that have default values for parameters are considered to match any call that omits zero or more of the defaultable parameter positions. If more than one such function matches a call, the one appearing earliest in the search path is used. If there are two or more such functions in the same schema with identical parameter types in the non-

[2] The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a secure schema usage pattern.

defaulted positions (which is possible if they have different sets of defaultable parameters), the system will not be able to determine which to prefer, and so an “ambiguous function call” error will result if no better match to the call can be found.

This creates an availability hazard when calling, via qualified name[2], any function found in a schema that permits untrusted users to create objects. A malicious user can create a function with the name of an existing function, replicating that function's parameters and appending novel parameters having default values. This precludes new calls to the original function. To forestall this hazard, place functions in schemas that permit only trusted users to create objects.

2. Check for a function accepting exactly the input argument types. If one exists (there can be only one exact match in the set of functions considered), use it. Lack of an exact match creates a security hazard when calling, via qualified name[2], a function found in a schema that permits untrusted users to create objects. In such situations, cast arguments to force an exact match. (Cases involving `unknown` will never find a match at this step.)

3. If no exact match is found, see if the function call appears to be a special type conversion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal, or a type that is binary-coercible to the named data type, or a type that could be converted to the named data type by applying that type's I/O functions (that is, the conversion is either to or from one of the standard string types). When these conditions are met, the function call is treated as a form of `CAST` specification. [3]

4. Look for the best match.

   a. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. `unknown` literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

   b. If any input argument is of a domain type, treat it as being of the domain's base type for all subsequent steps. This ensures that domains act like their base types for purposes of ambiguous-function resolution.

   c. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have exact matches. If only one candidate remains, use it; else continue to the next step.

   d. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

   e. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.

   f. If there are both `unknown` and known-type arguments, and all the known-type arguments have the same type, assume that the `unknown` arguments are also of that type, and check

[3] The reason for this step is to support function-style cast specifications in cases where there is not an actual cast function. If there is a cast function, it is conventionally named after its output type, and so there is no need to have a special case. See CREATE CAST for additional commentary.

which candidates can accept that type at the unknown-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Note that the “best match” rules are identical for operator and function type resolution. Some examples follow.

Example 10.6. Rounding Function Argument Type Resolution

There is only one round function that takes two arguments; it takes a first argument of type numeric and a second argument of type integer. So the following query automatically converts the first argument of type integer to numeric:

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

That query is actually transformed by the parser to:

```
SELECT round(CAST (4 AS numeric), 4);
```

Since numeric constants with decimal points are initially assigned the type numeric, the following query will require no type conversion and therefore might be slightly more efficient:

```
SELECT round(4.0, 4);
```

Example 10.7. Variadic Function Resolution

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS
 int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

This function accepts, but does not require, the VARIADIC keyword. It tolerates both integer and numeric arguments:

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

However, the first and second calls will prefer more-specific functions, if available:

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
```

```
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

Given the default configuration and only the first function existing, the first and second calls are insecure. Any user could intercept them by creating the second or third function. By matching the argument type exactly and using the `VARIADIC` keyword, the third call is secure.

Example 10.8. Substring Function Type Resolution

There are several `substr` functions, one of which takes types `text` and `integer`. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category `string` (namely of type `text`).

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to convert it to become `text`:

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

This is transformed by the parser to effectively become:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Note**
>
> The parser learns from the `pg_cast` catalog that `text` and `varchar` are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no type conversion call is really inserted in this case.

And, if the function is called with an argument of type `integer`, the parser will try to convert that to `text`:

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
```

```
HINT:  No function matches the given name and argument types. You
 might need
to add explicit type casts.
```

This does not work because `integer` does not have an implicit cast to `text`. An explicit cast will work, however:

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. Value Storage

Values to be inserted into a table are converted to the destination column's data type according to the following steps.

Value Storage Type Conversion

1. Check for an exact match with the target.

2. Otherwise, try to convert the expression to the target type. This is possible if an *assignment cast* between the two types is registered in the `pg_cast` catalog (see CREATE CAST). Alternatively, if the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.

3. Check to see if there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the `pg_cast` catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type `integer`, which receives the destination column's `atttypmod` value (typically its declared length, although the interpretation of `atttypmod` varies for different data types), and it may take a third `boolean` parameter that says whether the cast is explicit or implicit. The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

Example 10.9. `character` Storage Type Conversion

For a target column declared as `character(20)` the following statement shows that the stored value is sized correctly:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

What has really happened here is that the two unknown literals are resolved to `text` by default, allowing the `||` operator to be resolved as `text` concatenation. Then the `text` result of the operator is converted to `bpchar` (“blank-padded char”, the internal name of the `character` data type) to match the target column type. (Since the conversion from `text` to `bpchar` is binary-coercible, this conversion does not insert any real function call.) Finally, the sizing function `bpchar(bpchar,`

integer, boolean) is found in the system catalog and applied to the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

10.5. UNION, CASE, and Related Constructs

SQL UNION constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The INTERSECT and EXCEPT constructs resolve dissimilar types in the same way as UNION. Some other constructs, including CASE, ARRAY, VALUES, and the GREATEST and LEAST functions, use the identical algorithm to match up their component expressions and select a result data type.

Type Resolution for UNION, CASE, and Related Constructs

1. If all inputs are of the same type, and it is not unknown, resolve as that type.

2. If any input is of a domain type, treat it as being of the domain's base type for all subsequent steps. [4]

3. If all inputs are of type unknown, resolve as type text (the preferred type of the string category). Otherwise, unknown inputs are ignored for the purposes of the remaining rules.

4. If the non-unknown inputs are not all of the same type category, fail.

5. Select the first non-unknown input type as the candidate type, then consider each other non-unknown input type, left to right. [5] If the candidate type can be implicitly converted to the other type, but not vice-versa, select the other type as the new candidate type. Then continue considering the remaining inputs. If, at any stage of this process, a preferred type is selected, stop considering additional inputs.

6. Convert all inputs to the final candidate type. Fail if there is not an implicit conversion from a given input type to the candidate type.

Some examples follow.

Example 10.10. Type Resolution with Underspecified Types in a Union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal 'b' will be resolved to type text.

Example 10.11. Type Resolution in a Simple Union

```
SELECT 1.2 AS "numeric" UNION SELECT 1;
```

[4] Somewhat like the treatment of domain inputs for operators and functions, this behavior allows a domain type to be preserved through a UNION or similar construct, so long as the user is careful to ensure that all inputs are implicitly or explicitly of that exact type. Otherwise the domain's base type will be used.

[5] For historical reasons, CASE treats its ELSE clause (if any) as the “first” input, with the THEN clauses(s) considered after that. In all other cases, “left to right” means the order in which the expressions appear in the query text.

```
 numeric
---------
       1
     1.2
(2 rows)
```

The literal `1.2` is of type `numeric`, and the `integer` value `1` can be cast implicitly to `numeric`, so that type is used.

Example 10.12. Type Resolution in a Transposed Union

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Here, since type `real` cannot be implicitly cast to `integer`, but `integer` can be implicitly cast to `real`, the union result type is resolved as `real`.

Example 10.13. Type Resolution in a Nested Union

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

This failure occurs because PostgreSQL treats multiple `UNION`s as a nest of pairwise operations; that is, this input is the same as

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

The inner `UNION` is resolved as emitting type `text`, according to the rules given above. Then the outer `UNION` has inputs of types `text` and `integer`, leading to the observed error. The problem can be fixed by ensuring that the leftmost `UNION` has at least one input of the desired result type.

`INTERSECT` and `EXCEPT` operations are likewise resolved pairwise. However, the other constructs described in this section consider all of their inputs in one resolution step.

10.6. SELECT Output Columns

The rules given in the preceding sections will result in assignment of non-`unknown` data types to all expressions in an SQL query, except for unspecified-type literals that appear as simple output columns of a `SELECT` command. For example, in

```
SELECT 'Hello World';
```

there is nothing to identify what type the string literal should be taken as. In this situation PostgreSQL will fall back to resolving the literal's type as `text`.

When the `SELECT` is one arm of a `UNION` (or `INTERSECT` or `EXCEPT`) construct, or when it appears within `INSERT ... SELECT`, this rule is not applied since rules given in preceding sections take precedence. The type of an unspecified-type literal can be taken from the other `UNION` arm in the first case, or from the destination column in the second case.

`RETURNING` lists are treated the same as `SELECT` output lists for this purpose.

> **Note**
>
> Prior to PostgreSQL 10, this rule did not exist, and unspecified-type literals in a `SELECT` output list were left as type `unknown`. That had assorted bad consequences, so it's been changed.

Chapter 11. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

11.1. Introduction

Suppose we have a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application issues many queries of the form:

```
SELECT content FROM test1 WHERE id = constant;
```

With no advance preparation, the system would have to scan the entire `test1` table, row by row, to find all matching entries. If there are many rows in `test1` and only a few rows (perhaps zero or one) that would be returned by such a query, this is clearly an inefficient method. But if the system has been instructed to maintain an index on the `id` column, it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most non-fiction books: terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page(s), rather than having to read the entire book to find the material of interest. Just as it is the task of the author to anticipate the items that readers are likely to look up, it is the task of the database programmer to foresee which indexes will be useful.

The following command can be used to create an index on the `id` column, as discussed:

```
CREATE INDEX test1_id_index ON test1 (id);
```

The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the `DROP INDEX` command. Indexes can be added to and removed from tables at any time.

Once an index is created, no further intervention is required: the system will update the index when the table is modified, and it will use the index in queries when it thinks doing so would be more efficient than a sequential table scan. But you might have to run the `ANALYZE` command regularly to update statistics to allow the query planner to make educated decisions. See Chapter 14 for information about how to find out whether an index is used and when and why the planner might choose *not* to use an index.

Indexes can also benefit `UPDATE` and `DELETE` commands with search conditions. Indexes can moreover be used in join searches. Thus, an index defined on a column that is part of a join condition can also significantly speed up queries with joins.

In general, PostgreSQL indexes can be used to optimize queries that contain one or more `WHERE` or `JOIN` clauses of the form

```
indexed-column indexable-operator comparison-value
```

Here, the *indexed-column* is whatever column or expression the index has been defined on. The *indexable-operator* is an operator that is a member of the index's *operator class* for the indexed column. (More details about that appear below.) And the *comparison-value* can be any expression that is not volatile and does not reference the index's table.

In some cases the query planner can extract an indexable clause of this form from another SQL construct. A simple example is that if the original clause was

```
comparison-value operator indexed-column
```

then it can be flipped around into indexable form if the original *operator* has a commutator operator that is a member of the index's operator class.

Creating an index on a large table can take a long time. By default, PostgreSQL allows reads (`SELECT` statements) to occur on the table in parallel with index creation, but writes (`INSERT`, `UPDATE`, `DELETE`) are blocked until the index build is finished. In production environments this is often unacceptable. It is possible to allow writes to occur in parallel with index creation, but there are several caveats to be aware of — for more information see Building Indexes Concurrently.

After an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Indexes can also prevent the creation of heap-only tuples. Therefore indexes that are seldom or never used in queries should be removed.

11.2. Index Types

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN, BRIN, and the extension bloom. Each index type uses a different algorithm that is best suited to different types of indexable clauses. By default, the `CREATE INDEX` command creates B-tree indexes, which fit the most common situations. The other index types are selected by writing the keyword `USING` followed by the index type name. For example, to create a Hash index:

```
CREATE INDEX name ON table USING HASH (column);
```

11.2.1. B-Tree

B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

```
<   <=   =   >=   >
```

Constructs equivalent to combinations of these operators, such as `BETWEEN` and `IN`, can also be implemented with a B-tree index search. Also, an `IS NULL` or `IS NOT NULL` condition on an index column can be used with a B-tree index.

The optimizer can also use a B-tree index for queries involving the pattern matching operators `LIKE` and `~` *if* the pattern is a constant and is anchored to the beginning of the string — for example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`. However, if your database does not use the C locale you will need to create the index with a special operator class to support indexing of pattern-matching queries; see Section 11.10 below. It is also possible to use B-tree indexes for `ILIKE` and `~*`, but only if the pattern starts with non-alphabetic characters, i.e., characters that are not affected by upper/lower case conversion.

B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scan and sort, but it is often helpful.

11.2.2. Hash

Hash indexes store a 32-bit hash code derived from the value of the indexed column. Hence, such indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the equal operator:

```
=
```

11.2.3. GiST

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the *operator class*). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

```
<<   &<   &>   >>   <<|   &<|   |&>   |>>   @>   <@   ~=   &&
```

(See Section 9.11 for the meaning of these operators.) The GiST operator classes included in the standard distribution are documented in Table 64.1. Many other GiST operator classes are available in the `contrib` collection or as separate projects. For more information see Section 64.2.

GiST indexes are also capable of optimizing "nearest-neighbor" searches, such as

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT
 10;
```

which finds the ten places closest to a given target point. The ability to do this is again dependent on the particular operator class being used. In Table 64.1, operators that can be used in this way are listed in the column "Ordering Operators".

11.2.4. SP-GiST

SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). As an example, the standard distribution of PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries using these operators:

```
<<   >>   ~=   <@   <<|   |>>
```

(See Section 9.11 for the meaning of these operators.) The SP-GiST operator classes included in the standard distribution are documented in Table 64.2. For more information see Section 64.3.

Like GiST, SP-GiST supports "nearest-neighbor" searches. For SP-GiST operator classes that support distance ordering, the corresponding operator is listed in the "Ordering Operators" column in Table 64.2.

11.2.5. GIN

GIN indexes are “inverted indexes” which are appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.

Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies, and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes a GIN operator class for arrays, which supports indexed queries using these operators:

```
<@    @>    =    &&
```

(See Section 9.19 for the meaning of these operators.) The GIN operator classes included in the standard distribution are documented in Table 64.3. Many other GIN operator classes are available in the `contrib` collection or as separate projects. For more information see Section 64.4.

11.2.6. BRIN

BRIN indexes (a shorthand for Block Range INdexes) store summaries about the values stored in consecutive physical block ranges of a table. Thus, they are most effective for columns whose values are well-correlated with the physical order of the table rows. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range. This supports indexed queries using these operators:

```
<    <=    =    >=    >
```

The BRIN operator classes included in the standard distribution are documented in Table 64.4. For more information see Section 64.5.

11.3. Multicolumn Indexes

An index can be defined on more than one column of a table. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(say, you keep your `/dev` directory in a database...) and you frequently issue queries like:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it might be appropriate to define an index on the columns `major` and `minor` together, e.g.:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Currently, only the B-tree, GiST, GIN, and BRIN index types support multiple-key-column indexes. Whether there can be multiple key columns is independent of whether `INCLUDE` columns can be added to the index. Indexes can have up to 32 columns, including `INCLUDE` columns. (This limit can be altered when building PostgreSQL; see the file `pg_config_manual.h`.)

A multicolumn B-tree index can be used with query conditions that involve any subset of the index's columns, but the index is most efficient when there are constraints on the leading (leftmost) columns. The exact rule is that equality constraints on leading columns, plus any inequality constraints on the first column that does not have an equality constraint, will be used to limit the portion of the index that is scanned. Constraints on columns to the right of these columns are checked in the index, so they save visits to the table proper, but they do not reduce the portion of the index that has to be scanned. For example, given an index on `(a, b, c)` and a query condition `WHERE a = 5 AND b >= 42 AND c < 77`, the index would have to be scanned from the first entry with `a` = 5 and `b` = 42 up through the last entry with `a` = 5. Index entries with `c` >= 77 would be skipped, but they'd still have to be scanned through. This index could in principle be used for queries that have constraints on `b` and/or `c` with no constraint on `a` — but the entire index would have to be scanned, so in most cases the planner would prefer a sequential table scan over using the index.

A multicolumn GiST index can be used with query conditions that involve any subset of the index's columns. Conditions on additional columns restrict the entries returned by the index, but the condition on the first column is the most important one for determining how much of the index needs to be scanned. A GiST index will be relatively ineffective if its first column has only a few distinct values, even if there are many distinct values in additional columns.

A multicolumn GIN index can be used with query conditions that involve any subset of the index's columns. Unlike B-tree or GiST, index search effectiveness is the same regardless of which index column(s) the query conditions use.

A multicolumn BRIN index can be used with query conditions that involve any subset of the index's columns. Like GIN and unlike B-tree or GiST, index search effectiveness is the same regardless of which index column(s) the query conditions use. The only reason to have multiple BRIN indexes instead of one multicolumn BRIN index on a single table is to have a different `pages_per_range` storage parameter.

Of course, each column must be used with operators appropriate to the index type; clauses that involve other operators will not be considered.

Multicolumn indexes should be used sparingly. In most situations, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are unlikely to be helpful unless the usage of the table is extremely stylized. See also Section 11.5 and Section 11.9 for some discussion of the merits of different index configurations.

11.4. Indexes and `ORDER BY`

In addition to simply finding the rows to be returned by a query, an index may be able to deliver them in a specific sorted order. This allows a query's `ORDER BY` specification to be honored without a separate sorting step. Of the index types currently supported by PostgreSQL, only B-tree can produce sorted output — the other index types return matching rows in an unspecified, implementation-dependent order.

The planner will consider satisfying an `ORDER BY` specification either by scanning an available index that matches the specification, or by scanning the table in physical order and doing an explicit sort. For a query that requires scanning a large fraction of the table, an explicit sort is likely to be faster than using an index because it requires less disk I/O due to following a sequential access pattern. Indexes are more useful when only a few rows need be fetched. An important special case is `ORDER BY` in combination with `LIMIT` *n*: an explicit sort will have to process all the data to identify the first *n* rows, but if there is an index matching the `ORDER BY`, the first *n* rows can be retrieved directly, without scanning the remainder at all.

By default, B-tree indexes store their entries in ascending order with nulls last (table TID is treated as a tiebreaker column among otherwise equal entries). This means that a forward scan of an index on column `x` produces output satisfying `ORDER BY x` (or more verbosely, `ORDER BY x ASC NULLS LAST`). The index can also be scanned backward, producing output satisfying `ORDER BY`

`x DESC` (or more verbosely, `ORDER BY x DESC NULLS FIRST`, since `NULLS FIRST` is the default for `ORDER BY DESC`).

You can adjust the ordering of a B-tree index by including the options `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` when creating the index; for example:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

An index stored in ascending order with nulls first can satisfy either `ORDER BY x ASC NULLS FIRST` or `ORDER BY x DESC NULLS LAST` depending on which direction it is scanned in.

You might wonder why bother providing all four options, when two options together with the possibility of backward scan would cover all the variants of `ORDER BY`. In single-column indexes the options are indeed redundant, but in multicolumn indexes they can be useful. Consider a two-column index on `(x, y)`: this can satisfy `ORDER BY x, y` if we scan forward, or `ORDER BY x DESC, y DESC` if we scan backward. But it might be that the application frequently needs to use `ORDER BY x ASC, y DESC`. There is no way to get that ordering from a plain index, but it is possible if the index is defined as `(x ASC, y DESC)` or `(x DESC, y ASC)`.

Obviously, indexes with non-default sort orderings are a fairly specialized feature, but sometimes they can produce tremendous speedups for certain queries. Whether it's worth maintaining such an index depends on how often you use queries that require a special sort ordering.

11.5. Combining Multiple Indexes

A single index scan can only use query clauses that use the index's columns with operators of its operator class and are joined with `AND`. For example, given an index on `(a, b)` a query condition like `WHERE a = 5 AND b = 6` could use the index, but a query like `WHERE a = 5 OR b = 6` could not directly use the index.

Fortunately, PostgreSQL has the ability to combine multiple indexes (including multiple uses of the same index) to handle cases that cannot be implemented by single index scans. The system can form `AND` and `OR` conditions across several index scans. For example, a query like `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` could be broken down into four separate scans of an index on `x`, each scan using one of the query clauses. The results of these scans are then ORed together to produce the result. Another example is that if we have separate indexes on `x` and `y`, one possible implementation of a query like `WHERE x = 5 AND y = 6` is to use each index with the appropriate query clause and then AND together the index results to identify the result rows.

To combine multiple indexes, the system scans each needed index and prepares a *bitmap* in memory giving the locations of table rows that are reported as matching that index's conditions. The bitmaps are then ANDed and ORed together as needed by the query. Finally, the actual table rows are visited and returned. The table rows are visited in physical order, because that is how the bitmap is laid out; this means that any ordering of the original indexes is lost, and so a separate sort step will be needed if the query has an `ORDER BY` clause. For this reason, and because each additional index scan adds extra time, the planner will sometimes choose to use a simple index scan even though additional indexes are available that could have been used as well.

In all but the simplest applications, there are various combinations of indexes that might be useful, and the database developer must make trade-offs to decide which indexes to provide. Sometimes multicolumn indexes are best, but sometimes it's better to create separate indexes and rely on the index-combination feature. For example, if your workload includes a mix of queries that sometimes involve only column `x`, sometimes only column `y`, and sometimes both columns, you might choose to create two separate indexes on `x` and `y`, relying on index combination to process the queries that use both columns. You could also create a multicolumn index on `(x, y)`. This index would typically be more efficient than index combination for queries involving both columns, but as discussed in

Section 11.3, it would be almost useless for queries involving only `y`, so it should not be the only index. A combination of the multicolumn index and a separate index on `y` would serve reasonably well. For queries involving only `x`, the multicolumn index could be used, though it would be larger and hence slower than an index on `x` alone. The last alternative is to create all three indexes, but this is probably only reasonable if the table is searched much more often than it is updated and all three types of query are common. If one of the types of query is much less common than the others, you'd probably settle for creating just the two indexes that best match the common types.

11.6. Unique Indexes

Indexes can also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
CREATE UNIQUE INDEX name ON table (column [, ...]) [ NULLS [ NOT ]
 DISTINCT ];
```

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values are not allowed. By default, null values in a unique column are not considered equal, allowing multiple nulls in the column. The `NULLS NOT DISTINCT` option modifies this and causes the index to treat nulls as equal. A multicolumn unique index will only reject cases where all indexed columns are equal in multiple rows.

PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table. The index covers the columns that make up the primary key or unique constraint (a multicolumn index, if appropriate), and is the mechanism that enforces the constraint.

> **Note**
>
> There's no need to manually create indexes on unique columns; doing so would just duplicate the automatically-created index.

11.7. Indexes on Expressions

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index if one has been defined on the result of the `lower(col1)` function:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

If we were to declare this index `UNIQUE`, it would prevent creation of rows whose `col1` values differ only in case, as well as rows whose `col1` values are actually identical. Thus, indexes on expressions can be used to enforce constraints that are not definable as simple unique constraints.

As another example, if one often does queries like:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John
 Smith';
```

then it might be worth creating an index like this:

```
CREATE INDEX people_names ON people ((first_name || ' ' ||
 last_name));
```

The syntax of the `CREATE INDEX` command normally requires writing parentheses around index expressions, as shown in the second example. The parentheses can be omitted when the expression is just a function call, as in the first example.

Index expressions are relatively expensive to maintain, because the derived expression(s) must be computed for each row insertion and non-HOT update. However, the index expressions are *not* recomputed during an indexed search, since they are already stored in the index. In both examples above, the system sees the query as just `WHERE indexedcolumn = 'constant'` and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

11.8. Partial Indexes

A *partial index* is an index built over a subset of a table; the subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries only for those table rows that satisfy the predicate. Partial indexes are a specialized feature, but there are several situations in which they are useful.

One major reason for using a partial index is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up those queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases. Example 11.1 shows a possible application of this idea.

Example 11.1. Setting up a Partial Index to Exclude Common Values

Suppose you are storing web server access logs in a database. Most accesses originate from the IP address range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet.

Assume a table like this:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

To create a partial index that suits our example, use a command such as this:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

A typical query that can use this index would be:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Here the query's IP address is covered by the partial index. The following query cannot use the partial index, as it uses an IP address that is excluded from the index:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

Observe that this kind of partial index requires that the common values be predetermined, so such partial indexes are best used for data distributions that do not change. Such indexes can be recreated occasionally to adjust for new data distributions, but this adds maintenance effort.

Another possible use for a partial index is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 11.2. This results in the same advantages as listed above, but it prevents the “uninteresting” values from being accessed via that index, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

Example 11.2. Setting up a Partial Index to Exclude Uninteresting Values

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

A possible query to use this index would be:

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

However, the index can also be used in queries that do not involve `order_nr` at all, e.g.:

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

This is not as efficient as a partial index on the `amount` column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

The order 3501 might be among the billed or unbilled orders.

Example 11.2 also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the `WHERE` condition of the query mathematically implies the predicate of the index. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent expressions that are written in different forms.

(Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example “x < 1” implies “x < 2”; otherwise the predicate condition must exactly match part of the query's `WHERE` condition or the index will not be recognized as usable. Matching takes place at query planning time, not at run time. As a result, parameterized query clauses do not work with a partial index. For example a prepared query with a parameter might specify “x < ?” which will never imply “x < 2” for all possible values of the parameter.

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in Example 11.3. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 11.3. Setting up a Partial Unique Index

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one “successful” entry for a given subject and target combination, but there might be any number of “unsuccessful” entries. Here is one way to do it:

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject,
 target)
    WHERE success;
```

This is a particularly efficient approach when there are few successful tests and many unsuccessful ones. It is also possible to allow only one null in a column by creating a unique partial index with an `IS NULL` restriction.

Finally, a partial index can also be used to override the system's query plan choices. Also, data sets with peculiar distributions might cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will be minimal. There are cases where they are quite counterproductive, as in Example 11.4.

Example 11.4. Do Not Use Partial Indexes as a Substitute for Partitioning

You might be tempted to create a large set of non-overlapping partial indexes, for example

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

This is a bad idea! Almost certainly, you'll be better off with a single non-partial index, declared like

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

(Put the category column first, for the reasons described in Section 11.3.) While a search in this larger index might have to descend through a couple more tree levels than a search in a smaller index, that's almost certainly going to be cheaper than the planner effort needed to select the appropriate one of the partial indexes. The core of the problem is that the system does not understand the relationship among the partial indexes, and will laboriously test each one to see if it's applicable to the current query.

If your table is large enough that a single index really is a bad idea, you should look into using partitioning instead (see Section 5.12). With that mechanism, the system does understand that the tables and indexes are non-overlapping, so far better performance is possible.

More information about partial indexes can be found in [ston89b], [olson93], and [seshadri95].

11.9. Index-Only Scans and Covering Indexes

All indexes in PostgreSQL are *secondary* indexes, meaning that each index is stored separately from the table's main data area (which is called the table's *heap* in PostgreSQL terminology). This means that in an ordinary index scan, each row retrieval requires fetching data from both the index and the heap. Furthermore, while the index entries that match a given indexable `WHERE` condition are usually close together in the index, the table rows they reference might be anywhere in the heap. The heap-access portion of an index scan thus involves a lot of random access into the heap, which can be slow, particularly on traditional rotating media. (As described in Section 11.5, bitmap scans try to alleviate this cost by doing the heap accesses in sorted order, but that only goes so far.)

To solve this performance problem, PostgreSQL supports *index-only scans*, which can answer queries from an index alone without any heap access. The basic idea is to return values directly out of each index entry instead of consulting the associated heap entry. There are two fundamental restrictions on when this method can be used:

1. The index type must support index-only scans. B-tree indexes always do. GiST and SP-GiST indexes support index-only scans for some operator classes but not others. Other index types have no support. The underlying requirement is that the index must physically store, or else be able to reconstruct, the original data value for each index entry. As a counterexample, GIN indexes cannot support index-only scans because each index entry typically holds only part of the original data value.

2. The query must reference only columns stored in the index. For example, given an index on columns `x` and `y` of a table that also has a column `z`, these queries could use index-only scans:

   ```
   SELECT x, y FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND y < 42;
   ```

   but these queries could not:

   ```
   SELECT x, z FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND z < 42;
   ```

   (Expression indexes and partial indexes complicate this rule, as discussed below.)

If these two fundamental requirements are met, then all the data values required by the query are available from the index, so an index-only scan is physically possible. But there is an additional requirement for any table scan in PostgreSQL: it must verify that each retrieved row be “visible” to the query's MVCC snapshot, as discussed in Chapter 13. Visibility information is not stored in index entries, only in heap entries; so at first glance it would seem that every row retrieval would require a heap access anyway. And this is indeed the case, if the table row has been modified recently. How-

ever, for seldom-changing data there is a way around this problem. PostgreSQL tracks, for each page in a table's heap, whether all rows stored in that page are old enough to be visible to all current and future transactions. This information is stored in a bit in the table's *visibility map*. An index-only scan, after finding a candidate index entry, checks the visibility map bit for the corresponding heap page. If it's set, the row is known visible and so the data can be returned with no further work. If it's not set, the heap entry must be visited to find out whether it's visible, so no performance advantage is gained over a standard index scan. Even in the successful case, this approach trades visibility map accesses for heap accesses; but since the visibility map is four orders of magnitude smaller than the heap it describes, far less physical I/O is needed to access it. In most situations the visibility map remains cached in memory all the time.

In short, while an index-only scan is possible given the two fundamental requirements, it will be a win only if a significant fraction of the table's heap pages have their all-visible map bits set. But tables in which a large fraction of the rows are unchanging are common enough to make this type of scan very useful in practice.

To make effective use of the index-only scan feature, you might choose to create a *covering index*, which is an index specifically designed to include the columns needed by a particular type of query that you run frequently. Since queries typically need to retrieve more columns than just the ones they search on, PostgreSQL allows you to create an index in which some columns are just “payload” and are not part of the search key. This is done by adding an `INCLUDE` clause listing the extra columns. For example, if you commonly run queries like

```
SELECT y FROM tab WHERE x = 'key';
```

the traditional approach to speeding up such queries would be to create an index on `x` only. However, an index defined as

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

could handle these queries as index-only scans, because `y` can be obtained from the index without visiting the heap.

Because column `y` is not part of the index's search key, it does not have to be of a data type that the index can handle; it's merely stored in the index and is not interpreted by the index machinery. Also, if the index is a unique index, that is

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

the uniqueness condition applies to just column `x`, not to the combination of `x` and `y`. (An `INCLUDE` clause can also be written in `UNIQUE` and `PRIMARY KEY` constraints, providing alternative syntax for setting up an index like this.)

It's wise to be conservative about adding non-key payload columns to an index, especially wide columns. If an index tuple exceeds the maximum size allowed for the index type, data insertion will fail. In any case, non-key columns duplicate data from the index's table and bloat the size of the index, thus potentially slowing searches. And remember that there is little point in including payload columns in an index unless the table changes slowly enough that an index-only scan is likely to not need to access the heap. If the heap tuple must be visited anyway, it costs nothing more to get the column's value from there. Other restrictions are that expressions are not currently supported as included columns, and that only B-tree, GiST and SP-GiST indexes currently support included columns.

Before PostgreSQL had the `INCLUDE` feature, people sometimes made covering indexes by writing the payload columns as ordinary index columns, that is writing

```
CREATE INDEX tab_x_y ON tab(x, y);
```

even though they had no intention of ever using `y` as part of a `WHERE` clause. This works fine as long as the extra columns are trailing columns; making them be leading columns is unwise for the reasons explained in Section 11.3. However, this method doesn't support the case where you want the index to enforce uniqueness on the key column(s).

Suffix truncation always removes non-key columns from upper B-Tree levels. As payload columns, they are never used to guide index scans. The truncation process also removes one or more trailing key column(s) when the remaining prefix of key column(s) happens to be sufficient to describe tuples on the lowest B-Tree level. In practice, covering indexes without an `INCLUDE` clause often avoid storing columns that are effectively payload in the upper levels. However, explicitly defining payload columns as non-key columns *reliably* keeps the tuples in upper levels small.

In principle, index-only scans can be used with expression indexes. For example, given an index on `f(x)` where `x` is a table column, it should be possible to execute

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

as an index-only scan; and this is very attractive if `f()` is an expensive-to-compute function. However, PostgreSQL's planner is currently not very smart about such cases. It considers a query to be potentially executable by index-only scan only when all *columns* needed by the query are available from the index. In this example, `x` is not needed except in the context `f(x)`, but the planner does not notice that and concludes that an index-only scan is not possible. If an index-only scan seems sufficiently worthwhile, this can be worked around by adding `x` as an included column, for example

```
CREATE INDEX tab_f_x ON tab (f(x)) INCLUDE (x);
```

An additional caveat, if the goal is to avoid recalculating `f(x)`, is that the planner won't necessarily match uses of `f(x)` that aren't in indexable `WHERE` clauses to the index column. It will usually get this right in simple queries such as shown above, but not in queries that involve joins. These deficiencies may be remedied in future versions of PostgreSQL.

Partial indexes also have interesting interactions with index-only scans. Consider the partial index shown in Example 11.3:

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject,
 target)
    WHERE success;
```

In principle, we could do an index-only scan on this index to satisfy a query like

```
SELECT target FROM tests WHERE subject = 'some-subject' AND
 success;
```

But there's a problem: the `WHERE` clause refers to `success` which is not available as a result column of the index. Nonetheless, an index-only scan is possible because the plan does not need to recheck that part of the `WHERE` clause at run time: all entries found in the index necessarily have `success = true` so this need not be explicitly checked in the plan. PostgreSQL versions 9.6 and later will recognize such cases and allow index-only scans to be generated, but older versions will not.

11.10. Operator Classes and Operator Families

An index definition can specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [ ( opclass_options ) ]
 [sort options] [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type `int4` would use the `int4_ops` class; this operator class includes comparison functions for values of type `int4`. In practice the default operator class for the column's data type is usually sufficient. The main reason for having operator classes is that for some data types, there could be more than one meaningful index behavior. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. The operator class determines the basic sort ordering (which can then be modified by adding sort options `COLLATE`, `ASC`/`DESC` and/or `NULLS FIRST`/`NULLS LAST`).

There are also some built-in operator classes besides the default ones:

- The operator classes `text_pattern_ops`, `varchar_pattern_ops`, and `bpchar_pattern_ops` support B-tree indexes on the types `text`, `varchar`, and `char` respectively. The difference from the default operator classes is that the values are compared strictly character by character rather than according to the locale-specific collation rules. This makes these operator classes suitable for use by queries involving pattern matching expressions (`LIKE` or POSIX regular expressions) when the database does not use the standard “C” locale. As an example, you might index a `varchar` column like this:

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Note that you should also create an index with the default operator class if you want queries involving ordinary `<`, `<=`, `>`, or `>=` comparisons to use an index. Such queries cannot use the *`xxx`*`_pattern_ops` operator classes. (Ordinary equality comparisons can use these operator classes, however.) It is possible to create multiple indexes on the same column with different operator classes. If you do use the C locale, you do not need the *`xxx`*`_pattern_ops` operator classes, because an index with the default operator class is usable for pattern-matching queries in the C locale.

The following query shows all defined operator classes:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

An operator class is actually just a subset of a larger structure called an *operator family*. In cases where several data types have similar behaviors, it is frequently useful to define cross-data-type operators and allow these to work with indexes. To do this, the operator classes for each of the types must be grouped into the same operator family. The cross-type operators are members of the family, but are not associated with any single class within the family.

This expanded version of the previous query shows the operator family each operator class belongs to:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
```

```
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

This query shows all defined operator families and all the operators included in each family:

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

> **Tip**
>
> psql has commands `\dAc`, `\dAf`, and `\dAo`, which provide slightly more sophisticated versions of these queries.

11.11. Indexes and Collations

An index can support only one collation per index column. If multiple collations are of interest, multiple indexes may be needed.

Consider these statements:

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

The index automatically uses the collation of the underlying column. So a query of the form

```
SELECT * FROM test1c WHERE content > constant;
```

could use the index, because the comparison will by default use the collation of the column. However, this index cannot accelerate queries that involve some other collation. So if queries of the form, say,

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

are also of interest, an additional index could be created that supports the `"y"` collation, like this:

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE
 "y");
```

11.12. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance or tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage for an individual query is done with the EXPLAIN command; its application for this purpose is illustrated in

Section 14.1. It is also possible to gather overall statistics about index usage in a running server, as described in Section 27.2.

It is difficult to formulate a general procedure for determining which indexes to create. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation is often necessary. The rest of this section gives some tips for that:

- Always run ANALYZE first. This command collects statistics about the distribution of the values in the table. This information is required to estimate the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application's index usage without having run `ANALYZE` is therefore a lost cause. See Section 24.1.3 and Section 24.1.6 for more information.

- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

  It is especially fatal to use very small test data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

  Also be careful when making up test data, which is often unavoidable when the application is not yet in production. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (see Section 19.7.1). For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join then there is probably a more fundamental reason why the index is not being used; for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)

- If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The `EXPLAIN ANALYZE` command can be useful here.

- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs estimated for the plan nodes can be adjusted via run-time parameters (described in Section 19.7.2). An inaccurate selectivity estimate is due to insufficient statistics. It might be possible to improve this by tuning the statistics-gathering parameters (see ALTER TABLE).

  If you do not succeed in adjusting the costs to be more appropriate, then you might have to resort to forcing index usage explicitly. You might also want to contact the PostgreSQL developers to examine the issue.

Chapter 12. Full Text Search

12.1. Introduction

Full Text Searching (or just *text search*) provides the capability to identify natural-language *documents* that satisfy a *query*, and optionally to sort them by relevance to the query. The most common type of search is to find all documents containing given *query terms* and return them in order of their *similarity* to the query. Notions of `query` and `similarity` are very flexible and depend on the specific application. The simplest search considers `query` as a set of words and `similarity` as the frequency of query words in the document.

Textual search operators have existed in databases for years. PostgreSQL has `~`, `~*`, `LIKE`, and `ILIKE` operators for textual data types, but they lack many essential properties required by modern information systems:

- There is no linguistic support, even for English. Regular expressions are not sufficient because they cannot easily handle derived words, e.g., `satisfies` and `satisfy`. You might miss documents that contain `satisfies`, although you probably would like to find them when searching for `satisfy`. It is possible to use `OR` to search for multiple derived forms, but this is tedious and error-prone (some words can have several thousand derivatives).
- They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
- They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be *preprocessed* and an index saved for later rapid searching. Preprocessing includes:

Parsing documents into tokens. It is useful to identify various classes of tokens, e.g., numbers, words, complex words, email addresses, so that they can be processed differently. In principle token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes. PostgreSQL uses a *parser* to perform this step. A standard parser is provided, and custom parsers can be created for specific needs.

Converting tokens into lexemes. A lexeme is a string, just like a token, but it has been *normalized* so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often involves removal of suffixes (such as `s` or `es` in English). This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates *stop words*, which are words that are so common that they are useless for searching. (In short, then, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) PostgreSQL uses *dictionaries* to perform this step. Various standard dictionaries are provided, and custom ones can be created for specific needs.

Storing preprocessed documents optimized for searching. For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes it is often desirable to store positional information to use for *proximity ranking*, so that a document that contains a more "dense" region of query words is assigned a higher rank than one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can:

- Define stop words that should not be indexed.
- Map synonyms to a single word using Ispell.
- Map phrases to a single word using a thesaurus.
- Map different variations of a word to a canonical form using an Ispell dictionary.

- Map different variations of a word to a canonical form using Snowball stemmer rules.

A data type `tsvector` is provided for storing preprocessed documents, along with a type `tsquery` for representing processed queries (Section 8.11). There are many functions and operators available for these data types (Section 9.13), the most important of which is the match operator `@@`, which we introduce in Section 12.1.2. Full text searches can be accelerated using indexes (Section 12.9).

12.1.1. What Is a Document?

A *document* is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (key words) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within PostgreSQL, a document is normally a textual field within a row of a database table, or possibly a combination (concatenation) of such fields, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body
 AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' ||
 d.body AS document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

> **Note**
>
> Actually, in these example queries, `coalesce` should be used to prevent a single `NULL` attribute from causing a `NULL` result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires superuser permissions or special function support, so this is usually less convenient than keeping all the data inside PostgreSQL. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed `tsvector` format. Searching and ranking are performed entirely on the `tsvector` representation of a document — the original text need only be retrieved when the document has been selected for display to a user. We therefore often speak of the `tsvector` as being the document, but of course it is only a compact representation of the full document.

12.1.2. Basic Text Matching

Full text searching in PostgreSQL is based on the match operator `@@`, which returns `true` if a `tsvector` (document) matches a `tsquery` (query). It doesn't matter which data type is written first:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat
 & rat'::tsquery;
 ?column?
```

```
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a
 fat rat'::tsvector;
 ?column?
----------
 f
```

As the above example suggests, a `tsquery` is not just raw text, any more than a `tsvector` is. A `tsquery` contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, NOT, and FOLLOWED BY operators. (For syntax details see Section 8.11.2.) There are functions `to_tsquery`, `plainto_tsquery`, and `phraseto_tsquery` that are helpful in converting user-written text into a proper `tsquery`, primarily by normalizing words appearing in the text. Similarly, `to_tsvector` is used to parse and normalize a document string. So in practice a text search match would look more like this:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat &
 rat');
 ?column?
----------
 t
```

Observe that this match would not succeed if written as

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat &
 rat');
 ?column?
----------
 f
```

since here no normalization of the word `rats` will occur. The elements of a `tsvector` are lexemes, which are assumed already normalized, so `rats` does not match `rat`.

The `@@` operator also supports `text` input, allowing explicit conversion of a text string to `tsvector` or `tsquery` to be skipped in simple cases. The variants available are:

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

The first two of these we saw already. The form `text @@ tsquery` is equivalent to `to_tsvector(x) @@ y`. The form `text @@ text` is equivalent to `to_tsvector(x) @@ plainto_tsquery(y)`.

Within a `tsquery`, the `&` (AND) operator specifies that both its arguments must appear in the document to have a match. Similarly, the `|` (OR) operator specifies that at least one of its arguments must appear, while the `!` (NOT) operator specifies that its argument must *not* appear in order to have a match. For example, the query `fat & ! rat` matches documents that contain `fat` but not `rat`.

Searching for phrases is possible with the help of the `<->` (FOLLOWED BY) `tsquery` operator, which matches only if its arguments have matches that are adjacent and in the given order. For example:

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
```

```
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <->
 error');
 ?column?
----------
 f
```

There is a more general version of the FOLLOWED BY operator having the form `<N>`, where *N* is an integer standing for the difference between the positions of the matching lexemes. `<1>` is the same as `<->`, while `<2>` allows exactly one other lexeme to appear between the matches, and so on. The `phraseto_tsquery` function makes use of this operator to construct a `tsquery` that can match a multi-word phrase when some of the words are stop words. For example:

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

A special case that's sometimes useful is that `<0>` can be used to require that two patterns match the same word.

Parentheses can be used to control nesting of the `tsquery` operators. Without parentheses, `|` binds least tightly, then `&`, then `<->`, and `!` most tightly.

It's worth noticing that the AND/OR/NOT operators mean something subtly different when they are within the arguments of a FOLLOWED BY operator than when they are not, because within FOLLOWED BY the exact position of the match is significant. For example, normally `!x` matches only documents that do not contain `x` anywhere. But `!x <-> y` matches `y` if it is not immediately after an `x`; an occurrence of `x` elsewhere in the document does not prevent a match. Another example is that `x & y` normally only requires that `x` and `y` both appear somewhere in the document, but `(x & y) <-> z` requires `x` and `y` to match at the same place, immediately before a `z`. Thus this query behaves differently from `x <-> z & y <-> z`, which will match a document containing two separate sequences `x z` and `y z`. (This specific query is useless as written, since `x` and `y` could not match at the same place; but with more complex situations such as prefix-match patterns, a query of this form could be useful.)

12.1.3. Configurations

The above are all simple text search examples. As mentioned before, full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, e.g., parse based on more than just white space. This functionality is controlled by *text search configurations*. PostgreSQL comes with predefined configurations for many languages, and you can easily create your own configurations. (psql's `\dF` command shows all available configurations.)

During installation an appropriate configuration is selected and default_text_search_config is set accordingly in `postgresql.conf`. If you are using the same text search configuration for the entire cluster you can use the value in `postgresql.conf`. To use different configurations throughout the cluster but the same configuration within any one database, use `ALTER DATABASE ... SET`. Otherwise, you can set `default_text_search_config` in each session.

Each text search function that depends on a configuration has an optional `regconfig` argument, so that the configuration to use can be specified explicitly. `default_text_search_config` is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. PostgreSQL's text search facility provides four types of configuration-related database objects:

- *Text search parsers* break documents into tokens and classify each token (for example, as words or numbers).
- *Text search dictionaries* convert tokens to normalized form and reject stop words.
- *Text search templates* provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- *Text search configurations* select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

Text search parsers and templates are built from low-level C functions; therefore it requires C programming ability to develop new ones, and superuser privileges to install one into a database. (There are examples of add-on parsers and templates in the `contrib/` area of the PostgreSQL distribution.) Since dictionaries and configurations just parameterize and connect together some underlying parsers and templates, no special privilege is needed to create a new dictionary or configuration. Examples of creating custom dictionaries and configurations appear later in this chapter.

12.2. Tables and Indexes

The examples in the previous section illustrated full text matching using simple constant strings. This section shows how to search table data, optionally using indexes.

12.2.1. Searching a Table

It is possible to do a full text search without an index. A simple query to print the `title` of each row that contains the word `friend` in its `body` field is:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english',
 'friend');
```

This will also find related words such as `friends` and `friendly`, since all these are reduced to the same normalized lexeme.

The query above specifies that the `english` configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

This query will use the configuration set by default_text_search_config.

A more complex example is to select the ten most recent documents that contain `create` and `table` in the `title` or `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create &
 table')
```

```
ORDER BY last_mod_date DESC
LIMIT 10;
```

For clarity we omitted the `coalesce` function calls which would be needed to find rows that contain `NULL` in one of the two fields.

Although these queries will work without an index, most applications will find this approach too slow, except perhaps for occasional ad-hoc searches. Practical use of text searching usually requires creating an index.

12.2.2. Creating Indexes

We can create a GIN index (Section 12.9) to speed up text searches:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english',
 body));
```

Notice that the 2-argument version of `to_tsvector` is used. Only text search functions that specify a configuration name can be used in expression indexes (Section 11.7). This is because the index contents must be unaffected by default_text_search_config. If they were affected, the index contents might be inconsistent because different entries could contain `tsvector`s that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of `to_tsvector` was used in the index above, only a query reference that uses the 2-argument version of `to_tsvector` with the same configuration name will use that index. That is, `WHERE to_tsvector('english', body) @@ 'a & b'` can use the index, but `WHERE to_tsvector(body) @@ 'a & b'` cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

It is possible to set up more complex expression indexes wherein the configuration name is specified by another column, e.g.:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name,
 body));
```

where `config_name` is a column in the `pgweb` table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are meant to use the index must be phrased to match, e.g., `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Indexes can even concatenate columns:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english',
 title || ' ' || body));
```

Another approach is to create a separate `tsvector` column to hold the output of `to_tsvector`. To keep this column automatically up to date with its source data, use a stored generated column. This example is a concatenation of `title` and `body`, using `coalesce` to ensure that one field will still be indexed when the other is `NULL`:

```
ALTER TABLE pgweb
    ADD COLUMN textsearchable_index_col tsvector
               GENERATED ALWAYS AS (to_tsvector('english',
 coalesce(title, '') || ' ' || coalesce(body, ''))) STORED;
```

Then we create a GIN index to speed up the search:

```
CREATE INDEX textsearch_idx ON pgweb USING GIN
 (textsearchable_index_col);
```

Now we are ready to perform a fast full text search:

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

One advantage of the separate-column approach over an expression index is that it is not necessary to explicitly specify the text search configuration in queries in order to make use of the index. As shown in the example above, the query can depend on `default_text_search_config`. Another advantage is that searches will be faster, since it will not be necessary to redo the `to_tsvector` calls to verify index matches. (This is more important when using a GiST index than a GIN index; see Section 12.9.) The expression-index approach is simpler to set up, however, and it requires less disk space since the `tsvector` representation is not stored explicitly.

12.3. Controlling Text Search

To implement full text searching there must be a function to create a `tsvector` from a document and a `tsquery` from a user query. Also, we need to return results in a useful order, so we need a function that compares documents with respect to their relevance to the query. It's also important to be able to display the results nicely. PostgreSQL provides support for all of these functions.

12.3.1. Parsing Documents

PostgreSQL provides the function `to_tsvector` for converting a document to the `tsvector` data type.

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

`to_tsvector` parses a textual document into tokens, reduces the tokens to lexemes, and returns a `tsvector` which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a
 fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

In the example above we see that the resulting `tsvector` does not contain the words `a`, `on`, or `it`, the word `rats` became `rat`, and the punctuation sign `-` was ignored.

The `to_tsvector` function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries (Section 12.6) is consulted, where the list can vary depending on the token type. The first dictionary that *recognizes* the token emits one or more normalized *lexemes* to represent the token. For example, `rats` became `rat` because one of the dictionaries recognized that the word `rats` is a plural form of `rat`. Some words are recognized as *stop words* (Section 12.6.1), which causes them to be ignored since they occur too frequently to be useful

in searching. In our example these are a, on, and it. If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign - because there are in fact no dictionaries assigned for its token type (Space symbols), meaning space tokens will never be indexed. The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration (Section 12.7). It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration english for the English language.

The function setweight can be used to label the entries of a tsvector with a given *weight*, where a weight is one of the letters A, B, C, or D. This is typically used to mark entries coming from different parts of a document, such as title versus body. Later, this information can be used for ranking of search results.

Because to_tsvector(NULL) will return NULL, it is recommended to use coalesce whenever a field might be null. Here is the recommended method for creating a tsvector from a structured document:

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

Here we have used setweight to label the source of each lexeme in the finished tsvector, and then merged the labeled tsvector values using the tsvector concatenation operator ||. (Section 12.4.1 gives details about these operations.)

12.3.2. Parsing Queries

PostgreSQL provides the functions to_tsquery, plainto_tsquery, phraseto_tsquery and websearch_to_tsquery for converting a query to the tsquery data type. to_tsquery offers access to more features than either plainto_tsquery or phraseto_tsquery, but it is less forgiving about its input. websearch_to_tsquery is a simplified version of to_tsquery with an alternative syntax, similar to the one used by web search engines.

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsquery creates a tsquery value from *querytext*, which must consist of single tokens separated by the tsquery operators & (AND), | (OR), ! (NOT), and <-> (FOLLOWED BY), possibly grouped using parentheses. In other words, the input to to_tsquery must already follow the general rules for tsquery input, as described in Section 8.11.2. The difference is that while basic tsquery input takes the tokens at face value, to_tsquery normalizes each token into a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

As in basic tsquery input, weight(s) can be attached to each lexeme to restrict it to match only tsvector lexemes of those weight(s). For example:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
```

```
------------------
 'fat' | 'rat':AB
```

Also, * can be attached to a lexeme to specify prefix matching:

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Such a lexeme will match any word in a `tsvector` that begins with the given string.

`to_tsquery` can also accept single-quoted phrases. This is primarily useful when the configuration includes a thesaurus dictionary that may trigger on such phrases. In the example below, a thesaurus contains the rule `supernovae stars : sn`:

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

Without quotes, `to_tsquery` will generate a syntax error for tokens that are not separated by an AND, OR, or FOLLOWED BY operator.

```
plainto_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`plainto_tsquery` transforms the unformatted text *querytext* to a `tsquery` value. The text is parsed and normalized much as for `to_tsvector`, then the & (AND) `tsquery` operator is inserted between surviving words.

Example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Note that `plainto_tsquery` will not recognize `tsquery` operators, weight labels, or prefix-match labels in its input:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

Here, all the input punctuation was discarded.

```
phraseto_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`phraseto_tsquery` behaves much like `plainto_tsquery`, except that it inserts the `<->` (FOLLOWED BY) operator between surviving words instead of the & (AND) operator. Also, stop words are not simply discarded, but are accounted for by inserting `<N>` operators rather than `<->`

operators. This function is useful when searching for exact lexeme sequences, since the FOLLOWED BY operators check lexeme order not just the presence of all the lexemes.

Example:

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

Like `plainto_tsquery`, the `phraseto_tsquery` function will not recognize `tsquery` operators, weight labels, or prefix-match labels in its input:

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

```
websearch_to_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`websearch_to_tsquery` creates a `tsquery` value from *querytext* using an alternative syntax in which simple unformatted text is a valid query. Unlike `plainto_tsquery` and `phraseto_tsquery`, it also recognizes certain operators. Moreover, this function will never raise syntax errors, which makes it possible to use raw user-supplied input for search. The following syntax is supported:

- `unquoted text`: text not inside quote marks will be converted to terms separated by & operators, as if processed by `plainto_tsquery`.
- `"quoted text"`: text inside quote marks will be converted to terms separated by `<->` operators, as if processed by `phraseto_tsquery`.
- `OR`: the word “or” will be converted to the `|` operator.
- `-`: a dash will be converted to the `!` operator.

Other punctuation is ignored. So like `plainto_tsquery` and `phraseto_tsquery`, the `websearch_to_tsquery` function will not recognize `tsquery` operators, weight labels, or prefix-match labels in its input.

Examples:

```
SELECT websearch_to_tsquery('english', 'The fat rats');
 websearch_to_tsquery
----------------------
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
       websearch_to_tsquery
---------------------------------
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
       websearch_to_tsquery
----------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
```

```
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation
 fault"');
        websearch_to_tsquery
--------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""" )( dummy \\ query <-
>');
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

12.3.3. Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and very application-specific. Different applications might require additional information for ranking, e.g., document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

`ts_rank([ *weights* float4[], ] *vector* tsvector, *query* tsquery [, *normalization* integer ]) returns float4`

Ranks vectors based on the frequency of their matching lexemes.

`ts_rank_cd([ *weights* float4[], ] *vector* tsvector, *query* tsquery [, *normalization* integer ]) returns float4`

This function computes the *cover density* ranking for the given document vector and query, as described in Clarke, Cormack, and Tudhope's "Relevance Ranking for One to Three Term Queries" in the journal "Information Processing and Management", 1999. Cover density is similar to `ts_rank` ranking except that the proximity of matching lexemes to each other is taken into consideration.

This function requires lexeme positional information to perform its calculation. Therefore, it ignores any “stripped” lexemes in the `tsvector`. If there are no unstripped lexemes in the input, the result will be zero. (See Section 12.4.1 for more information about the `strip` function and positional information in `tsvector`s.)

For both these functions, the optional *`weights`* argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

```
{D-weight, C-weight, B-weight, A-weight}
```

If no *`weights`* are provided, then these defaults are used:

```
{0.1, 0.2, 0.4, 1.0}
```

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so they can be treated with more or less importance than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size, e.g., a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer *`normalization`* option that specifies whether and how a document's length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using `|` (for example, `2|4`).

- 0 (the default) ignores the document length
- 1 divides the rank by 1 + the logarithm of the document length
- 2 divides the rank by the document length
- 4 divides the rank by the mean harmonic distance between extents (this is implemented only by `ts_rank_cd`)
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by 1 + the logarithm of the number of unique words in document
- 32 divides the rank by itself + 1

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option 32 (`rank/(rank+1)`) can be applied to scale all ranks into the range zero to one, but of course this is just a cosmetic change; it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

This is the same example using normalized ranking:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1)
 */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
```

```
 The Sudbury Neutrino Detector                | 0.705882361190954
 A MACHO View of Galactic Dark Matter         | 0.668123210574724
 Hot Gas and Dark Matter                      |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                  | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter    | 0.650072921219637
 Hot Gas and Dark Matter                      | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos             | 0.615384618911517
 Weak Lensing Distorts the Universe           | 0.450010798361481
```

Ranking can be expensive since it requires consulting the `tsvector` of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.

12.3.4. Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. PostgreSQL provides a function `ts_headline` that implements this functionality.

```
ts_headline([ config regconfig, ] document text, query tsquery
 [, options text ]) returns text
```

`ts_headline` accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. Specifically, the function will use the query to select relevant text fragments, and then highlight all words that appear in the query, even if those word positions do not match the query's restrictions. The configuration to be used to parse the document can be specified by *`config`*; if *`config`* is omitted, the `default_text_search_config` configuration is used.

If an *`options`* string is specified it must consist of a comma-separated list of one or more *`option=value`* pairs. The available options are:

- `MaxWords`, `MinWords` (integers): these numbers determine the longest and shortest headlines to output. The default values are 35 and 15.
- `ShortWord` (integer): words of this length or less will be dropped at the start and end of a headline, unless they are query terms. The default value of three eliminates common English articles.
- `HighlightAll` (boolean): if `true` the whole document will be used as the headline, ignoring the preceding three parameters. The default is `false`.
- `MaxFragments` (integer): maximum number of text fragments to display. The default value of zero selects a non-fragment-based headline generation method. A value greater than zero selects fragment-based headline generation (see below).
- `StartSel`, `StopSel` (strings): the strings with which to delimit query words appearing in the document, to distinguish them from other excerpted words. The default values are "`<b>`" and "`</b>`", which can be suitable for HTML output.
- `FragmentDelimiter` (string): When more than one fragment is displayed, the fragments will be separated by this string. The default is "` ... `".

These option names are recognized case-insensitively. You must double-quote string values if they contain spaces or commas.

In non-fragment-based headline generation, `ts_headline` locates matches for the given *`query`* and chooses a single one to display, preferring matches that have more query words within the allowed headline length. In fragment-based headline generation, `ts_headline` locates the query matches and splits each match into "fragments" of no more than `MaxWords` words each, preferring fragments with more query words, and when possible "stretching" fragments to include surrounding words. The fragment-based mode is thus more useful when the query matches span large sections of the document,

or when it's desirable to display multiple matches. In either mode, if no query matches can be identified, then a single fragment of the first `MinWords` words in the document will be displayed.

For example:

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('english', 'query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms                       +
 and return them in order of their <b>similarity</b> to the+
 <b>query</b>.

SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<,
 StopSel=>>');
                        ts_headline
------------------------------------------------------------
 <<Search>> <<terms>> may occur                            +
 many times ... ranking of the <<search>> matches to decide
```

`ts_headline` uses the original document, not a `tsvector` summary, so it can be slow and should be used with care.

12.4. Additional Features

This section describes additional functions and operators that are useful in connection with text search.

12.4.1. Manipulating Documents

Section 12.3.1 showed how raw textual documents can be converted into `tsvector` values. PostgreSQL also provides functions and operators that can be used to manipulate documents that are already in `tsvector` form.

`tsvector || tsvector`

The `tsvector` concatenation operator returns a vector which combines the lexemes and positional information of the two vectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand vector are offset by the largest position mentioned in the left-hand vector, so that the result is nearly equivalent to the result of performing `to_tsvector` on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.)

One advantage of using concatenation in the vector form, rather than concatenating text before applying `to_tsvector`, is that you can use different configurations to parse different sections of the document. Also, because the `setweight` function marks all lexemes of the given vector

the same way, it is necessary to parse the text and do `setweight` before concatenating if you want to label different parts of the document with different weights.

`setweight(`*`vector`* `tsvector, `*`weight`* `"char") returns tsvector`

`setweight` returns a copy of the input vector in which every position has been labeled with the given *`weight`*, either `A`, `B`, `C`, or `D`. (`D` is the default for new vectors and as such is not displayed on output.) These labels are retained when vectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

Note that weight labels apply to *positions*, not *lexemes*. If the input vector has been stripped of positions then `setweight` does nothing.

`length(`*`vector`* `tsvector) returns integer`

Returns the number of lexemes stored in the vector.

`strip(`*`vector`* `tsvector) returns tsvector`

Returns a vector that lists the same lexemes as the given vector, but lacks any position or weight information. The result is usually much smaller than an unstripped vector, but it is also less useful. Relevance ranking does not work as well on stripped vectors as unstripped ones. Also, the `<->` (FOLLOWED BY) `tsquery` operator will never match stripped input, since it cannot determine the distance between lexeme occurrences.

A full list of `tsvector`-related functions is available in Table 9.43.

12.4.2. Manipulating Queries

Section 12.3.2 showed how raw textual queries can be converted into `tsquery` values. PostgreSQL also provides functions and operators that can be used to manipulate queries that are already in `tsquery` form.

`tsquery && tsquery`

Returns the AND-combination of the two given queries.

`tsquery || tsquery`

Returns the OR-combination of the two given queries.

`!! tsquery`

Returns the negation (NOT) of the given query.

`tsquery <-> tsquery`

Returns a query that searches for a match to the first given query immediately followed by a match to the second given query, using the `<->` (FOLLOWED BY) `tsquery` operator. For example:

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
          ?column?
----------------------------
 'fat' <-> ( 'cat' | 'rat' )
```

`tsquery_phrase(`*`query1`* `tsquery, `*`query2`* `tsquery [, `*`distance`* `integer ]) returns tsquery`

Returns a query that searches for a match to the first given query followed by a match to the second given query at a distance of exactly *`distance`* lexemes, using the `<`*`N`*`>` `tsquery` operator. For example:

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

`numnode(`*`query`* `tsquery) returns integer`

Returns the number of nodes (lexemes plus operators) in a `tsquery`. This function is useful to determine if the *`query`* is meaningful (returns > 0), or contains only stop words (returns 0). Examples:

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain
 lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(`*`query`* `tsquery) returns text`

Returns the portion of a `tsquery` that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

```
SELECT querytree(to_tsquery('defined'));
 querytree
-----------
 'defin'

SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
 T
```

12.4.2.1. Query Rewriting

The `ts_rewrite` family of functions search a given `tsquery` for occurrences of a target subquery, and replace each occurrence with a substitute subquery. In essence this operation is a `tsquery`-specific version of substring replacement. A target and substitute combination can be thought of as a *query rewrite rule*. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (e.g., `new york`, `big apple`, `nyc`, `gotham`) or narrow the search to direct the user to some hot topic. There is some overlap in functionality between this feature and thesaurus dictionaries (Section 12.6.4). However, you can modify a set of rewrite rules on-the-fly without reindexing, whereas updating a thesaurus requires reindexing to be effective.

`ts_rewrite (`*`query`* `tsquery,` *`target`* `tsquery,` *`substitute`* `tsquery) returns tsquery`

This form of `ts_rewrite` simply applies a single rewrite rule: *`target`* is replaced by *`substitute`* wherever it appears in *`query`*. For example:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

ts_rewrite (*query* tsquery, *select* text) returns tsquery

This form of ts_rewrite accepts a starting *query* and an SQL *select* command, which is given as a text string. The *select* must yield two columns of tsquery type. For each row of the *select* result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current *query* value. For example:

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to ORDER BY some ordering key.

Let's consider a real-life astronomical example. We'll expand query supernovae using table-driven rewriting rules:

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'),
 to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM
 aliases');
          ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

We can change the rewriting rules just by updating the table:

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM
 aliases');
                ts_rewrite
-------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible match. To filter out obvious non-candidate rules we can use the containment operators for the tsquery type. In the example below, we select only those rules which might match the original query:

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery
 @> t');
 ts_rewrite
```

```
------------
 'b' & 'c'
```

12.4.3. Triggers for Automatic Updates

> **Note**
>
> The method described in this section has been obsoleted by the use of stored generated columns, as described in Section 12.2.2.

When using a separate column to store the `tsvector` representation of your documents, it is necessary to create a trigger to update the `tsvector` column when the document content columns change. Two built-in trigger functions are available for this, or you can write your own.

```
tsvector_update_trigger(tsvector_column_name,
 config_name, text_column_name [, ... ])
tsvector_update_trigger_column(tsvector_column_name,
 config_column_name, text_column_name [, ... ])
```

These trigger functions automatically compute a `tsvector` column from one or more textual columns, under the control of parameters specified in the `CREATE TRIGGER` command. An example of their use is:

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title &
 body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

Having created this trigger, any change in `title` or `body` will automatically be reflected into `tsv`, without the application having to worry about it.

The first trigger argument must be the name of the `tsvector` column to be updated. The second argument specifies the text search configuration to be used to perform the conversion. For `tsvector_update_trigger`, the configuration name is simply given as the second trigger argument. It must be schema-qualified as shown above, so that the trigger behavior will not change with changes in `search_path`. For `tsvector_update_trigger_column`, the second trigger argument is the name of another table column, which must be of type `regconfig`. This allows a per-row selection

of configuration to be made. The remaining argument(s) are the names of textual columns (of type `text`, `varchar`, or `char`). These will be included in the document in the order given. NULL values will be skipped (but the other columns will still be indexed).

A limitation of these built-in triggers is that they treat all the input columns alike. To process columns differently — for example, to weight title differently from body — it is necessary to write a custom trigger. Here is an example using PL/pgSQL as the trigger language:

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english',
 coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english',
 coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

Keep in mind that it is important to specify the configuration name explicitly when creating `tsvector` values inside triggers, so that the column's contents will not be affected by changes to `default_text_search_config`. Failure to do this is likely to lead to problems such as search results changing after a dump and restore.

12.4.4. Gathering Document Statistics

The function `ts_stat` is useful for checking your configuration and for finding stop-word candidates.

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

`sqlquery` is a text value containing an SQL query which must return a single `tsvector` column. `ts_stat` executes the query and returns statistics about each distinct lexeme (word) contained in the `tsvector` data. The columns returned are

- *`word`* `text` — the value of a lexeme
- *`ndoc`* `integer` — number of documents (`tsvector`s) the word occurred in
- *`nentry`* `integer` — total number of occurrences of the word

If *`weights`* is supplied, only occurrences having one of those weights are counted.

For example, to find the ten most frequent words in a document collection:

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

The same, but counting only word occurrences with weight `A` or `B`:

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
```

```
LIMIT 10;
```

12.5. Parsers

Text search parsers are responsible for splitting raw document text into *tokens* and identifying each token's type, where the set of possible types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope, there is less need for application-specific custom parsers than there is for custom dictionaries. At present PostgreSQL provides just one built-in parser, which has been found to be useful for a wide range of applications.

The built-in parser is named `pg_catalog.default`. It recognizes 23 token types, shown in Table 12.1.

Table 12.1. Default Parser's Token Types

Alias	Description	Example
`asciiword`	Word, all ASCII letters	`elephant`
`word`	Word, all letters	`mañana`
`numword`	Word, letters and digits	`beta1`
`asciihword`	Hyphenated word, all ASCII	`up-to-date`
`hword`	Hyphenated word, all letters	`lógico-matemática`
`numhword`	Hyphenated word, letters and digits	`postgresql-beta1`
`hword_asciipart`	Hyphenated word part, all ASCII	`postgresql` in the context `postgresql-beta1`
`hword_part`	Hyphenated word part, all letters	`lógico` or `matemática` in the context `lógico-matemática`
`hword_numpart`	Hyphenated word part, letters and digits	`beta1` in the context `postgresql-beta1`
`email`	Email address	`foo@example.com`
`protocol`	Protocol head	`http://`
`url`	URL	`example.com/stuff/index.html`
`host`	Host	`example.com`
`url_path`	URL path	`/stuff/index.html`, in the context of a URL
`file`	File or path name	`/usr/local/foo.txt`, if not within a URL
`sfloat`	Scientific notation	`-1.234e56`
`float`	Decimal notation	`-1.234`
`int`	Signed integer	`-1234`
`uint`	Unsigned integer	`1234`
`version`	Version number	`8.3.0`
`tag`	XML tag	`<a href="dictionaries.html">`
`entity`	XML entity	`&amp;`
`blank`	Space symbols	(any whitespace or punctuation not otherwise recognized)

> **Note**
>
> The parser's notion of a “letter” is determined by the database's locale setting, specifically `lc_ctype`. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types `word` and `asciiword` should be treated alike.
>
> `email` does not support all valid email characters as defined by RFC 5322[1]. Specifically, the only non-alphanumeric characters supported for email user names are period, dash, and underscore.

It is possible for the parser to produce overlapping tokens from the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |              description               |
 token
-----------------+----------------------------------------
+---------------
 numhword        | Hyphenated word, letters and digits      | foo-
bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

```
SELECT alias, description, token FROM ts_debug('http://example.com/
stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. Dictionaries

Dictionaries are used to eliminate words that should not be considered in a search (*stop words*), and to *normalize* words so that different derived forms of the same word will match. A successfully normalized word is called a *lexeme*. Aside from improving search quality, normalization and removal of stop words reduce the size of the `tsvector` representation of a document, thereby improving performance. Normalization does not always have linguistic meaning and usually depends on application semantics.

Some examples of normalization:

- Linguistic — Ispell dictionaries try to reduce input words to a normalized form; stemmer dictionaries remove word endings

[1] https://datatracker.ietf.org/doc/html/rfc5322

- URL locations can be canonicalized to make equivalent URLs match:
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Color names can be replaced by their hexadecimal values, e.g., `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- If indexing numbers, we can remove some fractional digits to reduce the range of possible numbers, so for example *3.14*159265359, *3.14*15926, *3.14* will be the same after normalization if only two digits are kept after the decimal point.

A dictionary is a program that accepts a token as input and returns:

- an array of lexemes if the input token is known to the dictionary (notice that one token can produce more than one lexeme)
- a single lexeme with the `TSL_FILTER` flag set, to replace the original token with a new token to be passed to subsequent dictionaries (a dictionary that does this is called a *filtering dictionary*)
- an empty array if the dictionary knows the token, but it is a stop word
- `NULL` if the dictionary does not recognize the input token

PostgreSQL provides predefined dictionaries for many languages. There are also several predefined templates that can be used to create new dictionaries with custom parameters. Each predefined dictionary template is described below. If no existing template is suitable, it is possible to create new ones; see the `contrib/` area of the PostgreSQL distribution for examples.

A text search configuration binds a parser together with a set of dictionaries to process the parser's output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until some dictionary recognizes it as a known word. If it is identified as a stop word, or if no dictionary recognizes the token, it will be discarded and not indexed or searched for. Normally, the first dictionary that returns a non-`NULL` output determines the result, and any remaining dictionaries are not consulted; but a filtering dictionary can replace the given word with a modified word, which is then passed to subsequent dictionaries.

The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a Snowball stemmer or `simple`, which recognizes everything. For example, for an astronomy-specific search (`astro_en` configuration) one could bind token type `asciiword` (ASCII word) to a synonym dictionary of astronomical terms, a general English dictionary and a Snowball English stemmer:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell,
 english_stem;
```

A filtering dictionary can be placed anywhere in the list, except at the end where it'd be useless. Filtering dictionaries are useful to partially normalize words to simplify the task of later dictionaries. For example, a filtering dictionary could be used to remove accents from accented letters, as is done by the unaccent module.

12.6.1. Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. For example, every English text contains words like `a` and `the`, so it is useless to store them in an index. However, stop words do affect the positions in `tsvector`, which in turn affect ranking:

```
SELECT to_tsvector('english', 'in the list of stop words');
```

```
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

The missing positions 1,2,4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop
 words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'),
 to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

It is up to the specific dictionary how it treats stop words. For example, `ispell` dictionaries first normalize words and then look at the list of stop words, while `Snowball` stemmers first check the list of stop words. The reason for the different behavior is an attempt to decrease noise.

12.6.2. Simple Dictionary

The `simple` dictionary template operates by converting the input token to lower case and checking it against a file of stop words. If it is found in the file then an empty array is returned, causing the token to be discarded. If not, the lower-cased form of the word is returned as the normalized lexeme. Alternatively, the dictionary can be configured to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

Here is an example of a dictionary definition using the `simple` template:

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Here, `english` is the base name of a file of stop words. The file's full name will be `$SHAREDIR/tsearch_data/english.stop`, where `$SHAREDIR` means the PostgreSQL installation's shared-data directory, often `/usr/local/share/postgresql` (use `pg_config --sharedir` to determine it if you're not sure). The file format is simply a list of words, one per line. Blank lines and trailing spaces are ignored, and upper case is folded to lower case, but no other processing is done on the file contents.

Now we can test our dictionary:

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

We can also choose to return NULL, instead of the lower-cased word, if it is not found in the stop words file. This behavior is selected by setting the dictionary's Accept parameter to false. Continuing the example:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

With the default setting of Accept = true, it is only useful to place a simple dictionary at the end of a list of dictionaries, since it will never pass on any token to a following dictionary. Conversely, Accept = false is only useful when there is at least one following dictionary.

> **Caution**
>
> Most types of dictionaries rely on configuration files, such as files of stop words. These files *must* be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.

> **Caution**
>
> Normally, a database session will read a dictionary configuration file only once, when it is first used within the session. If you modify a configuration file and want to force existing sessions to pick up the new contents, issue an ALTER TEXT SEARCH DICTIONARY command on the dictionary. This can be a "dummy" update that doesn't actually change any parameter values.

12.6.3. Synonym Dictionary

This dictionary template is used to create dictionaries that replace a word with a synonym. Phrases are not supported (use the thesaurus template (Section 12.6.4) for that). A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word "Paris" to "pari". It is enough to have a Paris paris line in the synonym dictionary and put it before the english_stem dictionary. For example:

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary
  | lexemes
-----------+-----------------+-------+----------------
+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} |
 english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
```

```
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        |
 dictionary | lexemes
-----------+-----------------+-------+---------------------------
+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} |
 my_synonym | {paris}
```

The only parameter required by the `synonym` template is `SYNONYMS`, which is the base name of its configuration file — `my_synonyms` in the above example. The file's full name will be `$SHAREDIR/tsearch_data/my_synonyms.syn` (where `$SHAREDIR` means the PostgreSQL installation's shared-data directory). The file format is just one line per word to be substituted, with the word followed by its synonym, separated by white space. Blank lines and trailing spaces are ignored.

The `synonym` template also has an optional parameter `CaseSensitive`, which defaults to `false`. When `CaseSensitive` is `false`, words in the synonym file are folded to lower case, as are input tokens. When it is `true`, words and tokens are not folded to lower case, but are compared as-is.

An asterisk (`*`) can be placed at the end of a synonym in the configuration file. This indicates that the synonym is a prefix. The asterisk is ignored when the entry is used in `to_tsvector()`, but when it is used in `to_tsquery()`, the result will be a query item with the prefix match marker (see Section 12.3.2). For example, suppose we have these entries in `$SHAREDIR/tsearch_data/synonym_sample.syn`:

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

Then we will get these results:

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym,
 synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR
 asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
```

```
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@
 to_tsquery('tst', 'indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. Thesaurus Dictionary

A thesaurus dictionary (sometimes abbreviated as TZ) is a collection of words that includes information about the relationships of words and phrases, i.e., broader terms (BT), narrower terms (NT), preferred terms, non-preferred terms, related terms, etc.

Basically a thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. PostgreSQL's current implementation of the thesaurus dictionary is an extension of the synonym dictionary with added *phrase* support. A thesaurus dictionary requires a configuration file of the following format:

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

where the colon (`:`) symbol acts as a delimiter between a phrase and its replacement.

A thesaurus dictionary uses a *subdictionary* (which is specified in the dictionary's configuration) to normalize the input text before checking for phrase matches. It is only possible to select one subdictionary. An error is reported if the subdictionary fails to recognize a word. In that case, you should remove the use of the word or teach the subdictionary about it. You can place an asterisk (`*`) at the beginning of an indexed word to skip applying the subdictionary to it, but all sample words *must* be known to the subdictionary.

The thesaurus dictionary chooses the longest match if there are multiple phrases matching the input, and ties are broken by using the last definition.

Specific stop words recognized by the subdictionary cannot be specified; instead use `?` to mark the location where any stop word can appear. For example, assuming that `a` and `the` are stop words according to the subdictionary:

```
? one ? two : swsw
```

matches `a one the two` and `the one a two`; both would be replaced by `swsw`.

Since a thesaurus dictionary has the capability to recognize phrases it must remember its state and interact with the parser. A thesaurus dictionary uses these assignments to check if it should handle the next word or stop accumulation. The thesaurus dictionary must be configured carefully. For example, if the thesaurus dictionary is assigned to handle only the `asciiword` token, then a thesaurus dictionary definition like `one 7` will not work since token type `uint` is not assigned to the thesaurus dictionary.

> **Caution**
>
> Thesauruses are used during indexing so any change in the thesaurus dictionary's parameters *requires* reindexing. For most other dictionary types, small changes such as adding or removing stopwords does not force reindexing.

12.6.4.1. Thesaurus Configuration

To define a new thesaurus dictionary, use the `thesaurus` template. For example:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Here:

- `thesaurus_simple` is the new dictionary's name
- `mythesaurus` is the base name of the thesaurus configuration file. (Its full name will be `$SHAREDIR/tsearch_data/mythesaurus.ths`, where `$SHAREDIR` means the installation shared-data directory.)
- `pg_catalog.english_stem` is the subdictionary (here, a Snowball English stemmer) to use for thesaurus normalization. Notice that the subdictionary will have its own configuration (for example, stop words), which is not shown here.

Now it is possible to bind the thesaurus dictionary `thesaurus_simple` to the desired token types in a configuration, for example:

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. Thesaurus Example

Consider a simple astronomical thesaurus `thesaurus_astro`, which contains some astronomical word combinations:

```
supernovae stars : sn
crab nebulae : crab
```

Below we create a dictionary and bind some token types to an astronomical thesaurus and English stemmer:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

Now we can see how it works. `ts_lexize` is not very useful for testing a thesaurus, because it treats its input as a single token. Instead we can use `plainto_tsquery` and `to_tsvector` which will break their input strings into multiple tokens:

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

In principle, one can use `to_tsquery` if you quote the argument:

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

Notice that `supernova star` matches `supernovae stars` in `thesaurus_astro` because we specified the `english_stem` stemmer in the thesaurus definition. The stemmer removed the `e` and `s`.

To index the original phrase as well as the substitute, just include it in the right-hand part of the definition:

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Ispell Dictionary

The Ispell dictionary template supports *morphological dictionaries*, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictionary can match all declensions and conjugations of the search term `bank`, e.g., `banking`, `banked`, `banks`, `banks'`, and `bank's`.

The standard PostgreSQL distribution does not include any Ispell configuration files. Dictionaries for a large number of languages are available from Ispell[2]. Also, some more modern dictionary file formats are supported — MySpell[3] (OO < 2.0.1) and Hunspell[4] (OO >= 2.0.2). A large list of dictionaries is available on the OpenOffice Wiki[5].

To create an Ispell dictionary perform these steps:

- download dictionary configuration files. OpenOffice extension files have the `.oxt` extension. It is necessary to extract `.aff` and `.dic` files, change extensions to `.affix` and `.dict`. For some dictionary files it is also needed to convert characters to the UTF-8 encoding with commands (for example, for a Norwegian language dictionary):

[2] https://www.cs.hmc.edu/~geoff/ispell.html
[3] https://en.wikipedia.org/wiki/MySpell
[4] https://hunspell.github.io/
[5] https://wiki.openoffice.org/wiki/Dictionaries

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- copy files to the `$SHAREDIR/tsearch_data` directory
- load files into PostgreSQL with the following command:

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

Here, `DictFile`, `AffFile`, and `StopWords` specify the base names of the dictionary, affixes, and stop-words files. The stop-words file has the same format explained above for the `simple` dictionary type. The format of the other files is not specified here but is available from the above-mentioned web sites.

Ispell dictionaries usually recognize a limited set of words, so they should be followed by another broader dictionary; for example, a Snowball dictionary, which recognizes everything.

The `.affix` file of Ispell has the following structure:

```
prefixes
flag *A:
    .           >   RE      # As in enter > reenter
suffixes
flag T:
    E           >   ST      # As in late > latest
    [^AEIOU]Y   >   -Y,IEST # As in dirty > dirtiest
    [AEIOU]Y    >   EST     # As in gray > grayest
    [^EY]       >   EST     # As in small > smallest
```

And the `.dict` file has the following structure:

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

Format of the `.dict` file is:

```
basic_form/affix_class_name
```

In the `.affix` file every affix flag is described in the following format:

```
condition > [-stripping_letters,] adding_affix
```

Here, condition has a format similar to the format of regular expressions. It can use groupings `[...]` and `[^...]`. For example, `[AEIOU]Y` means that the last letter of the word is `"y"` and the penultimate letter is `"a"`, `"e"`, `"i"`, `"o"` or `"u"`. `[^EY]` means that the last letter is neither `"e"` nor `"y"`.

Ispell dictionaries support splitting compound words; a useful feature. Notice that the affix file should specify a special flag using the `compoundwords controlled` statement that marks dictionary words that can participate in compound formation:

```
compoundwords  controlled z
```

Here are some examples for the Norwegian language:

```
SELECT ts_lexize('norwegian_ispell',
 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

MySpell format is a subset of Hunspell. The `.affix` file of Hunspell has the following structure:

```
PFX A Y 1
PFX A   0     re         .
SFX T N 4
SFX T   0     st         e
SFX T   y     iest       [^aeiou]y
SFX T   0     est        [aeiou]y
SFX T   0     est        [^ey]
```

The first line of an affix class is the header. Fields of an affix rules are listed after the header:

- parameter name (PFX or SFX)
- flag (name of the affix class)
- stripping characters from beginning (at prefix) or end (at suffix) of the word
- adding affix
- condition that has a format similar to the format of regular expressions.

The `.dict` file looks like the `.dict` file of Ispell:

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

> **Note**
>
> MySpell does not support compound words. Hunspell has sophisticated support for compound words. At present, PostgreSQL implements only the basic compound word operations of Hunspell.

12.6.6. Snowball Dictionary

The Snowball dictionary template is based on a project by Martin Porter, inventor of the popular Porter's stemming algorithm for the English language. Snowball now provides stemming algorithms for many languages (see the Snowball site[6] for more information). Each algorithm understands how to reduce common variant forms of words to a base, or stem, spelling within its language. A Snowball dictionary requires a `language` parameter to identify which stemmer to use, and optionally can specify a `stopword` file name that gives a list of words to eliminate. (PostgreSQL's standard stopword lists are also provided by the Snowball project.) For example, there is a built-in definition equivalent to

[6] https://snowballstem.org/

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

The stopword file format is the same as already explained.

A Snowball dictionary recognizes everything, whether or not it is able to simplify the word, so it should be placed at the end of the dictionary list. It is useless to have it before any other dictionary because a token will never pass through it to the next dictionary.

12.7. Configuration Example

A text search configuration specifies all options necessary to transform a document into a `tsvector`: the parser to use to break text into tokens, and the dictionaries to use to transform each token into a lexeme. Every call of `to_tsvector` or `to_tsquery` needs a text search configuration to perform its processing. The configuration parameter default_text_search_config specifies the name of the default configuration, which is the one used by text search functions if an explicit configuration parameter is omitted. It can be set in `postgresql.conf`, or set for an individual session using the `SET` command.

Several predefined text search configurations are available, and you can create custom configurations easily. To facilitate management of text search objects, a set of SQL commands is available, and there are several psql commands that display information about text search objects (Section 12.10).

As an example we will create a configuration `pg`, starting by duplicating the built-in `english` configuration:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY =
 pg_catalog.english );
```

We will use a PostgreSQL-specific synonym list and store it in `$SHAREDIR/tsearch_data/pg_dict.syn`. The file contents look like:

```
postgres    pg
pgsql       pg
postgresql  pg
```

We define the synonym dictionary like this:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Next we register the Ispell dictionary `english_ispell`, which has its own configuration files:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Now we can set up the mappings for words in configuration `pg`:

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

We choose not to index or search some token types that the built-in configuration does handle:

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Now we can test our configuration:

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-
relational
database management system, is now undergoing beta testing of the
 next
version of our software.
');
```

The next step is to set the session to use the new configuration, which was created in the `public` schema:

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
---------------------------
 public.pg
```

12.8. Testing and Debugging Text Search

The behavior of a custom text search configuration can easily become confusing. The functions described in this section are useful for testing text search objects. You can test a complete configuration, or test parsers and dictionaries separately.

12.8.1. Configuration Testing

The function `ts_debug` allows easy testing of a text search configuration.

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
```

```
        OUT token text,
        OUT dictionaries regdictionary[],
        OUT dictionary regdictionary,
        OUT lexemes text[])
        returns setof record
```

`ts_debug` displays information about every token of *`document`* as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by *`config`*, or `default_text_search_config` if that argument is omitted.

`ts_debug` returns one row for each token identified in the text by the parser. The columns returned are

- *`alias`* `text` — short name of the token type
- *`description`* `text` — description of the token type
- *`token`* `text` — text of the token
- *`dictionaries`* `regdictionary[]` — the dictionaries selected by the configuration for this token type
- *`dictionary`* `regdictionary` — the dictionary that recognized the token, or `NULL` if none did
- *`lexemes`* `text[]` — the lexeme(s) produced by the dictionary that recognized the token, or `NULL` if none did; an empty array (`{}`) means it was recognized as a stop word

Here is a simple example:

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate
 a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary
  | lexemes
-----------+-----------------+-------+----------------
+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | fat   | {english_stem} |
 english_stem | {fat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | cat   | {english_stem} |
 english_stem | {cat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | sat   | {english_stem} |
 english_stem | {sat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | on    | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | a     | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | mat   | {english_stem} |
 english_stem | {mat}
 blank     | Space symbols   |       | {}             |
  |
```

```
 blank     | Space symbols   | -     | {}             |
  |
 asciiword | Word, all ASCII | it    | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | ate   | {english_stem} |
 english_stem | {ate}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | a     | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | fat   | {english_stem} |
 english_stem | {fat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | rats  | {english_stem} |
 english_stem | {rat}
```

For a more extensive demonstration, we first create a `public.english` configuration and Ispell dictionary for the English language:

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY =
 pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;


SELECT * FROM ts_debug('public.english', 'The Brightest
 supernovaes');
   alias   |   description   |    token    |         dictionaries
        |   dictionary   |   lexemes
-----------+-----------------+-------------
+-------------------------------+----------------+-------------
 asciiword | Word, all ASCII | The         |
 {english_ispell,english_stem} | english_ispell | {}
 blank     | Space symbols   |             | {}
       |                 |
 asciiword | Word, all ASCII | Brightest   |
 {english_ispell,english_stem} | english_ispell | {bright}
 blank     | Space symbols   |             | {}
       |                 |
 asciiword | Word, all ASCII | supernovaes |
 {english_ispell,english_stem} | english_stem   | {supernova}
```

In this example, the word `Brightest` was recognized by the parser as an `ASCII word` (alias `asciiword`). For this token type the dictionary list is `english_ispell` and `english_stem`. The word was recognized by `english_ispell`, which reduced it to the noun `bright`. The word

supernovaes is unknown to the english_ispell dictionary so it was passed to the next dictionary, and, fortunately, was recognized (in fact, english_stem is a Snowball dictionary which recognizes everything; that is why it was placed at the end of the dictionary list).

The word The was recognized by the english_ispell dictionary as a stop word (Section 12.6.1) and will not be indexed. The spaces are discarded too, since the configuration provides no dictionaries at all for them.

You can reduce the width of the output by explicitly specifying which columns you want to see:

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. Parser Testing

The following functions allow direct testing of a text search parser.

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

ts_parse parses the given *document* and returns a series of records, one for each token produced by parsing. Each record includes a tokid showing the assigned token type and a token which is the text of the token. For example:

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof
 record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof
 record
```

ts_token_type returns a table which describes each type of token the specified parser can recognize. For each token type, the table gives the integer tokid that the parser uses to label a token of that type, the alias that names the token type in configuration commands, and a short description. For example:

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------
+-----------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. Dictionary Testing

The `ts_lexize` function facilitates dictionary testing.

```
ts_lexize(dict regdictionary, token text) returns text[]
```

`ts_lexize` returns an array of lexemes if the input *`token`* is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or `NULL` if it is an unknown word.

Examples:

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **Note**
>
> The `ts_lexize` function expects a single *token*, not text. Here is a case where this can be confusing:

```
SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is
 null;
 ?column?
----------
 t
```

The thesaurus dictionary `thesaurus_astro` does know the phrase `supernovae stars`, but `ts_lexize` fails since it does not parse the input text but treats it as a single token. Use `plainto_tsquery` or `to_tsvector` to test thesaurus dictionaries, for example:

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

12.9. Preferred Index Types for Text Search

There are two kinds of indexes that can be used to speed up full text searches: GIN and GiST. Note that indexes are not mandatory for full text searching, but in cases where a column is searched on a regular basis, an index is usually desirable.

To create such an index, do one of:

```
CREATE INDEX name ON table USING GIN (column);
```

Creates a GIN (Generalized Inverted Index)-based index. The *column* must be of `tsvector` type.

```
CREATE INDEX name ON table USING GIST (column [ { DEFAULT | tsvector_ops } (siglen = number) ] );
```

Creates a GiST (Generalized Search Tree)-based index. The *column* can be of `tsvector` or `tsquery` type. Optional integer parameter `siglen` determines signature length in bytes (see below for details).

GIN indexes are the preferred text search index type. As inverted indexes, they contain an index entry for each word (lexeme), with a compressed list of matching locations. Multi-word searches can find the first match, then use the index to remove rows that are lacking additional words. GIN indexes store only the words (lexemes) of `tsvector` values, and not their weight labels. Thus a table row recheck is needed when using a query that involves weights.

A GiST index is *lossy*, meaning that the index might produce false matches, and it is necessary to check the actual table row to eliminate such false matches. (PostgreSQL does this automatically when needed.) GiST indexes are lossy because each document is represented in the index by a fixed-length signature. The signature length in bytes is determined by the value of the optional integer parameter `siglen`. The default signature length (when `siglen` is not specified) is 124 bytes, the maximum signature length is 2024 bytes. The signature is generated by hashing each word into a single bit in an n-bit string, with all these bits OR-ed together to produce an n-bit document signature. When two words hash to the same bit position there will be a false match. If all words in the query have matches (real or false) then the table row must be retrieved to see if the match is correct. Longer signatures lead to a more precise search (scanning a smaller fraction of the index and fewer heap pages), at the cost of a larger index.

A GiST index can be covering, i.e., use the `INCLUDE` clause. Included columns can have data types without any GiST operator class. Included attributes will be stored uncompressed.

Lossiness causes performance degradation due to unnecessary fetches of table records that turn out to be false matches. Since random access to table records is slow, this limits the usefulness of GiST

indexes. The likelihood of false matches depends on several factors, in particular the number of unique words, so using dictionaries to reduce this number is recommended.

Note that GIN index build time can often be improved by increasing maintenance_work_mem, while GiST index build time is not sensitive to that parameter.

Partitioning of big collections and the proper use of GIN and GiST indexes allows the implementation of very fast searches with online update. Partitioning can be done at the database level using table inheritance, or by distributing documents over servers and collecting external search results, e.g., via Foreign Data access. The latter is possible because ranking functions use only local information.

12.10. psql Support

Information about text search configuration objects can be obtained in psql using a set of commands:

```
\dF{d,p,t}[+] [PATTERN]
```

An optional + produces more details.

The optional parameter *PATTERN* can be the name of a text search object, optionally schema-qualified. If *PATTERN* is omitted then information about all visible objects will be displayed. *PATTERN* can be a regular expression and can provide *separate* patterns for the schema and object names. The following examples illustrate this:

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |


=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

The available commands are:

```
\dF[+] [PATTERN]
```

List text search configurations (add + for more detail).

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
```

```
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

```
\dFd[+] [PATTERN]
```

List text search dictionaries (add + for more detail).

```
=> \dFd
                                 List of text search dictionaries
   Schema   |      Name       |
 Description
------------+-----------------
+------------------------------------------------------------
 pg_catalog | arabic_stem     | snowball stemmer for arabic
 language
 pg_catalog | armenian_stem   | snowball stemmer for armenian
 language
 pg_catalog | basque_stem     | snowball stemmer for basque
 language
 pg_catalog | catalan_stem    | snowball stemmer for catalan
 language
 pg_catalog | danish_stem     | snowball stemmer for danish
 language
 pg_catalog | dutch_stem      | snowball stemmer for dutch
 language
 pg_catalog | english_stem    | snowball stemmer for english
 language
 pg_catalog | finnish_stem    | snowball stemmer for finnish
 language
 pg_catalog | french_stem     | snowball stemmer for french
 language
 pg_catalog | german_stem     | snowball stemmer for german
 language
 pg_catalog | greek_stem      | snowball stemmer for greek
 language
 pg_catalog | hindi_stem      | snowball stemmer for hindi
 language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian
 language
 pg_catalog | indonesian_stem | snowball stemmer for indonesian
 language
 pg_catalog | irish_stem      | snowball stemmer for irish
 language
```

```
 pg_catalog | italian_stem    | snowball stemmer for italian
 language
 pg_catalog | lithuanian_stem | snowball stemmer for lithuanian
 language
 pg_catalog | nepali_stem     | snowball stemmer for nepali
 language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian
 language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese
 language
 pg_catalog | romanian_stem   | snowball stemmer for romanian
 language
 pg_catalog | russian_stem    | snowball stemmer for russian
 language
 pg_catalog | serbian_stem    | snowball stemmer for serbian
 language
 pg_catalog | simple          | simple dictionary: just lower
 case and check for stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish
 language
 pg_catalog | swedish_stem    | snowball stemmer for swedish
 language
 pg_catalog | tamil_stem      | snowball stemmer for tamil
 language
 pg_catalog | turkish_stem    | snowball stemmer for turkish
 language
 pg_catalog | yiddish_stem    | snowball stemmer for yiddish
 language
```

`\dFp[+] [PATTERN]`

List text search parsers (add + for more detail).

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
```

```
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

```
\dFt[+] [PATTERN]
```

List text search templates (add + for more detail).

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description
------------+-----------
+-----------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and
 check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by
 its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase
 substitution
```

12.11. Limitations

The current limitations of PostgreSQL's text search features are:

- The length of each lexeme must be less than 2 kilobytes
- The length of a `tsvector` (lexemes + positions) must be less than 1 megabyte
- The number of lexemes must be less than 2^{64}
- Position values in `tsvector` must be greater than 0 and no more than 16,383
- The match distance in a *<N>* (FOLLOWED BY) `tsquery` operator cannot be more than 16,384
- No more than 256 positions per lexeme
- The number of nodes (lexemes + operators) in a `tsquery` must be less than 32,768

For comparison, the PostgreSQL 8.1 documentation contained 10,441 unique words, a total of 335,420 words, and the most frequent word “postgresql” was mentioned 6,127 times in 655 documents.

Another example — the PostgreSQL mailing list archives contained 910,989 unique words with 57,491,343 lexemes in 461,020 messages.

Chapter 13. Concurrency Control

This chapter describes the behavior of the PostgreSQL database system when two or more sessions try to access the same data at the same time. The goals in that situation are to allow efficient access for all sessions while maintaining strict data integrity. Every developer of database applications should be familiar with the topics covered in this chapter.

13.1. Introduction

PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that each SQL statement sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This prevents statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows, providing *transaction isolation* for each database session. MVCC, by eschewing the locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

The main advantage of using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading. PostgreSQL maintains this guarantee even when providing the strictest level of transaction isolation through the use of an innovative *Serializable Snapshot Isolation* (SSI) level.

Table- and row-level locking facilities are also available in PostgreSQL for applications which don't generally need full transaction isolation and prefer to explicitly manage particular points of conflict. However, proper use of MVCC will generally provide better performance than locks. In addition, application-defined advisory locks provide a mechanism for acquiring locks that are not tied to a single transaction.

13.2. Transaction Isolation

The SQL standard defines four levels of transaction isolation. The most strict is Serializable, which is defined by the standard in a paragraph which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in some order. The other three levels are defined in terms of phenomena, resulting from interaction between concurrent transactions, which must not occur at each level. The standard notes that due to the definition of Serializable, none of these phenomena are possible at that level. (This is hardly surprising -- if the effect of the transactions must be consistent with having been run one at a time, how could you see any phenomena caused by interactions?)

The phenomena which are prohibited at various levels are:

dirty read

A transaction reads data written by a concurrent uncommitted transaction.

nonrepeatable read

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

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 due to another recently-committed transaction.

serialization anomaly

The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

The SQL standard and PostgreSQL-implemented transaction isolation levels are described in Table 13.1.

Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

In PostgreSQL, you can request any of the four standard transaction isolation levels, but internally only three distinct isolation levels are implemented, i.e., PostgreSQL's Read Uncommitted mode behaves like Read Committed. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multiversion concurrency control architecture.

The table also shows that PostgreSQL's Repeatable Read implementation does not allow phantom reads. This is acceptable under the SQL standard because the standard specifies which anomalies must *not* occur at certain isolation levels; higher guarantees are acceptable. The behavior of the available isolation levels is detailed in the following subsections.

To set the transaction isolation level of a transaction, use the command SET TRANSACTION.

Important

Some PostgreSQL data types and functions have special rules regarding transactional behavior. In particular, changes made to a sequence (and therefore the counter of a column declared using `serial`) are immediately visible to all other transactions and are not rolled back if the transaction that made the changes aborts. See Section 9.17 and Section 8.1.4.

13.2.1. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction uses this isolation level, a `SELECT` query (without a `FOR UPDATE/SHARE` clause) sees only data committed before the query began; it never sees either uncommitted data or changes committed by concurrent transactions during the query's execution. In effect, a `SELECT` query sees a snapshot of the database as of the instant the query begins to run. However, `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that two successive `SELECT` commands can see different data, even though they are within a single transaction, if other transactions commit changes after the first `SELECT` starts and before the second `SELECT` starts.

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the command start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the would-be updater will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the

originally found row. If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row. The search condition of the command (the WHERE clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation using the updated version of the row. In the case of SELECT FOR UPDATE and SELECT FOR SHARE, this means it is the updated version of the row that is locked and returned to the client.

INSERT with an ON CONFLICT DO UPDATE clause behaves similarly. In Read Committed mode, each row proposed for insertion will either insert or update. Unless there are unrelated errors, one of those two outcomes is guaranteed. If a conflict originates in another transaction whose effects are not yet visible to the INSERT, the UPDATE clause will affect that row, even though possibly *no* version of that row is conventionally visible to the command.

INSERT with an ON CONFLICT DO NOTHING clause may have insertion not proceed for a row due to the outcome of another transaction whose effects are not visible to the INSERT snapshot. Again, this is only the case in Read Committed mode.

MERGE allows the user to specify various combinations of INSERT, UPDATE and DELETE subcommands. A MERGE command with both INSERT and UPDATE subcommands looks similar to INSERT with an ON CONFLICT DO UPDATE clause but does not guarantee that either INSERT or UPDATE will occur. If MERGE attempts an UPDATE or DELETE and the row is concurrently updated but the join condition still passes for the current target and the current source tuple, then MERGE will behave the same as the UPDATE or DELETE commands and perform its action on the updated version of the row. However, because MERGE can specify several actions and they can be conditional, the conditions for each action are re-evaluated on the updated version of the row, starting from the first action, even if the action that had originally matched appears later in the list of actions. On the other hand, if the row is concurrently updated so that the join condition fails, then MERGE will evaluate the command's NOT MATCHED BY SOURCE and NOT MATCHED [BY TARGET] actions next, and execute the first one of each kind that succeeds. If the row is concurrently deleted, then MERGE will evaluate the command's NOT MATCHED [BY TARGET] actions, and execute the first one that succeeds. If MERGE attempts an INSERT and a unique index is present and a duplicate row is concurrently inserted, then a uniqueness violation error is raised; MERGE does not attempt to avoid such errors by restarting evaluation of MATCHED conditions.

Because of the above rules, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database. This behavior makes Read Committed mode unsuitable for commands that involve complex search conditions; however, it is just right for simpler cases. For example, consider updating bank balances with transactions like:

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum =
 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum =
 7534;
COMMIT;
```

If two such transactions concurrently try to change the balance of account 12345, we clearly want the second transaction to start with the updated version of the account's row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

More complex usage can produce undesirable results in Read Committed mode. For example, consider a DELETE command operating on data that is being both added and removed from its restriction criteria by another command, e.g., assume website is a two-row table with website.hits equaling 9 and 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

The `DELETE` will have no effect even though there is a `website.hits = 10` row before and after the `UPDATE`. This occurs because the pre-update row value `9` is skipped, and when the `UPDATE` completes and `DELETE` obtains a lock, the new row value is no longer `10` but `11`, which no longer matches the criteria.

Because Read Committed mode starts each command with a new snapshot that includes all transactions committed up to that instant, subsequent commands in the same transaction will see the effects of the committed concurrent transaction in any case. The point at issue above is whether or not a *single* command sees an absolutely consistent view of the database.

The partial transaction isolation provided by Read Committed mode is adequate for many applications, and this mode is fast and simple to use; however, it is not sufficient for all cases. Applications that do complex queries and updates might require a more rigorously consistent view of the database than Read Committed mode provides.

13.2.2. Repeatable Read Isolation Level

The *Repeatable Read* isolation level only sees data committed before the transaction began; it never sees either uncommitted data or changes committed by concurrent transactions during the transaction's execution. (However, each query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) This is a stronger guarantee than is required by the SQL standard for this isolation level, and prevents all of the phenomena described in Table 13.1 except for serialization anomalies. As mentioned above, this is specifically allowed by the standard, which only describes the *minimum* protections each isolation level must provide.

This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the first non-transaction-control statement in the *transaction*, not as of the start of the current statement within the transaction. Thus, successive `SELECT` commands within a *single* transaction see the same data, i.e., they do not see changes made by other transactions that committed after their own transaction started.

Applications using this level must be prepared to retry transactions due to serialization failures.

`UPDATE`, `DELETE`, `MERGE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the repeatable read transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the repeatable read transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the repeatable read transaction will be rolled back with the message

```
ERROR:  could not serialize access due to concurrent update
```

because a repeatable read transaction cannot modify or lock rows changed by other transactions after the repeatable read transaction began.

When an application receives this error message, it should abort the current transaction and retry the whole transaction from the beginning. The second time through, the transaction will see the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update.

Note that only updating transactions might need to be retried; read-only transactions will never have serialization conflicts.

The Repeatable Read mode provides a rigorous guarantee that each transaction sees a completely stable view of the database. However, this view will not necessarily always be consistent with some serial (one at a time) execution of concurrent transactions of the same level. For example, even a read-only transaction at this level may see a control record updated to show that a batch has been completed but *not* see one of the detail records which is logically part of the batch because it read an earlier revision of the control record. Attempts to enforce business rules by transactions running at this isolation level are not likely to work correctly without careful use of explicit locks to block conflicting transactions.

The Repeatable Read isolation level is implemented using a technique known in academic database literature and in some other database products as *Snapshot Isolation*. Differences in behavior and performance may be observed when compared with systems that use a traditional locking technique that reduces concurrency. Some other systems may even offer Repeatable Read and Snapshot Isolation as distinct isolation levels with different behavior. The permitted phenomena that distinguish the two techniques were not formalized by database researchers until after the SQL standard was developed, and are outside the scope of this manual. For a full treatment, please see [berenson95].

> **Note**
>
> Prior to PostgreSQL version 9.1, a request for the Serializable transaction isolation level provided exactly the same behavior described here. To retain the legacy Serializable behavior, Repeatable Read should now be requested.

13.2.3. Serializable Isolation Level

The *Serializable* isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if transactions had been executed one after another, serially, rather than concurrently. However, like the Repeatable Read level, applications using this level must be prepared to retry transactions due to serialization failures. In fact, this isolation level works exactly the same as Repeatable Read except that it also monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions. This monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring, and detection of the conditions which could cause a *serialization anomaly* will trigger a *serialization failure*.

As an example, consider a table `mytab`, initially containing:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Suppose that serializable transaction A computes:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

and then inserts the result (30) as the `value` in a new row with `class = 2`. Concurrently, serializable transaction B computes:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

and obtains the result 300, which it inserts in a new row with `class = 1`. Then both transactions try to commit. If either transaction were running at the Repeatable Read isolation level, both would be allowed to commit; but since there is no serial order of execution consistent with the result, using Serializable transactions will allow one transaction to commit and will roll the other back with this message:

```
ERROR:  could not serialize access due to read/write dependencies
 among transactions
```

This is because if A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

When relying on Serializable transactions to prevent anomalies, it is important that any data read from a permanent user table not be considered valid until the transaction which read it has successfully committed. This is true even for read-only transactions, except that data read within a *deferrable* read-only transaction is known to be valid as soon as it is read, because such a transaction waits until it can acquire a snapshot guaranteed to be free from such problems before starting to read any data. In all other cases applications must not depend on results read during a transaction that later aborted; instead, they should retry the transaction until it succeeds.

To guarantee true serializability PostgreSQL uses *predicate locking*, which means that it keeps locks which allow it to determine when a write would have had an impact on the result of a previous read from a concurrent transaction, had it run first. In PostgreSQL these locks do not cause any blocking and therefore can *not* play any part in causing a deadlock. They are used to identify and flag dependencies among concurrent Serializable transactions which in certain combinations can lead to serialization anomalies. In contrast, a Read Committed or Repeatable Read transaction which wants to ensure data consistency may need to take out a lock on an entire table, which could block other users attempting to use that table, or it may use `SELECT FOR UPDATE` or `SELECT FOR SHARE` which not only can block other transactions but cause disk access.

Predicate locks in PostgreSQL, like in most other database systems, are based on data actually accessed by a transaction. These will show up in the `pg_locks` system view with a `mode` of `SIReadLock`. The particular locks acquired during execution of a query will depend on the plan used by the query, and multiple finer-grained locks (e.g., tuple locks) may be combined into fewer coarser-grained locks (e.g., page locks) during the course of the transaction to prevent exhaustion of the memory used to track the locks. A `READ ONLY` transaction may be able to release its SIRead locks before completion, if it detects that no conflicts can still occur which could lead to a serialization anomaly. In fact, `READ ONLY` transactions will often be able to establish that fact at startup and avoid taking any predicate locks. If you explicitly request a `SERIALIZABLE READ ONLY DEFERRABLE` transaction, it will block until it can establish this fact. (This is the *only* case where Serializable transactions block but Repeatable Read transactions don't.) On the other hand, SIRead locks often need to be kept past transaction commit, until overlapping read write transactions complete.

Consistent use of Serializable transactions can simplify development. The guarantee that any set of successfully committed concurrent Serializable transactions will have the same effect as if they were run one at a time means that if you can demonstrate that a single transaction, as written, will do the right thing when run by itself, you can have confidence that it will do the right thing in any mix of Serializable transactions, even without any information about what those other transactions might do, or it will not successfully commit. It is important that an environment which uses this technique have a generalized way of handling serialization failures (which always return with an SQLSTATE value of '40001'), because it will be very hard to predict exactly which transactions might contribute to the read/write dependencies and need to be rolled back to prevent serialization anomalies. The monitoring of read/write dependencies has a cost, as does the restart of transactions which are terminated with a serialization failure, but balanced against the cost and blocking involved in use of explicit locks and `SELECT FOR UPDATE` or `SELECT FOR SHARE`, Serializable transactions are the best performance choice for some environments.

While PostgreSQL's Serializable transaction isolation level only allows concurrent transactions to commit if it can prove there is a serial order of execution that would produce the same effect, it doesn't always prevent errors from being raised that would not occur in true serial execution. In particular, it is possible to see unique constraint violations caused by conflicts with overlapping Serializable transactions even after explicitly checking that the key isn't present before attempting to insert it. This can be avoided by making sure that *all* Serializable transactions that insert potentially conflicting keys explicitly check if they can do so first. For example, imagine an application that asks the user for a new key and then checks that it doesn't exist already by trying to select it first, or generates a new key by selecting the maximum existing key and adding one. If some Serializable transactions insert new keys directly without following this protocol, unique constraints violations might be reported even in cases where they could not occur in a serial execution of the concurrent transactions.

For optimal performance when relying on Serializable transactions for concurrency control, these issues should be considered:

- Declare transactions as `READ ONLY` when possible.
- Control the number of active connections, using a connection pool if needed. This is always an important performance consideration, but it can be particularly important in a busy system using Serializable transactions.
- Don't put more into a single transaction than needed for integrity purposes.
- Don't leave connections dangling “idle in transaction” longer than necessary. The configuration parameter idle_in_transaction_session_timeout may be used to automatically disconnect lingering sessions.
- Eliminate explicit locks, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` where no longer needed due to the protections automatically provided by Serializable transactions.
- When the system is forced to combine multiple page-level predicate locks into a single relation-level predicate lock because the predicate lock table is short of memory, an increase in the rate of serialization failures may occur. You can avoid this by increasing max_pred_locks_per_transaction, max_pred_locks_per_relation, and/or max_pred_locks_per_page.
- A sequential scan will always necessitate a relation-level predicate lock. This can result in an increased rate of serialization failures. It may be helpful to encourage the use of index scans by reducing random_page_cost and/or increasing cpu_tuple_cost. Be sure to weigh any decrease in transaction rollbacks and restarts against any overall change in query execution time.

The Serializable isolation level is implemented using a technique known in academic database literature as Serializable Snapshot Isolation, which builds on Snapshot Isolation by adding checks for serialization anomalies. Some differences in behavior and performance may be observed when compared with other systems that use a traditional locking technique. Please see [ports12] for detailed information.

13.3. Explicit Locking

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. (For example, `TRUNCATE` cannot safely be executed concurrently with other operations on the same table, so it obtains an `ACCESS EXCLUSIVE` lock on the table to enforce that.)

To examine a list of the currently outstanding locks in a database server, use the `pg_locks` system view. For more information on monitoring the status of the lock manager subsystem, refer to Chapter 27.

13.3.1. Table-Level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. You can also acquire any of these locks explicitly with the command LOCK. Remember that all of these lock modes are table-level locks, even if the name contains the word “row”; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The only real difference between one lock mode and another is the set of lock modes with which each conflicts (see Table 13.2). Two transactions cannot hold locks of conflicting modes on the same table at the same time. (However, a transaction never conflicts with itself. For example, it might acquire ACCESS EXCLUSIVE lock and later acquire ACCESS SHARE lock on the same table.) Non-conflicting lock modes can be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, an ACCESS EXCLUSIVE lock cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, an ACCESS SHARE lock can be held by multiple transactions).

Table-Level Lock Modes

ACCESS SHARE (AccessShareLock)

Conflicts with the ACCESS EXCLUSIVE lock mode only.

The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only *reads* a table and does not modify it will acquire this lock mode.

ROW SHARE (RowShareLock)

Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

The SELECT command acquires a lock of this mode on all tables on which one of the FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, or FOR KEY SHARE options is specified (in addition to ACCESS SHARE locks on any other tables that are referenced without any explicit FOR ... locking option).

ROW EXCLUSIVE (RowExclusiveLock)

Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

The commands UPDATE, DELETE, INSERT, and MERGE acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that *modifies data* in a table.

SHARE UPDATE EXCLUSIVE (ShareUpdateExclusiveLock)

Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent schema changes and VACUUM runs.

Acquired by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS, COMMENT ON, REINDEX CONCURRENTLY, and certain ALTER INDEX and ALTER TABLE variants (for full details see the documentation of these commands).

SHARE (ShareLock)

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes.

Acquired by CREATE INDEX (without CONCURRENTLY).

`SHARE ROW EXCLUSIVE` (`ShareRowExclusiveLock`)

Conflicts with the `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. This mode protects a table against concurrent data changes, and is self-exclusive so that only one session can hold it at a time.

Acquired by `CREATE TRIGGER` and some forms of `ALTER TABLE`.

`EXCLUSIVE` (`ExclusiveLock`)

Conflicts with the `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. This mode allows only concurrent `ACCESS SHARE` locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

Acquired by `REFRESH MATERIALIZED VIEW CONCURRENTLY`.

`ACCESS EXCLUSIVE` (`AccessExclusiveLock`)

Conflicts with locks of all modes (`ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE`). This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired by the `DROP TABLE`, `TRUNCATE`, `REINDEX`, `CLUSTER`, `VACUUM FULL`, and `REFRESH MATERIALIZED VIEW` (without `CONCURRENTLY`) commands. Many forms of `ALTER INDEX` and `ALTER TABLE` also acquire a lock at this level. This is also the default lock mode for `LOCK TABLE` statements that do not specify a mode explicitly.

> **Tip**
>
> Only an `ACCESS EXCLUSIVE` lock blocks a `SELECT` (without `FOR UPDATE/SHARE`) statement.

Once acquired, a lock is normally held until the end of the transaction. But if a lock is acquired after establishing a savepoint, the lock is released immediately if the savepoint is rolled back to. This is consistent with the principle that `ROLLBACK` cancels all effects of the commands since the savepoint. The same holds for locks acquired within a PL/pgSQL exception block: an error escape from the block releases locks acquired within it.

Table 13.2. Conflicting Lock Modes

Requested Lock Mode	Existing Lock Mode							
	ACCESS SHARE	**ROW SHARE**	**ROW EXCL.**	**SHARE UPDATE EXCL.**	**SHARE**	**SHARE ROW EXCL.**	**EXCL.**	**ACCESS EXCL.**
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL.					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X
SHARE			X	X		X	X	X

Requested Lock Mode	Existing Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
SHARE ROW EXCL.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACCESS EXCL.	X	X	X	X	X	X	X	X

13.3.2. Row-Level Locks

In addition to table-level locks, there are row-level locks, which are listed as below with the contexts in which they are used automatically by PostgreSQL. See Table 13.3 for a complete table of row-level lock conflicts. Note that a transaction can hold conflicting locks on the same row, even in different subtransactions; but other than that, two transactions can never hold conflicting locks on the same row. Row-level locks do not affect data querying; they block only *writers and lockers* to the same row. Row-level locks are released at transaction end or during savepoint rollback, just like table-level locks.

Row-Level Lock Modes

`FOR UPDATE`

`FOR UPDATE` causes the rows retrieved by the `SELECT` statement to be locked as though for update. This prevents them from being locked, modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` or `SELECT FOR KEY SHARE` of these rows will be blocked until the current transaction ends; conversely, `SELECT FOR UPDATE` will wait for a concurrent transaction that has run any of those commands on the same row, and will then lock and return the updated row (or no row, if the row was deleted). Within a `REPEATABLE READ` or `SERIALIZABLE` transaction, however, an error will be thrown if a row to be locked has changed since the transaction started. For further discussion see Section 13.4.

The `FOR UPDATE` lock mode is also acquired by any `DELETE` on a row, and also by an `UPDATE` that modifies the values of certain columns. Currently, the set of columns considered for the `UPDATE` case are those that have a unique index on them that can be used in a foreign key (so partial indexes and expressional indexes are not considered), but this may change in the future.

`FOR NO KEY UPDATE`

Behaves similarly to `FOR UPDATE`, except that the lock acquired is weaker: this lock will not block `SELECT FOR KEY SHARE` commands that attempt to acquire a lock on the same rows. This lock mode is also acquired by any `UPDATE` that does not acquire a `FOR UPDATE` lock.

`FOR SHARE`

Behaves similarly to `FOR NO KEY UPDATE`, except that it acquires a shared lock rather than exclusive lock on each retrieved row. A shared lock blocks other transactions from performing `UPDATE`, `DELETE`, `SELECT FOR UPDATE` or `SELECT FOR NO KEY UPDATE` on these rows, but it does not prevent them from performing `SELECT FOR SHARE` or `SELECT FOR KEY SHARE`.

`FOR KEY SHARE`

Behaves similarly to `FOR SHARE`, except that the lock is weaker: `SELECT FOR UPDATE` is blocked, but not `SELECT FOR NO KEY UPDATE`. A key-shared lock blocks other transactions

from performing DELETE or any UPDATE that changes the key values, but not other UPDATE, and neither does it prevent SELECT FOR NO KEY UPDATE, SELECT FOR SHARE, or SELECT FOR KEY SHARE.

PostgreSQL doesn't remember any information about modified rows in memory, so there is no limit on the number of rows locked at one time. However, locking a row might cause a disk write, e.g., SELECT FOR UPDATE modifies selected rows to mark them locked, and so will result in disk writes.

Table 13.3. Conflicting Row-Level Locks

Requested Lock Mode	Current Lock Mode			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. Page-Level Locks

In addition to table and row locks, page-level share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a row is fetched or updated. Application developers normally need not be concerned with page-level locks, but they are mentioned here for completeness.

13.3.4. Deadlocks

The use of explicit locking can increase the likelihood of *deadlocks*, wherein two (or more) transactions each hold locks that the other wants. For example, if transaction 1 acquires an exclusive lock on table A and then tries to acquire an exclusive lock on table B, while transaction 2 has already exclusive-locked table B and now wants an exclusive lock on table A, then neither one can proceed. PostgreSQL automatically detects deadlock situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete. (Exactly which transaction will be aborted is difficult to predict and should not be relied upon.)

Note that deadlocks can also occur as the result of row-level locks (and thus, they can occur even if explicit locking is not used). Consider the case in which two concurrent transactions modify a table. The first transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum =
 11111;
```

This acquires a row-level lock on the row with the specified account number. Then, the second transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum =
 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum =
 11111;
```

The first UPDATE statement successfully acquires a row-level lock on the specified row, so it succeeds in updating that row. However, the second UPDATE statement finds that the row it is attempting to update has already been locked, so it waits for the transaction that acquired the lock to complete.

Transaction two is now waiting on transaction one to complete before it continues execution. Now, transaction one executes:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum =
 22222;
```

Transaction one attempts to acquire a row-level lock on the specified row, but it cannot: transaction two already holds such a lock. So it waits for transaction two to complete. Thus, transaction one is blocked on transaction two, and transaction two is blocked on transaction one: a deadlock condition. PostgreSQL will detect this situation and abort one of the transactions.

The best defense against deadlocks is generally to avoid them by being certain that all applications using a database acquire locks on multiple objects in a consistent order. In the example above, if both transactions had updated the rows in the same order, no deadlock would have occurred. One should also ensure that the first lock acquired on an object in a transaction is the most restrictive mode that will be needed for that object. If it is not feasible to verify this in advance, then deadlocks can be handled on-the-fly by retrying transactions that abort due to deadlocks.

So long as no deadlock situation is detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).

13.3.5. Advisory Locks

PostgreSQL provides a means for creating locks that have application-defined meanings. These are called *advisory locks*, because the system does not enforce their use — it is up to the application to use them correctly. Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model. For example, a common use of advisory locks is to emulate pessimistic locking strategies typical of so-called “flat file” data management systems. While a flag stored in a table could be used for the same purpose, advisory locks are faster, avoid table bloat, and are automatically cleaned up by the server at the end of the session.

There are two ways to acquire an advisory lock in PostgreSQL: at session level or at transaction level. Once acquired at session level, an advisory lock is held until explicitly released or the session ends. Unlike standard lock requests, session-level advisory lock requests do not honor transaction semantics: a lock acquired during a transaction that is later rolled back will still be held following the rollback, and likewise an unlock is effective even if the calling transaction fails later. A lock can be acquired multiple times by its owning process; for each completed lock request there must be a corresponding unlock request before the lock is actually released. Transaction-level lock requests, on the other hand, behave more like regular lock requests: they are automatically released at the end of the transaction, and there is no explicit unlock operation. This behavior is often more convenient than the session-level behavior for short-term usage of an advisory lock. Session-level and transaction-level lock requests for the same advisory lock identifier will block each other in the expected way. If a session already holds a given advisory lock, additional requests by it will always succeed, even if other sessions are awaiting the lock; this statement is true regardless of whether the existing lock hold and new request are at session level or transaction level.

Like all locks in PostgreSQL, a complete list of advisory locks currently held by any session can be found in the `pg_locks` system view.

Both advisory locks and regular locks are stored in a shared memory pool whose size is defined by the configuration variables max_locks_per_transaction and max_connections. Care must be taken not to exhaust this memory or the server will be unable to grant any locks at all. This imposes an upper limit on the number of advisory locks grantable by the server, typically in the tens to hundreds of thousands depending on how the server is configured.

In certain cases using advisory locking methods, especially in queries involving explicit ordering and `LIMIT` clauses, care must be taken to control the locks acquired because of the order in which SQL expressions are evaluated. For example:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; --
 danger!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

In the above queries, the second form is dangerous because the `LIMIT` is not guaranteed to be applied before the locking function is executed. This might cause some locks to be acquired that the application was not expecting, and hence would fail to release (until it ends the session). From the point of view of the application, such locks would be dangling, although still viewable in `pg_locks`.

The functions provided to manipulate advisory locks are described in Section 9.28.10.

13.4. Data Consistency Checks at the Application Level

It is very difficult to enforce business rules regarding data integrity using Read Committed transactions because the view of the data is shifting with each statement, and even a single statement may not restrict itself to the statement's snapshot if a write conflict occurs.

While a Repeatable Read transaction has a stable view of the data throughout its execution, there is a subtle issue with using MVCC snapshots for data consistency checks, involving something known as *read/write conflicts*. If one transaction writes data and a concurrent transaction attempts to read the same data (whether before or after the write), it cannot see the work of the other transaction. The reader then appears to have executed first regardless of which started first or which committed first. If that is as far as it goes, there is no problem, but if the reader also writes data which is read by a concurrent transaction there is now a transaction which appears to have run before either of the previously mentioned transactions. If the transaction which appears to have executed last actually commits first, it is very easy for a cycle to appear in a graph of the order of execution of the transactions. When such a cycle appears, integrity checks will not work correctly without some help.

As mentioned in Section 13.2.3, Serializable transactions are just Repeatable Read transactions which add nonblocking monitoring for dangerous patterns of read/write conflicts. When a pattern is detected which could cause a cycle in the apparent order of execution, one of the transactions involved is rolled back to break the cycle.

13.4.1. Enforcing Consistency with Serializable Transactions

If the Serializable transaction isolation level is used for all writes and for all reads which need a consistent view of the data, no other effort is required to ensure consistency. Software from other environments which is written to use serializable transactions to ensure consistency should “just work” in this regard in PostgreSQL.

When using this technique, it will avoid creating an unnecessary burden for application programmers if the application software goes through a framework which automatically retries transactions which are rolled back with a serialization failure. It may be a good idea to set `default_transaction_isolation` to `serializable`. It would also be wise to take some action to ensure that no other transaction isolation level is used, either inadvertently or to subvert integrity checks, through checks of the transaction isolation level in triggers.

See Section 13.2.3 for performance suggestions.

> **Warning: Serializable Transactions and Data Replication**
>
> This level of integrity protection using Serializable transactions does not yet extend to hot standby mode (Section 26.4) or logical replicas. Because of that, those using hot standby or logical replication may want to use Repeatable Read and explicit locking on the primary.

13.4.2. Enforcing Consistency with Explicit Blocking Locks

When non-serializable writes are possible, to ensure the current validity of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE`, `SELECT FOR SHARE`, or an appropriate `LOCK TABLE` statement. (`SELECT FOR UPDATE` and `SELECT FOR SHARE` lock just the returned rows against concurrent updates, while `LOCK TABLE` locks the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Also of note to those converting from other environments is the fact that `SELECT FOR UPDATE` does not ensure that a concurrent transaction will not update or delete a selected row. To do that in PostgreSQL you must actually update the row, even if no values need to be changed. `SELECT FOR UPDATE` *temporarily blocks* other transactions from acquiring the same lock or executing an `UPDATE` or `DELETE` which would affect the locked row, but once the transaction holding this lock commits or rolls back, a blocked transaction will proceed with the conflicting operation unless an actual `UPDATE` of the row was performed while the lock was held.

Global validity checks require extra thought under non-serializable MVCC. For example, a banking application might wish to check that the sum of all credits in one table equals the sum of debits in another table, when both tables are being actively updated. Comparing the results of two successive `SELECT sum(...)` commands will not work reliably in Read Committed mode, since the second query will likely include the results of transactions not counted by the first. Doing the two sums in a single repeatable read transaction will give an accurate picture of only the effects of transactions that committed before the repeatable read transaction started — but one might legitimately wonder whether the answer is still relevant by the time it is delivered. If the repeatable read transaction itself applied some changes before trying to make the consistency check, the usefulness of the check becomes even more debatable, since now it includes some but not all post-transaction-start changes. In such cases a careful person might wish to lock all tables needed for the check, in order to get an indisputable picture of current reality. A `SHARE` mode (or higher) lock guarantees that there are no uncommitted changes in the locked table, other than those of the current transaction.

Note also that if one is relying on explicit locking to prevent concurrent changes, one should either use Read Committed mode, or in Repeatable Read mode be careful to obtain locks before performing queries. A lock obtained by a repeatable read transaction guarantees that no other transactions modifying the table are still running, but if the snapshot seen by the transaction predates obtaining the lock, it might predate some now-committed changes in the table. A repeatable read transaction's snapshot is actually frozen at the start of its first query or data-modification command (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `MERGE`), so it is possible to obtain locks explicitly before the snapshot is frozen.

13.5. Serialization Failure Handling

Both Repeatable Read and Serializable isolation levels can produce errors that are designed to prevent serialization anomalies. As previously stated, applications using these levels must be prepared to retry transactions that fail due to serialization errors. Such an error's message text will vary according to the precise circumstances, but it will always have the SQLSTATE code `40001` (`serialization_failure`).

It may also be advisable to retry deadlock failures. These have the SQLSTATE code `40P01` (`deadlock_detected`).

In some cases it is also appropriate to retry unique-key failures, which have SQLSTATE code `23505` (`unique_violation`), and exclusion constraint failures, which have SQLSTATE code `23P01` (`exclusion_violation`). For example, if the application selects a new value for a primary key column after inspecting the currently stored keys, it could get a unique-key failure because another application instance selected the same new key concurrently. This is effectively a serialization failure, but the server will not detect it as such because it cannot "see" the connection between the inserted value and the previous reads. There are also some corner cases in which the server will issue a unique-key or exclusion constraint error even though in principle it has enough information to determine that a serialization problem is the underlying cause. While it's recommendable to just retry `serialization_failure` errors unconditionally, more care is needed when retrying these other error codes, since they might represent persistent error conditions rather than transient failures.

It is important to retry the complete transaction, including all logic that decides which SQL to issue and/or which values to use. Therefore, PostgreSQL does not offer an automatic retry facility, since it cannot do so with any guarantee of correctness.

Transaction retry does not guarantee that the retried transaction will complete; multiple retries may be needed. In cases with very high contention, it is possible that completion of a transaction may take many attempts. In cases involving a conflicting prepared transaction, it may not be possible to make progress until the prepared transaction commits or rolls back.

13.6. Caveats

Some DDL commands, currently only `TRUNCATE` and the table-rewriting forms of `ALTER TABLE`, are not MVCC-safe. This means that after the truncation or rewrite commits, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the DDL command committed. This will only be an issue for a transaction that did not access the table in question before the DDL command started — any transaction that has done so would hold at least an `ACCESS SHARE` table lock, which would block the DDL command until that transaction completes. So these commands will not cause any apparent inconsistency in the table contents for successive queries on the target table, but they could cause visible inconsistency between the contents of the target table and other tables in the database.

Support for the Serializable transaction isolation level has not yet been added to hot standby replication targets (described in Section 26.4). The strictest isolation level currently supported in hot standby mode is Repeatable Read. While performing all permanent database writes within Serializable transactions on the primary will ensure that all standbys will eventually reach a consistent state, a Repeatable Read transaction run on the standby can sometimes see a transient state that is inconsistent with any serial execution of the transactions on the primary.

Internal access to the system catalogs is not done using the isolation level of the current transaction. This means that newly created database objects such as tables are visible to concurrent Repeatable Read and Serializable transactions, even though the rows they contain are not. In contrast, queries that explicitly examine the system catalogs don't see rows representing concurrently created database objects, in the higher isolation levels.

13.7. Locking and Indexes

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL. The various index types are handled as follows:

B-tree, GiST and SP-GiST indexes

> Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. These index types provide the highest concurrency without deadlock conditions.

Hash indexes

Share/exclusive hash-bucket-level locks are used for read/write access. Locks are released after the whole bucket is processed. Bucket-level locks provide better concurrency than index-level ones, but deadlock is possible since the locks are held longer than one index operation.

GIN indexes

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. But note that insertion of a GIN-indexed value usually produces several index key insertions per row, so GIN might do substantial work for a single value's insertion.

Currently, B-tree indexes offer the best performance for concurrent applications; since they also have more features than hash indexes, they are the recommended index type for concurrent applications that need to index scalar data. When dealing with non-scalar data, B-trees are not useful, and GiST, SP-GiST or GIN indexes should be used instead.

Chapter 14. Performance Tips

Query performance can be affected by many things. Some of these can be controlled by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

14.1. Using EXPLAIN

PostgreSQL devises a *query plan* for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex *planner* that tries to choose good plans. You can use the EXPLAIN command to see what query plan the planner creates for any query. Plan-reading is an art that requires some experience to master, but this section attempts to cover the basics.

Examples in this section are drawn from the regression test database after doing a VACUUM ANALYZE, using v17 development sources. You should be able to get similar results if you try the examples yourself, but your estimated costs and row counts might vary slightly because ANALYZE's statistics are random samples rather than exact, and because costs are inherently somewhat platform-dependent.

The examples use EXPLAIN's default "text" output format, which is compact and convenient for humans to read. If you want to feed EXPLAIN's output to a program for further analysis, you should use one of its machine-readable output formats (XML, JSON, or YAML) instead.

14.1.1. EXPLAIN Basics

The structure of a query plan is a tree of *plan nodes*. Nodes at the bottom level of the tree are scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. There are also non-table row sources, such as VALUES clauses and set-returning functions in FROM, which have their own scan node types. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes above the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. Additional lines might appear, indented from the node's summary line, to show additional properties of the node. The very first line (the summary line for the topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

Since this query has no WHERE clause, it must scan all the rows of the table, so the planner has chosen to use a simple sequential scan plan. The numbers that are quoted in parentheses are (left to right):

- Estimated start-up cost. This is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node.
- Estimated total cost. This is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved. In practice a node's parent node might stop short of reading all available rows (see the LIMIT example below).
- Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.

- Estimated average width of rows output by this plan node (in bytes).

The costs are measured in arbitrary units determined by the planner's cost parameters (see Section 19.7.2). Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to `1.0` and the other cost parameters are set relative to that. The examples in this section are run with the default cost parameters.

It's important to understand that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent to convert output values to text form or to transmit them to the client, which could be important factors in the real elapsed time; but the planner ignores those costs because it cannot change them by altering the plan. (Every correct plan will output the same row set, we trust.)

The `rows` value is a little tricky because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node. This is often less than the number scanned, as a result of filtering by any `WHERE`-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Returning to our example:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

These numbers are derived very straightforwardly. If you do:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

you will find that `tenk1` has 345 disk pages and 10000 rows. The estimated cost is computed as (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost). By default, `seq_page_cost` is 1.0 and `cpu_tuple_cost` is 0.01, so the estimated cost is (345 * 1.0) + (10000 * 0.01) = 445.

Now let's modify the query to add a `WHERE` condition:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=7000 width=244)
   Filter: (unique1 < 7000)
```

Notice that the `EXPLAIN` output shows the `WHERE` clause being applied as a “filter” condition attached to the Seq Scan plan node. This means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has been reduced because of the `WHERE` clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit (by 10000 * cpu_operator_cost, to be exact) to reflect the extra CPU time spent checking the `WHERE` condition.

The actual number of rows this query would select is 7000, but the `rows` estimate is only approximate. If you try to duplicate this experiment, you may well get a slightly different estimate; moreover, it can change after each `ANALYZE` command, because the statistics produced by `ANALYZE` are taken from a randomized sample of the table.

Now, let's make the condition more restrictive:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                     QUERY PLAN
-------------------------------------------------------------------------
-----------
 Bitmap Heap Scan on tenk1  (cost=5.06..224.98 rows=100 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=100 width=0)
         Index Cond: (unique1 < 100)
```

Here the planner has decided to use a two-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two plan levels is that the upper plan node sorts the row locations identified by the index into physical order before reading them, to minimize the cost of separate fetches. The "bitmap" mentioned in the node names is the mechanism that does the sorting.)

Now let's add another condition to the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 =
 'xxx';

                                     QUERY PLAN
-------------------------------------------------------------------------
-----------
 Bitmap Heap Scan on tenk1  (cost=5.04..225.20 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=100 width=0)
         Index Cond: (unique1 < 100)
```

The added condition `stringu1 = 'xxx'` reduces the output row count estimate, but not the cost because we still have to visit the same set of rows. That's because the `stringu1` clause cannot be applied as an index condition, since this index is only on the `unique1` column. Instead it is applied as a filter on the rows retrieved using the index. Thus the cost has actually gone up slightly to reflect this extra checking.

In some cases the planner will prefer a "simple" index scan plan:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                     QUERY PLAN
-------------------------------------------------------------------------
----------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1
 width=244)
   Index Cond: (unique1 = 42)
```

In this type of plan the table rows are fetched in index order, which makes them even more expensive to read, but there are so few that the extra cost of sorting the row locations is not worth it. You'll most often see this plan type for queries that fetch just a single row. It's also often used for queries that have an `ORDER BY` condition that matches the index order, because then no extra sorting step is needed to satisfy the `ORDER BY`. In this example, adding `ORDER BY unique1` would use the same plan because the index already implicitly provides the requested ordering.

The planner may implement an `ORDER BY` clause in several ways. The above example shows that such an ordering clause may be implemented implicitly. The planner may also add an explicit `Sort` step:

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;

                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

If a part of the plan guarantees an ordering on a prefix of the required sort keys, then the planner may instead decide to use an `Incremental Sort` step:

```
EXPLAIN SELECT * FROM tenk1 ORDER BY hundred, ten LIMIT 100;

                                          QUERY PLAN
-------------------------------------------------------------------
----------------------------
 Limit  (cost=19.35..39.49 rows=100 width=244)
   ->  Incremental Sort  (cost=19.35..2033.39 rows=10000 width=244)
         Sort Key: hundred, ten
         Presorted Key: hundred
         ->  Index Scan using tenk1_hundred on tenk1
 (cost=0.29..1574.20 rows=10000 width=244)
```

Compared to regular sorts, sorting incrementally allows returning tuples before the entire result set has been sorted, which particularly enables optimizations with `LIMIT` queries. It may also reduce memory usage and the likelihood of spilling sorts to disk, but it comes at the cost of the increased overhead of splitting the result set into multiple sorting batches.

If there are separate indexes on several of the columns referenced in `WHERE`, the planner might choose to use an AND or OR combination of the indexes:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------
-----------------
 Bitmap Heap Scan on tenk1  (cost=25.07..60.11 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.07..25.07 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=100 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78
 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

But this requires visiting both indexes, so it's not necessarily a win compared to using just one index and treating the other condition as a filter. If you vary the ranges involved you'll see the plan change accordingly.

Here is an example showing the effects of `LIMIT`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000
 LIMIT 2;
```

```
                                      QUERY PLAN
------------------------------------------------------------------------
------------------
 Limit  (cost=0.29..14.28 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..70.27
 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

This is the same query as above, but we added a `LIMIT` so that not all the rows need be retrieved, and the planner changed its mind about what to do. Notice that the total cost and row count of the Index Scan node are shown as if it were run to completion. However, the Limit node is expected to stop after retrieving only a fifth of those rows, so its total cost is only a fifth as much, and that's the actual estimated cost of the query. This plan is preferred over adding a Limit node to the previous plan because the Limit could not avoid paying the startup cost of the bitmap scan, so the total cost would be something over 25 units with that approach.

Let's try joining two tables, using the columns we have been discussing:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                       QUERY PLAN
------------------------------------------------------------------------
--------------------
 Nested Loop  (cost=4.65..118.50 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10
 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.90
 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

In this plan, we have a nested-loop join node with two table scans as inputs, or children. The indentation of the node summary lines reflects the plan tree structure. The join's first, or “outer”, child is a bitmap scan similar to those we saw before. Its cost and row count are the same as we'd get from `SELECT ... WHERE unique1 < 10` because we are applying the `WHERE` clause `unique1 < 10` at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn't affect the row count of the outer scan. The nested-loop join node will run its second, or “inner” child once for each row obtained from the outer child. Column values from the current outer row can be plugged into the inner scan; here, the `t1.unique2` value from the outer row is available, so we get a plan and costs similar to what we saw above for a simple `SELECT ... WHERE t2.unique2 = constant` case. (The estimated cost is actually a bit lower than what was seen above, as a result of caching that's expected to occur during the repeated index scans on `t2`.) The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer row (10 * 7.90, here), plus a little CPU time for join processing.

In this example the join's output row count is the same as the product of the two scans' row counts, but that's not true in all cases because there can be additional `WHERE` clauses that mention both tables and so can only be applied at the join point, not to either input scan. Here's an example:

```
EXPLAIN SELECT *
```

```
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred <
 t2.hundred;

                                        QUERY PLAN
-------------------------------------------------------------------
-------------------------
 Nested Loop  (cost=4.65..49.36 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10
 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2
 (cost=0.29..8.46 rows=10 width=244)
               Index Cond: (unique2 < 10)
```

The condition `t1.hundred < t2.hundred` can't be tested in the `tenk2_unique2` index, so it's applied at the join node. This reduces the estimated output row count of the join node, but does not change either input scan.

Notice that here the planner has chosen to “materialize” the inner relation of the join, by putting a Materialize plan node atop it. This means that the `t2` index scan will be done just once, even though the nested-loop join node needs to read that data ten times, once for each row from the outer relation. The Materialize node saves the data in memory as it's read, and then returns the data from memory on each subsequent pass.

When dealing with outer joins, you might see join plan nodes with both “Join Filter” and plain “Filter” conditions attached. Join Filter conditions come from the outer join's `ON` clause, so a row that fails the Join Filter condition could still get emitted as a null-extended row. But a plain Filter condition is applied after the outer-join rules and so acts to remove rows unconditionally. In an inner join there is no semantic difference between these types of filters.

If we change the query's selectivity a bit, we might get a very different join plan:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-------------------------------------------------------------------
----------------------
 Hash Join  (cost=226.23..709.73 rows=100 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000
 width=244)
   ->  Hash  (cost=224.98..224.98 rows=100 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98
 rows=100 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1
 (cost=0.00..5.04 rows=100 width=0)
                     Index Cond: (unique1 < 100)
```

Here, the planner has chosen to use a hash join, in which rows of one table are entered into an in-memory hash table, after which the other table is scanned and the hash table is probed for matches to

each row. Again note how the indentation reflects the plan structure: the bitmap scan on `tenk1` is the input to the Hash node, which constructs the hash table. That's then returned to the Hash Join node, which reads rows from its outer child plan and searches the hash table for each one.

Another possible type of join is a merge join, illustrated here:

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------
----------------------
 Merge Join  (cost=0.56..233.49 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1
 (cost=0.29..643.28 rows=100 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..166.28
 rows=1000 width=244)
```

Merge join requires its input data to be sorted on the join keys. In this example each input is sorted by using an index scan to visit the rows in the correct order; but a sequential scan and sort could also be used. (Sequential-scan-and-sort frequently beats an index scan for sorting many rows, because of the nonsequential disk access required by the index scan.)

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the cheapest, using the enable/disable flags described in Section 19.7.1. (This is a crude tool, but useful. See also Section 14.3.) For example, if we're unconvinced that merge join is the best join type for the previous example, we could try

```
SET enable_mergejoin = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------
----------------------
 Hash Join  (cost=226.23..344.08 rows=10 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on onek t2  (cost=0.00..114.00 rows=1000 width=244)
   ->  Hash  (cost=224.98..224.98 rows=100 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98
 rows=100 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1
 (cost=0.00..5.04 rows=100 width=0)
                     Index Cond: (unique1 < 100)
```

which shows that the planner thinks that hash join would be nearly 50% more expensive than merge join for this case. Of course, the next question is whether it's right about that. We can investigate that using `EXPLAIN ANALYZE`, as discussed below.

Some query plans involve *subplans*, which arise from sub-`SELECT`s in the original query. Such queries can sometimes be transformed into ordinary join plans, but when they cannot be, we get plans like:

```
EXPLAIN VERBOSE SELECT unique1
FROM tenk1 t
WHERE t.ten < ALL (SELECT o.ten FROM onek o WHERE o.four = t.four);

                                QUERY PLAN
------------------------------------------------------------------------
------
 Seq Scan on public.tenk1 t  (cost=0.00..586095.00 rows=5000
 width=4)
   Output: t.unique1
   Filter: (ALL (t.ten < (SubPlan 1).col1))
   SubPlan 1
     ->  Seq Scan on public.onek o  (cost=0.00..116.50 rows=250
 width=4)
           Output: o.ten
           Filter: (o.four = t.four)
```

This rather artificial example serves to illustrate a couple of points: values from the outer plan level can be passed down into a subplan (here, `t.four` is passed down) and the results of the sub-select are available to the outer plan. Those result values are shown by `EXPLAIN` with notations like (*sub-plan_name*).col*N*, which refers to the *N*'th output column of the sub-`SELECT`.

In the example above, the `ALL` operator runs the subplan again for each row of the outer query (which accounts for the high estimated cost). Some queries can use a *hashed subplan* to avoid that:

```
EXPLAIN SELECT *
FROM tenk1 t
WHERE t.unique1 NOT IN (SELECT o.unique1 FROM onek o);

                                        QUERY PLAN
------------------------------------------------------------------------
-------------------------
 Seq Scan on tenk1 t  (cost=61.77..531.77 rows=5000 width=244)
   Filter: (NOT (ANY (unique1 = (hashed SubPlan 1).col1)))
   SubPlan 1
     ->  Index Only Scan using onek_unique1 on onek o
 (cost=0.28..59.27 rows=1000 width=4)
(4 rows)
```

Here, the subplan is run a single time and its output is loaded into an in-memory hash table, which is then probed by the outer `ANY` operator. This requires that the sub-`SELECT` not reference any variables of the outer query, and that the `ANY`'s comparison operator be amenable to hashing.

If, in addition to not referencing any variables of the outer query, the sub-`SELECT` cannot return more than one row, it may instead be implemented as an *initplan*:

```
EXPLAIN VERBOSE SELECT unique1
FROM tenk1 t1 WHERE t1.ten = (SELECT (random() * 10)::integer);

                               QUERY PLAN
------------------------------------------------------------
--------
 Seq Scan on public.tenk1 t1  (cost=0.02..470.02 rows=1000 width=4)
   Output: t1.unique1
   Filter: (t1.ten = (InitPlan 1).col1)
   InitPlan 1
```

```
     ->  Result  (cost=0.00..0.02 rows=1 width=4)
           Output: ((random() * '10'::double precision))::integer
```

An initplan is run only once per execution of the outer plan, and its results are saved for re-use in later rows of the outer plan. So in this example `random()` is evaluated only once and all the values of `t1.ten` are compared to the same randomly-chosen integer. That's quite different from what would happen without the sub-`SELECT` construct.

14.1.2. EXPLAIN ANALYZE

It is possible to check the accuracy of the planner's estimates by using `EXPLAIN`'s `ANALYZE` option. With this option, `EXPLAIN` actually executes the query, and then displays the true row counts and true run time accumulated within each plan node, along with the same estimates that a plain `EXPLAIN` shows. For example, we might get a result like this:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                  QUERY
 PLAN
-----------------------------------------------------------------------
-------------------------------------------------------------
 Nested Loop  (cost=4.65..118.50 rows=10 width=488) (actual
 time=0.017..0.051 rows=10 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10
 width=244) (actual time=0.009..0.017 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         Heap Blocks: exact=10
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
 rows=10 width=0) (actual time=0.004..0.004 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.90
 rows=1 width=244) (actual time=0.003..0.003 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning Time: 0.485 ms
 Execution Time: 0.073 ms
```

Note that the “actual time” values are in milliseconds of real time, whereas the `cost` estimates are expressed in arbitrary units; so they are unlikely to match up. The thing that's usually most important to look for is whether the estimated row counts are reasonably close to reality. In this example the estimates were all dead-on, but that's quite unusual in practice.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan will be executed once per outer row in the above nested-loop plan. In such cases, the `loops` value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the `loops` value to get the total time actually spent in the node. In the above example, we spent a total of 0.030 milliseconds executing the index scans on `tenk2`.

In some cases `EXPLAIN ANALYZE` shows additional execution statistics beyond the plan node execution times and row counts. For example, Sort and Hash nodes provide extra information:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY
 t1.fivethous;
```

```
 QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------------------------------
------
 Sort  (cost=713.05..713.30 rows=100 width=488) (actual
 time=2.995..3.002 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 74kB
   ->  Hash Join  (cost=226.23..709.73 rows=100 width=488) (actual
 time=0.515..2.920 rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000
 width=244) (actual time=0.026..1.790 rows=10000 loops=1)
         ->  Hash  (cost=224.98..224.98 rows=100 width=244) (actual
 time=0.476..0.477 rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 35kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98
 rows=100 width=244) (actual time=0.030..0.450 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     Heap Blocks: exact=90
                     ->  Bitmap Index Scan on tenk1_unique1
 (cost=0.00..5.04 rows=100 width=0) (actual time=0.013..0.013
 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning Time: 0.187 ms
 Execution Time: 3.036 ms
```

The Sort node shows the sort method used (in particular, whether the sort was in-memory or on-disk) and the amount of memory or disk space needed. The Hash node shows the number of hash buckets and batches as well as the peak amount of memory used for the hash table. (If the number of batches exceeds one, there will also be disk space usage involved, but that is not shown.)

Another type of extra information is the number of rows removed by a filter condition:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                  QUERY PLAN
-------------------------------------------------------------------
-------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=7000 width=244) (actual
 time=0.030..1.995 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning Time: 0.102 ms
 Execution Time: 2.145 ms
```

These counts can be particularly valuable for filter conditions applied at join nodes. The “Rows Removed” line only appears when at least one scanned row, or potential join pair in the case of a join node, is rejected by the filter condition.

A case similar to filter conditions occurs with “lossy” index scans. For example, consider this search for polygons containing a specific point:

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon
 '(0.5,2.0)';
```

```
                                                       QUERY PLAN
-------------------------------------------------------------------------
-----------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.09 rows=1 width=85) (actual
 time=0.023..0.023 rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 7
 Planning Time: 0.039 ms
 Execution Time: 0.033 ms
```

The planner thinks (quite correctly) that this sample table is too small to bother with an index scan, so we have a plain sequential scan in which all the rows got rejected by the filter condition. But if we force an index scan to be used, we see:

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon
 '(0.5,2.0)';

                                                             QUERY PLAN
-------------------------------------------------------------------------
-------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15
 rows=1 width=85) (actual time=0.074..0.074 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning Time: 0.039 ms
 Execution Time: 0.098 ms
```

Here we can see that the index returned one candidate row, which was then rejected by a recheck of the index condition. This happens because a GiST index is “lossy” for polygon containment tests: it actually returns the rows with polygons that overlap the target, and then we have to do the exact containment test on those rows.

`EXPLAIN` has a `BUFFERS` option that can be used with `ANALYZE` to get even more run time statistics:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100
 AND unique2 > 9000;

                                                                QUERY
 PLAN
-------------------------------------------------------------------------
-------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.07..60.11 rows=10 width=244)
 (actual time=0.105..0.114 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Heap Blocks: exact=10
   Buffers: shared hit=14 read=3
   ->  BitmapAnd  (cost=25.07..25.07 rows=10 width=0) (actual
 time=0.100..0.101 rows=0 loops=1)
         Buffers: shared hit=4 read=3
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=100 width=0) (actual time=0.027..0.027 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78
 rows=999 width=0) (actual time=0.070..0.070 rows=999 loops=1)
```

```
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=2 read=3
 Planning:
   Buffers: shared hit=3
 Planning Time: 0.162 ms
 Execution Time: 0.143 ms
```

The numbers provided by BUFFERS help to identify which parts of the query are the most I/O-intensive.

Keep in mind that because EXPLAIN ANALYZE actually runs the query, any side-effects will happen as usual, even though whatever results the query might output are discarded in favor of printing the EXPLAIN data. If you want to analyze a data-modifying query without changing your tables, you can roll the command back afterwards, for example:

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE
 unique1 < 100;

                                                                QUERY
 PLAN
-------------------------------------------------------------------
-------------------------------------------------------------
 Update on tenk1  (cost=5.06..225.23 rows=0 width=0) (actual
 time=1.634..1.635 rows=0 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.06..225.23 rows=100
 width=10) (actual time=0.065..0.141 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         Heap Blocks: exact=90
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=100 width=0) (actual time=0.031..0.031 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning Time: 0.151 ms
 Execution Time: 1.856 ms

ROLLBACK;
```

As seen in this example, when the query is an INSERT, UPDATE, DELETE, or MERGE command, the actual work of applying the table changes is done by a top-level Insert, Update, Delete, or Merge plan node. The plan nodes underneath this node perform the work of locating the old rows and/or computing the new data. So above, we see the same sort of bitmap table scan we've seen already, and its output is fed to an Update node that stores the updated rows. It's worth noting that although the data-modifying node can take a considerable amount of run time (here, it's consuming the lion's share of the time), the planner does not currently add anything to the cost estimates to account for that work. That's because the work to be done is the same for every correct query plan, so it doesn't affect planning decisions.

When an UPDATE, DELETE, or MERGE command affects a partitioned table or inheritance hierarchy, the output might look like this:

```
EXPLAIN UPDATE gtest_parent SET f1 = CURRENT_DATE WHERE f2 = 101;

                                       QUERY PLAN
-------------------------------------------------------------------
---------------------
 Update on gtest_parent  (cost=0.00..3.06 rows=0 width=0)
   Update on gtest_child gtest_parent_1
```

```
  Update on gtest_child2 gtest_parent_2
  Update on gtest_child3 gtest_parent_3
  ->  Append  (cost=0.00..3.06 rows=3 width=14)
        ->  Seq Scan on gtest_child gtest_parent_1
(cost=0.00..1.01 rows=1 width=14)
              Filter: (f2 = 101)
        ->  Seq Scan on gtest_child2 gtest_parent_2
(cost=0.00..1.01 rows=1 width=14)
              Filter: (f2 = 101)
        ->  Seq Scan on gtest_child3 gtest_parent_3
(cost=0.00..1.01 rows=1 width=14)
              Filter: (f2 = 101)
```

In this example the Update node needs to consider three child tables, but not the originally-mentioned partitioned table (since that never stores any data). So there are three input scanning subplans, one per table. For clarity, the Update node is annotated to show the specific target tables that will be updated, in the same order as the corresponding subplans.

The `Planning time` shown by `EXPLAIN ANALYZE` is the time it took to generate the query plan from the parsed query and optimize it. It does not include parsing or rewriting.

The `Execution time` shown by `EXPLAIN ANALYZE` includes executor start-up and shut-down time, as well as the time to run any triggers that are fired, but it does not include parsing, rewriting, or planning time. Time spent executing `BEFORE` triggers, if any, is included in the time for the related Insert, Update, or Delete node; but time spent executing `AFTER` triggers is not counted there because `AFTER` triggers are fired after completion of the whole plan. The total time spent in each trigger (either `BEFORE` or `AFTER`) is also shown separately. Note that deferred constraint triggers will not be executed until end of transaction and are thus not considered at all by `EXPLAIN ANALYZE`.

The time shown for the top-level node does not include any time needed to convert the query's output data into displayable form or to send it to the client. While `EXPLAIN ANALYZE` will never send the data to the client, it can be told to convert the query's output data to displayable form and measure the time needed for that, by specifying the `SERIALIZE` option. That time will be shown separately, and it's also included in the total `Execution time`.

14.1.3. Caveats

There are two significant ways in which run times measured by `EXPLAIN ANALYZE` can deviate from normal execution of the same query. First, since no output rows are delivered to the client, network transmission costs are not included. I/O conversion costs are not included either unless `SERIALIZE` is specified. Second, the measurement overhead added by `EXPLAIN ANALYZE` can be significant, especially on machines with slow `gettimeofday()` operating-system calls. You can use the pg_test_timing tool to measure the overhead of timing on your system.

`EXPLAIN` results should not be extrapolated to situations much different from the one you are actually testing; for example, results on a toy-sized table cannot be assumed to apply to large tables. The planner's cost estimates are not linear and so it might choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you'll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it's going to take one disk page read to process the table in any case, so there's no value in expending additional page reads to look at an index. (We saw this happening in the `polygon_tbl` example above.)

There are cases in which the actual and estimated values won't match up well, but nothing is really wrong. One such case occurs when plan node execution is stopped short by a `LIMIT` or similar effect. For example, in the `LIMIT` query we used before,

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2
 > 9000 LIMIT 2;
```

```
                                                                 QUERY
 PLAN
-----------------------------------------------------------------------
-------------------------------------------------------------
 Limit  (cost=0.29..14.33 rows=2 width=244) (actual
 time=0.051..0.071 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..70.50
 rows=10 width=244) (actual time=0.051..0.070 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning Time: 0.077 ms
 Execution Time: 0.086 ms
```

the estimated cost and row count for the Index Scan node are shown as though it were run to completion. But in reality the Limit node stopped requesting rows after it got two, so the actual row count is only 2 and the run time is less than the cost estimate would suggest. This is not an estimation error, only a discrepancy in the way the estimates and true values are displayed.

Merge joins also have measurement artifacts that can confuse the unwary. A merge join will stop reading one input if it's exhausted the other input and the next key value in the one input is greater than the last key value of the other input; in such a case there can be no more matches and so no need to scan the rest of the first input. This results in not reading all of one child, with results like those mentioned for `LIMIT`. Also, if the outer (first) child contains rows with duplicate key values, the inner (second) child is backed up and rescanned for the portion of its rows matching that key value. `EXPLAIN ANALYZE` counts these repeated emissions of the same inner rows as if they were real additional rows. When there are many outer duplicates, the reported actual row count for the inner child plan node can be significantly larger than the number of rows that are actually in the inner relation.

BitmapAnd and BitmapOr nodes always report their actual row counts as zero, due to implementation limitations.

Normally, `EXPLAIN` will display every plan node created by the planner. However, there are cases where the executor can determine that certain nodes need not be executed because they cannot produce any rows, based on parameter values that were not available at planning time. (Currently this can only happen for child nodes of an Append or MergeAppend node that is scanning a partitioned table.) When this happens, those plan nodes are omitted from the `EXPLAIN` output and a `Subplans Removed: N` annotation appears instead.

14.2. Statistics Used by the Planner

14.2.1. Single-Column Statistics

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table `pg_class`, in the columns `reltuples` and `relpages`. We can look at it with queries similar to this one:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
```

```
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      345
 tenk1_hundred        | i       |     10000 |       11
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Here we can see that `tenk1` contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, `reltuples` and `relpages` are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`. A `VACUUM` or `ANALYZE` operation that does not scan the entire table (which is commonly the case) will incrementally update the `reltuples` count on the basis of the part of the table it did scan, resulting in an approximate value. In any case, the planner will scale the values it finds in `pg_class` to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to `WHERE` clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the *selectivity* of `WHERE` clauses, that is, the fraction of rows that match each condition in the `WHERE` clause. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by the `ANALYZE` and `VACUUM ANALYZE` commands, and are always approximate even when freshly updated.

Rather than look at `pg_statistic` directly, it's better to look at its view `pg_stats` when examining the statistics manually. `pg_stats` is designed to be more easily readable. Furthermore, `pg_stats` is readable by all, whereas `pg_statistic` is only readable by a superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The `pg_stats` view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------
+-----------------------------------
 name    | f         | -0.5681108 | I- 580
 Ramp+
         |           |            | I- 880
 Ramp+
         |           |            | Sp Railroad
   +
         |           |            | I- 580
   +
         |           |            | I- 680
 Ramp+
         |           |            | I- 80
 Ramp+
         |           |            | 14th
 St  +
         |           |            | I- 880
   +
```

```
         |           |            | Mac Arthur
 Blvd+
         |           |            | Mission
 Blvd+
...
 name    | t         |    -0.5125 | I- 580
 Ramp+
         |           |            | I- 880
 Ramp+
         |           |            | I- 580
   +
         |           |            | I- 680
 Ramp+
         |           |            | I- 80
 Ramp+
         |           |            | Sp Railroad
   +
         |           |            | I- 880
   +
         |           |            | State Hwy 13
 Ramp+
         |           |            | I- 80
   +
         |           |            | State Hwy 24
 Ramp+
...
 thepath | f         |          0 |
 thepath | t         |          0 |
(4 rows)
```

Note that two rows are displayed for the same column, one corresponding to the complete inheritance hierarchy starting at the `road` table (`inherited=t`), and another one including only the `road` table itself (`inherited=f`). (For brevity, we have only shown the first ten most-common values for the `name` column.)

The amount of information stored in `pg_statistic` by `ANALYZE`, in particular the maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays for each column, can be set on a column-by-column basis using the `ALTER TABLE SET STATISTICS` command, or globally by setting the default_statistics_target configuration variable. The default limit is presently 100 entries. Raising the limit might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit might be sufficient for columns with simple data distributions.

Further details about the planner's use of statistics can be found in Chapter 68.

14.2.2. Extended Statistics

It is common to see slow queries running bad execution plans because multiple columns used in the query clauses are correlated. The planner normally assumes that multiple conditions are independent of each other, an assumption that does not hold when column values are correlated. Regular statistics, because of their per-individual-column nature, cannot capture any knowledge about cross-column correlation. However, PostgreSQL has the ability to compute *multivariate statistics*, which can capture such information.

Because the number of possible column combinations is very large, it's impractical to compute multivariate statistics automatically. Instead, *extended statistics objects*, more often called just *statistics objects*, can be created to instruct the server to obtain statistics across interesting sets of columns.

Statistics objects are created using the CREATE STATISTICS command. Creation of such an object merely creates a catalog entry expressing interest in the statistics. Actual data collection is performed by ANALYZE (either a manual command, or background auto-analyze). The collected values can be examined in the pg_statistic_ext_data catalog.

ANALYZE computes extended statistics based on the same sample of table rows that it takes for computing regular single-column statistics. Since the sample size is increased by increasing the statistics target for the table or any of its columns (as described in the previous section), a larger statistics target will normally result in more accurate extended statistics, as well as more time spent calculating them.

The following subsections describe the kinds of extended statistics that are currently supported.

14.2.2.1. Functional Dependencies

The simplest kind of extended statistics tracks *functional dependencies*, a concept used in definitions of database normal forms. We say that column b is functionally dependent on column a if knowledge of the value of a is sufficient to determine the value of b, that is there are no two rows having the same value of a but different values of b. In a fully normalized database, functional dependencies should exist only on primary keys and superkeys. However, in practice many data sets are not fully normalized for various reasons; intentional denormalization for performance reasons is a common example. Even in a fully normalized database, there may be partial correlation between some columns, which can be expressed as partial functional dependency.

The existence of functional dependencies directly affects the accuracy of estimates in certain queries. If a query contains conditions on both the independent and the dependent column(s), the conditions on the dependent columns do not further reduce the result size; but without knowledge of the functional dependency, the query planner will assume that the conditions are independent, resulting in underestimating the result size.

To inform the planner about functional dependencies, ANALYZE can collect measurements of cross-column dependency. Assessing the degree of dependency between all sets of columns would be prohibitively expensive, so data collection is limited to those groups of columns appearing together in a statistics object defined with the dependencies option. It is advisable to create dependencies statistics only for column groups that are strongly correlated, to avoid unnecessary overhead in both ANALYZE and later query planning.

Here is an example of collecting functional-dependency statistics:

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid =
 stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
---------+---------+------------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

Here it can be seen that column 1 (zip code) fully determines column 5 (city) so the coefficient is 1.0, while city only determines zip code about 42% of the time, meaning that there are many cities (58%) that are represented by more than a single ZIP code.

When computing the selectivity for a query involving functionally dependent columns, the planner adjusts the per-condition selectivity estimates using the dependency coefficients so as not to produce an underestimate.

14.2.2.1.1. Limitations of Functional Dependencies

Functional dependencies are currently only applied when considering simple equality conditions that compare columns to constant values, and `IN` clauses with constant values. They are not used to improve estimates for equality conditions comparing two columns or comparing a column to an expression, nor for range clauses, `LIKE` or any other type of condition.

When estimating with functional dependencies, the planner assumes that conditions on the involved columns are compatible and hence redundant. If they are incompatible, the correct estimate would be zero rows, but that possibility is not considered. For example, given a query like

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip =
 '94105';
```

the planner will disregard the `city` clause as not changing the selectivity, which is correct. However, it will make the same assumption about

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip =
 '90210';
```

even though there will really be zero rows satisfying this query. Functional dependency statistics do not provide enough information to conclude that, however.

In many practical situations, this assumption is usually satisfied; for example, there might be a GUI in the application that only allows selecting compatible city and ZIP code values to use in a query. But if that's not the case, functional dependencies may not be a viable option.

14.2.2.2. Multivariate N-Distinct Counts

Single-column statistics store the number of distinct values in each column. Estimates of the number of distinct values when combining more than one column (for example, for `GROUP BY a, b`) are frequently wrong when the planner only has single-column statistical data, causing it to select bad plans.

To improve such estimates, `ANALYZE` can collect n-distinct statistics for groups of columns. As before, it's impractical to do this for every possible column grouping, so data is collected only for those groups of columns appearing together in a statistics object defined with the `ndistinct` option. Data will be collected for each possible combination of two or more columns from the set of listed columns.

Continuing the previous example, the n-distinct counts in a table of ZIP codes might look like the following:

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM
 zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid =
 stxoid)
  WHERE stxname = 'stts2';
-[ RECORD 1 ]-----------------------------------------------------
--
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5":
 33178}
(1 row)
```

This indicates that there are three combinations of columns that have 33178 distinct values: ZIP code and state; ZIP code and city; and ZIP code, city and state (the fact that they are all equal is expected given that ZIP code alone is unique in this table). On the other hand, the combination of city and state has only 27435 distinct values.

It's advisable to create `ndistinct` statistics objects only on combinations of columns that are actually used for grouping, and for which misestimation of the number of groups is resulting in bad plans. Otherwise, the `ANALYZE` cycles are just wasted.

14.2.2.3. Multivariate MCV Lists

Another type of statistic stored for each column are most-common value lists. This allows very accurate estimates for individual columns, but may result in significant misestimates for queries with conditions on multiple columns.

To improve such estimates, `ANALYZE` can collect MCV lists on combinations of columns. Similarly to functional dependencies and n-distinct coefficients, it's impractical to do this for every possible column grouping. Even more so in this case, as the MCV list (unlike functional dependencies and n-distinct coefficients) does store the common column values. So data is collected only for those groups of columns appearing together in a statistics object defined with the `mcv` option.

Continuing the previous example, the MCV list for a table of ZIP codes might look like the following (unlike for simpler types of statistics, a function is required for inspection of MCV contents):

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid
 = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname =
 'stts3';

 index |         values         | nulls | frequency |
 base_frequency
-------+------------------------+-------+-----------
+----------------
     0 | {Washington, DC}       | {f,f} |  0.003467 |
 2.7e-05
     1 | {Apo, AE}              | {f,f} |  0.003067 |
 1.9e-05
     2 | {Houston, TX}          | {f,f} |  0.002167 |
 0.000133
     3 | {El Paso, TX}          | {f,f} |     0.002 |
 0.000113
     4 | {New York, NY}         | {f,f} |  0.001967 |
 0.000114
     5 | {Atlanta, GA}          | {f,f} |  0.001633 |
 3.3e-05
     6 | {Sacramento, CA}       | {f,f} |  0.001433 |
 7.8e-05
     7 | {Miami, FL}            | {f,f} |    0.0014 |
 6e-05
     8 | {Dallas, TX}           | {f,f} |  0.001367 |
 8.8e-05
     9 | {Chicago, IL}          | {f,f} |  0.001333 |
 5.1e-05
   ...
```

```
(99 rows)
```

This indicates that the most common combination of city and state is Washington in DC, with actual frequency (in the sample) about 0.35%. The base frequency of the combination (as computed from the simple per-column frequencies) is only 0.0027%, resulting in two orders of magnitude under-estimates.

It's advisable to create MCV statistics objects only on combinations of columns that are actually used in conditions together, and for which misestimation of the number of groups is resulting in bad plans. Otherwise, the ANALYZE and planning cycles are just wasted.

14.3. Controlling the Planner with Explicit JOIN Clauses

It is possible to control the query planner to some extent by using the explicit JOIN syntax. To see why this matters, we first need some background.

In a simple join query, such as:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the WHERE condition a.id = b.id, and then joins C to this joined table, using the other WHERE condition. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B — but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable condition in the WHERE clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it's necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but might have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren't many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning might take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a *genetic* probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the geqo_threshold run-time parameter.) The genetic search takes less time, but it won't necessarily find the best possible plan.

When the query involves outer joins, the planner has less freedom than it does for plain (inner) joins. For example, consider:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id =
 b.id);
```

Although this query's restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query. In other cases, the planner might be able to determine that more than one join order is safe. For example, given:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid
 = c.id);
```

it is valid to join A to either B or C first. Currently, only FULL JOIN completely constrains the join order. Most practical cases involving LEFT JOIN or RIGHT JOIN can be rearranged to some extent.

Explicit inner join syntax (INNER JOIN, CROSS JOIN, or unadorned JOIN) is semantically the same as listing the input relations in FROM, so it does not constrain the join order.

Even though most kinds of JOIN don't completely constrain the join order, it is possible to instruct the PostgreSQL query planner to treat all JOIN clauses as constraining the join order anyway. For example, these three queries are logically equivalent:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND
 b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

But if we tell the planner to honor the JOIN order, the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

To force the planner to follow the join order laid out by explicit JOINs, set the join_collapse_limit run-time parameter to 1. (Other possible values are discussed below.)

You do not need to constrain the join order completely in order to cut search time, because it's OK to use JOIN operators within items of a plain FROM list. For example, consider:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

With join_collapse_limit = 1, this forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via JOIN syntax — assuming that you know of a better order, that is. Experimentation is recommended.

A closely related issue that affects planning time is collapsing of subqueries into their parent query. For example, consider:

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

This situation might arise from use of a view that contains a join; the view's SELECT rule will be inserted in place of the view reference, yielding a query much like the above. Normally, the planner will try to collapse the subquery into the parent, yielding:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

This usually results in a better plan than planning the subquery separately. (For example, the outer WHERE conditions might be such that joining X to A first eliminates many rows of A, thus avoiding the need to form the full logical output of the subquery.) But at the same time, we have increased the planning time; here, we have a five-way join problem replacing two separate three-way join problems. Because of the exponential growth of the number of possibilities, this makes a big difference. The

planner tries to avoid getting stuck in huge join search problems by not collapsing a subquery if more than `from_collapse_limit FROM` items would result in the parent query. You can trade off planning time against quality of plan by adjusting this run-time parameter up or down.

from_collapse_limit and join_collapse_limit are similarly named because they do almost the same thing: one controls when the planner will “flatten out” subqueries, and the other controls when it will flatten out explicit joins. Typically you would either set `join_collapse_limit` equal to `from_collapse_limit` (so that explicit joins and subqueries act similarly) or set `join_collapse_limit` to 1 (if you want to control join order with explicit joins). But you might set them differently if you are trying to fine-tune the trade-off between planning time and run time.

14.4. Populating a Database

One might need to insert a large amount of data when first populating a database. This section contains some suggestions on how to make this process as efficient as possible.

14.4.1. Disable Autocommit

When using multiple `INSERT`s, turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing `BEGIN` at the start and `COMMIT` at the end. Some client libraries might do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each row that is added. An additional benefit of doing all insertions in one transaction is that if the insertion of one row were to fail then the insertion of all rows inserted up to that point would be rolled back, so you won't be stuck with partially loaded data.

14.4.2. Use `COPY`

Use `COPY` to load all the rows in one command, instead of using a series of `INSERT` commands. The `COPY` command is optimized for loading large numbers of rows; it is less flexible than `INSERT`, but incurs significantly less overhead for large data loads. Since `COPY` is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use `COPY`, it might help to use `PREPARE` to create a prepared `INSERT` statement, and then use `EXECUTE` as many times as required. This avoids some of the overhead of repeatedly parsing and planning `INSERT`. Different interfaces provide this facility in different ways; look for “prepared statements” in the interface documentation.

Note that loading a large number of rows using `COPY` is almost always faster than using `INSERT`, even if `PREPARE` is used and multiple insertions are batched into a single transaction.

`COPY` is fastest when used within the same transaction as an earlier `CREATE TABLE` or `TRUNCATE` command. In such cases no WAL needs to be written, because in case of an error, the files containing the newly loaded data will be removed anyway. However, this consideration only applies when wal_level is `minimal` as all commands must write WAL otherwise.

14.4.3. Remove Indexes

If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using `COPY`, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded.

If you are adding large amounts of data to an existing table, it might be a win to drop the indexes, load the table, and then recreate the indexes. Of course, the database performance for other users might suffer during the time the indexes are missing. One should also think twice before dropping a unique index, since the error checking afforded by the unique constraint will be lost while the index is missing.

14.4.4. Remove Foreign Key Constraints

Just as with indexes, a foreign key constraint can be checked “in bulk” more efficiently than row-by-row. So it might be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

What's more, when you load data into a table with existing foreign key constraints, each new row requires an entry in the server's list of pending trigger events (since it is the firing of a trigger that checks the row's foreign key constraint). Loading many millions of rows can cause the trigger event queue to overflow available memory, leading to intolerable swapping or even outright failure of the command. Therefore it may be *necessary*, not just desirable, to drop and re-apply foreign keys when loading large amounts of data. If temporarily removing the constraint isn't acceptable, the only other recourse may be to split up the load operation into smaller transactions.

14.4.5. Increase `maintenance_work_mem`

Temporarily increasing the maintenance_work_mem configuration variable when loading large amounts of data can lead to improved performance. This will help to speed up `CREATE INDEX` commands and `ALTER TABLE ADD FOREIGN KEY` commands. It won't do much for `COPY` itself, so this advice is only useful when you are using one or both of the above techniques.

14.4.6. Increase `max_wal_size`

Temporarily increasing the max_wal_size configuration variable can also make large data loads faster. This is because loading a large amount of data into PostgreSQL will cause checkpoints to occur more often than the normal checkpoint frequency (specified by the `checkpoint_timeout` configuration variable). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing `max_wal_size` temporarily during bulk data loads, the number of checkpoints that are required can be reduced.

14.4.7. Disable WAL Archival and Streaming Replication

When loading large amounts of data into an installation that uses WAL archiving or streaming replication, it might be faster to take a new base backup after the load has completed than to process a large amount of incremental WAL data. To prevent incremental WAL logging while loading, disable archiving and streaming replication, by setting wal_level to `minimal`, archive_mode to `off`, and max_wal_senders to zero. But note that changing these settings requires a server restart, and makes any base backups taken before unavailable for archive recovery and standby server, which may lead to data loss.

Aside from avoiding the time for the archiver or WAL sender to process the WAL data, doing this will actually make certain commands faster, because they do not to write WAL at all if `wal_level` is `minimal` and the current subtransaction (or top-level transaction) created or truncated the table or index they change. (They can guarantee crash safety more cheaply by doing an `fsync` at the end than by writing WAL.)

14.4.8. Run `ANALYZE` Afterwards

Whenever you have significantly altered the distribution of data within a table, running `ANALYZE` is strongly recommended. This includes bulk loading large amounts of data into the table. Running `ANALYZE` (or `VACUUM ANALYZE`) ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner might make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics. Note that if the

autovacuum daemon is enabled, it might run ANALYZE automatically; see Section 24.1.3 and Section 24.1.6 for more information.

14.4.9. Some Notes about pg_dump

Dump scripts generated by pg_dump automatically apply several, but not all, of the above guidelines. To restore a pg_dump dump as quickly as possible, you need to do a few extra things manually. (Note that these points apply while *restoring* a dump, not while *creating* it. The same points apply whether loading a text dump with psql or using pg_restore to load from a pg_dump archive file.)

By default, pg_dump uses COPY, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case several guidelines are handled automatically. What is left for you to do is to:

- Set appropriate (i.e., larger than normal) values for maintenance_work_mem and max_wal_size.
- If using WAL archiving or streaming replication, consider disabling them during the restore. To do that, set archive_mode to off, wal_level to minimal, and max_wal_senders to zero before loading the dump. Afterwards, set them back to the right values and take a fresh base backup.
- Experiment with the parallel dump and restore modes of both pg_dump and pg_restore and find the optimal number of concurrent jobs to use. Dumping and restoring in parallel by means of the -j option should give you a significantly higher performance over the serial mode.
- Consider whether the whole dump should be restored as a single transaction. To do that, pass the -1 or --single-transaction command-line option to psql or pg_restore. When using this mode, even the smallest of errors will rollback the entire restore, possibly discarding many hours of processing. Depending on how interrelated the data is, that might seem preferable to manual cleanup, or not. COPY commands will run fastest if you use a single transaction and have WAL archiving turned off.
- If multiple CPUs are available in the database server, consider using pg_restore's --jobs option. This allows concurrent data loading and index creation.
- Run ANALYZE afterwards.

A data-only dump will still use COPY, but it does not drop or recreate indexes, and it does not normally touch foreign keys. [1] So when loading a data-only dump, it is up to you to drop and recreate indexes and foreign keys if you wish to use those techniques. It's still useful to increase max_wal_size while loading the data, but don't bother increasing maintenance_work_mem; rather, you'd do that while manually recreating indexes and foreign keys afterwards. And don't forget to ANALYZE when you're done; see Section 24.1.3 and Section 24.1.6 for more information.

14.5. Non-Durable Settings

Durability is a database feature that guarantees the recording of committed transactions even if the server crashes or loses power. However, durability adds significant database overhead, so if your site does not require such a guarantee, PostgreSQL can be configured to run much faster. The following are configuration changes you can make to improve performance in such cases. Except as noted below, durability is still guaranteed in case of a crash of the database software; only an abrupt operating system crash creates a risk of data loss or corruption when these settings are used.

- Place the database cluster's data directory in a memory-backed file system (i.e., RAM disk). This eliminates all database disk I/O, but limits data storage to the amount of available memory (and perhaps swap).

[1] You can get the effect of disabling foreign keys by using the --disable-triggers option — but realize that that eliminates, rather than just postpones, foreign key validation, and so it is possible to insert bad data if you use it.

- Turn off fsync; there is no need to flush data to disk.
- Turn off synchronous_commit; there might be no need to force WAL writes to disk on every commit. This setting does risk transaction loss (though not data corruption) in case of a crash of the *database*.
- Turn off full_page_writes; there is no need to guard against partial page writes.
- Increase max_wal_size and checkpoint_timeout; this reduces the frequency of checkpoints, but increases the storage requirements of `/pg_wal`.
- Create unlogged tables to avoid WAL writes, though it makes the tables non-crash-safe.

Chapter 15. Parallel Query

PostgreSQL can devise query plans that can leverage multiple CPUs in order to answer queries faster. This feature is known as parallel query. Many queries cannot benefit from parallel query, either due to limitations of the current implementation or because there is no imaginable query plan that is any faster than the serial query plan. However, for queries that can benefit, the speedup from parallel query is often very significant. Many queries can run more than twice as fast when using parallel query, and some queries can run four times faster or even more. Queries that touch a large amount of data but return only a few rows to the user will typically benefit most. This chapter explains some details of how parallel query works and in which situations it can be used so that users who wish to make use of it can understand what to expect.

15.1. How Parallel Query Works

When the optimizer determines that parallel query is the fastest execution strategy for a particular query, it will create a query plan that includes a *Gather* or *Gather Merge* node. Here is a simple example:

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------
------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33
 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

In all cases, the `Gather` or `Gather Merge` node will have exactly one child plan, which is the portion of the plan that will be executed in parallel. If the `Gather` or `Gather Merge` node is at the very top of the plan tree, then the entire query will execute in parallel. If it is somewhere else in the plan tree, then only the portion of the plan below it will run in parallel. In the example above, the query accesses only one table, so there is only one plan node other than the `Gather` node itself; since that plan node is a child of the `Gather` node, it will run in parallel.

Using EXPLAIN, you can see the number of workers chosen by the planner. When the `Gather` node is reached during query execution, the process that is implementing the user's session will request a number of background worker processes equal to the number of workers chosen by the planner. The number of background workers that the planner will consider using is limited to at most max_parallel_workers_per_gather. The total number of background workers that can exist at any one time is limited by both max_worker_processes and max_parallel_workers. Therefore, it is possible for a parallel query to run with fewer workers than planned, or even with no workers at all. The optimal plan may depend on the number of workers that are available, so this can result in poor query performance. If this occurrence is frequent, consider increasing `max_worker_processes` and `max_parallel_workers` so that more workers can be run simultaneously or alternatively reducing `max_parallel_workers_per_gather` so that the planner requests fewer workers.

Every background worker process that is successfully started for a given parallel query will execute the parallel portion of the plan. The leader will also execute that portion of the plan, but it has an additional responsibility: it must also read all of the tuples generated by the workers. When the parallel portion of the plan generates only a small number of tuples, the leader will often behave very much like an additional worker, speeding up query execution. Conversely, when the parallel portion of the plan generates a large number of tuples, the leader may be almost entirely occupied with reading the tuples generated by the workers and performing any further processing steps that are required by plan

nodes above the level of the `Gather` node or `Gather Merge` node. In such cases, the leader will do very little of the work of executing the parallel portion of the plan.

When the node at the top of the parallel portion of the plan is `Gather Merge` rather than `Gather`, it indicates that each process executing the parallel portion of the plan is producing tuples in sorted order, and that the leader is performing an order-preserving merge. In contrast, `Gather` reads tuples from the workers in whatever order is convenient, destroying any sort order that may have existed.

15.2. When Can Parallel Query Be Used?

There are several settings that can cause the query planner not to generate a parallel query plan under any circumstances. In order for any parallel query plans whatsoever to be generated, the following settings must be configured as indicated.

- max_parallel_workers_per_gather must be set to a value that is greater than zero. This is a special case of the more general principle that no more workers should be used than the number configured via `max_parallel_workers_per_gather`.

In addition, the system must not be running in single-user mode. Since the entire database system is running as a single process in this situation, no background workers will be available.

Even when it is in general possible for parallel query plans to be generated, the planner will not generate them for a given query if any of the following are true:

- The query writes any data or locks any database rows. If a query contains a data-modifying operation either at the top level or within a CTE, no parallel plans for that query will be generated. As an exception, the following commands, which create a new table and populate it, can use a parallel plan for the underlying `SELECT` part of the query:
  - `CREATE TABLE ... AS`
  - `SELECT INTO`
  - `CREATE MATERIALIZED VIEW`
  - `REFRESH MATERIALIZED VIEW`
- The query might be suspended during execution. In any situation in which the system thinks that partial or incremental execution might occur, no parallel plan is generated. For example, a cursor created using DECLARE CURSOR will never use a parallel plan. Similarly, a PL/pgSQL loop of the form `FOR x IN query LOOP .. END LOOP` will never use a parallel plan, because the parallel query system is unable to verify that the code in the loop is safe to execute while parallel query is active.
- The query uses any function marked `PARALLEL UNSAFE`. Most system-defined functions are `PARALLEL SAFE`, but user-defined functions are marked `PARALLEL UNSAFE` by default. See the discussion of Section 15.4.
- The query is running inside of another query that is already parallel. For example, if a function called by a parallel query issues an SQL query itself, that query will never use a parallel plan. This is a limitation of the current implementation, but it may not be desirable to remove this limitation, since it could result in a single query using a very large number of processes.

Even when a parallel query plan is generated for a particular query, there are several circumstances under which it will be impossible to execute that plan in parallel at execution time. If this occurs, the leader will execute the portion of the plan below the `Gather` node entirely by itself, almost as if the `Gather` node were not present. This will happen if any of the following conditions are met:

- No background workers can be obtained because of the limitation that the total number of background workers cannot exceed max_worker_processes.

- No background workers can be obtained because of the limitation that the total number of background workers launched for purposes of parallel query cannot exceed max_parallel_workers.

- The client sends an Execute message with a non-zero fetch count. See the discussion of the extended query protocol. Since libpq currently provides no way to send such a message, this can only occur when using a client that does not rely on libpq. If this is a frequent occurrence, it may be a good idea to set max_parallel_workers_per_gather to zero in sessions where it is likely, so as to avoid generating query plans that may be suboptimal when run serially.

15.3. Parallel Plans

Because each worker executes the parallel portion of the plan to completion, it is not possible to simply take an ordinary query plan and run it using multiple workers. Each worker would produce a full copy of the output result set, so the query would not run any faster than normal but would produce incorrect results. Instead, the parallel portion of the plan must be what is known internally to the query optimizer as a *partial plan*; that is, it must be constructed so that each process that executes the plan will generate only a subset of the output rows in such a way that each required output row is guaranteed to be generated by exactly one of the cooperating processes. Generally, this means that the scan on the driving table of the query must be a parallel-aware scan.

15.3.1. Parallel Scans

The following types of parallel-aware table scans are currently supported.

- In a *parallel sequential scan*, the table's blocks will be divided into ranges and shared among the cooperating processes. Each worker process will complete the scanning of its given range of blocks before requesting an additional range of blocks.

- In a *parallel bitmap heap scan*, one process is chosen as the leader. That process performs a scan of one or more indexes and builds a bitmap indicating which table blocks need to be visited. These blocks are then divided among the cooperating processes as in a parallel sequential scan. In other words, the heap scan is performed in parallel, but the underlying index scan is not.

- In a *parallel index scan* or *parallel index-only scan*, the cooperating processes take turns reading data from the index. Currently, parallel index scans are supported only for btree indexes. Each process will claim a single index block and will scan and return all tuples referenced by that block; other processes can at the same time be returning tuples from a different index block. The results of a parallel btree scan are returned in sorted order within each worker process.

Other scan types, such as scans of non-btree indexes, may support parallel scans in the future.

15.3.2. Parallel Joins

Just as in a non-parallel plan, the driving table may be joined to one or more other tables using a nested loop, hash join, or merge join. The inner side of the join may be any kind of non-parallel plan that is otherwise supported by the planner provided that it is safe to run within a parallel worker. Depending on the join type, the inner side may also be a parallel plan.

- In a *nested loop join*, the inner side is always non-parallel. Although it is executed in full, this is efficient if the inner side is an index scan, because the outer tuples and thus the loops that look up values in the index are divided over the cooperating processes.

- In a *merge join*, the inner side is always a non-parallel plan and therefore executed in full. This may be inefficient, especially if a sort must be performed, because the work and resulting data are duplicated in every cooperating process.

- In a *hash join* (without the "parallel" prefix), the inner side is executed in full by every cooperating process to build identical copies of the hash table. This may be inefficient if the hash table is large

or the plan is expensive. In a *parallel hash join*, the inner side is a *parallel hash* that divides the work of building a shared hash table over the cooperating processes.

15.3.3. Parallel Aggregation

PostgreSQL supports parallel aggregation by aggregating in two stages. First, each process participating in the parallel portion of the query performs an aggregation step, producing a partial result for each group of which that process is aware. This is reflected in the plan as a `Partial Aggregate` node. Second, the partial results are transferred to the leader via `Gather` or `Gather Merge`. Finally, the leader re-aggregates the results across all workers in order to produce the final result. This is reflected in the plan as a `Finalize Aggregate` node.

Because the `Finalize Aggregate` node runs on the leader process, queries that produce a relatively large number of groups in comparison to the number of input rows will appear less favorable to the query planner. For example, in the worst-case scenario the number of groups seen by the `Finalize Aggregate` node could be as many as the number of input rows that were seen by all worker processes in the `Partial Aggregate` stage. For such cases, there is clearly going to be no performance benefit to using parallel aggregation. The query planner takes this into account during the planning process and is unlikely to choose parallel aggregate in this scenario.

Parallel aggregation is not supported in all situations. Each aggregate must be safe for parallelism and must have a combine function. If the aggregate has a transition state of type `internal`, it must have serialization and deserialization functions. See CREATE AGGREGATE for more details. Parallel aggregation is not supported if any aggregate function call contains `DISTINCT` or `ORDER BY` clause and is also not supported for ordered set aggregates or when the query involves `GROUPING SETS`. It can only be used when all joins involved in the query are also part of the parallel portion of the plan.

15.3.4. Parallel Append

Whenever PostgreSQL needs to combine rows from multiple sources into a single result set, it uses an `Append` or `MergeAppend` plan node. This commonly happens when implementing `UNION ALL` or when scanning a partitioned table. Such nodes can be used in parallel plans just as they can in any other plan. However, in a parallel plan, the planner may instead use a `Parallel Append` node.

When an `Append` node is used in a parallel plan, each process will execute the child plans in the order in which they appear, so that all participating processes cooperate to execute the first child plan until it is complete and then move to the second plan at around the same time. When a `Parallel Append` is used instead, the executor will instead spread out the participating processes as evenly as possible across its child plans, so that multiple child plans are executed simultaneously. This avoids contention, and also avoids paying the startup cost of a child plan in those processes that never execute it.

Also, unlike a regular `Append` node, which can only have partial children when used within a parallel plan, a `Parallel Append` node can have both partial and non-partial child plans. Non-partial children will be scanned by only a single process, since scanning them more than once would produce duplicate results. Plans that involve appending multiple results sets can therefore achieve coarse-grained parallelism even when efficient partial plans are not available. For example, consider a query against a partitioned table that can only be implemented efficiently by using an index that does not support parallel scans. The planner might choose a `Parallel Append` of regular `Index Scan` plans; each individual index scan would have to be executed to completion by a single process, but different scans could be performed at the same time by different processes.

enable_parallel_append can be used to disable this feature.

15.3.5. Parallel Plan Tips

If a query that is expected to do so does not produce a parallel plan, you can try reducing parallel_setup_cost or parallel_tuple_cost. Of course, this plan may turn out to be slower than the serial plan that the planner preferred, but this will not always be the case. If you don't get a parallel plan even with very

small values of these settings (e.g., after setting them both to zero), there may be some reason why the query planner is unable to generate a parallel plan for your query. See Section 15.2 and Section 15.4 for information on why this may be the case.

When executing a parallel plan, you can use `EXPLAIN (ANALYZE, VERBOSE)` to display per-worker statistics for each plan node. This may be useful in determining whether the work is being evenly distributed between all plan nodes and more generally in understanding the performance characteristics of the plan.

15.4. Parallel Safety

The planner classifies operations involved in a query as either *parallel safe*, *parallel restricted*, or *parallel unsafe*. A parallel safe operation is one that does not conflict with the use of parallel query. A parallel restricted operation is one that cannot be performed in a parallel worker, but that can be performed in the leader while parallel query is in use. Therefore, parallel restricted operations can never occur below a `Gather` or `Gather Merge` node, but can occur elsewhere in a plan that contains such a node. A parallel unsafe operation is one that cannot be performed while parallel query is in use, not even in the leader. When a query contains anything that is parallel unsafe, parallel query is completely disabled for that query.

The following operations are always parallel restricted:

- Scans of common table expressions (CTEs).
- Scans of temporary tables.
- Scans of foreign tables, unless the foreign data wrapper has an `IsForeignScanParallelSafe` API that indicates otherwise.
- Plan nodes that reference a correlated `SubPlan`.

15.4.1. Parallel Labeling for Functions and Aggregates

The planner cannot automatically determine whether a user-defined function or aggregate is parallel safe, parallel restricted, or parallel unsafe, because this would require predicting every operation that the function could possibly perform. In general, this is equivalent to the Halting Problem and therefore impossible. Even for simple functions where it could conceivably be done, we do not try, since this would be expensive and error-prone. Instead, all user-defined functions are assumed to be parallel unsafe unless otherwise marked. When using CREATE FUNCTION or ALTER FUNCTION, markings can be set by specifying `PARALLEL SAFE`, `PARALLEL RESTRICTED`, or `PARALLEL UNSAFE` as appropriate. When using CREATE AGGREGATE, the `PARALLEL` option can be specified with `SAFE`, `RESTRICTED`, or `UNSAFE` as the corresponding value.

Functions and aggregates must be marked `PARALLEL UNSAFE` if they write to the database, change the transaction state (other than by using a subtransaction for error recovery), access sequences, or make persistent changes to settings. Similarly, functions must be marked `PARALLEL RESTRICTED` if they access temporary tables, client connection state, cursors, prepared statements, or miscellaneous backend-local state that the system cannot synchronize across workers. For example, `setseed` and `random` are parallel restricted for this last reason.

In general, if a function is labeled as being safe when it is restricted or unsafe, or if it is labeled as being restricted when it is in fact unsafe, it may throw errors or produce wrong answers when used in a parallel query. C-language functions could in theory exhibit totally undefined behavior if mislabeled, since there is no way for the system to protect itself against arbitrary C code, but in most likely cases the result will be no worse than for any other function. If in doubt, it is probably best to label functions as `UNSAFE`.

If a function executed within a parallel worker acquires locks that are not held by the leader, for example by querying a table not referenced in the query, those locks will be released at worker exit, not

end of transaction. If you write a function that does this, and this behavior difference is important to you, mark such functions as `PARALLEL RESTRICTED` to ensure that they execute only in the leader.

Note that the query planner does not consider deferring the evaluation of parallel-restricted functions or aggregates involved in the query in order to obtain a superior plan. So, for example, if a `WHERE` clause applied to a particular table is parallel restricted, the query planner will not consider performing a scan of that table in the parallel portion of a plan. In some cases, it would be possible (and perhaps even efficient) to include the scan of that table in the parallel portion of the query and defer the evaluation of the `WHERE` clause so that it happens above the `Gather` node. However, the planner does not do this.