PostgreSQL 7.1 User's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.1 User’s Guide

by The PostgreSQL Global Development Group

Legal Notice

Table of Contents

List of Tables

Preface

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2 (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html), developed at the University of California at Berkeley Computer Science Department. 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.

PostgreSQL is an open-source descendant of this original Berkeley code. It provides SQL92/SQL99 language support and other modern features.

POSTGRES pioneered many of the object-relational concepts now becoming available in some commercial databases. Traditional relational database management systems (RDBMS) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this model is inadequate for future data processing applications. The relational model successfully replaced previous models in part because of its Spartan simplicity . However, as mentioned, this simplicity often makes the implementation of certain applications very difficult. Postgres offers substantial additional power by incorporating the following additional concepts in such a way that users can easily extend the system:

inheritance
data types
functions

Other features provide additional power and flexibility:

constraints
triggers
rules
transaction integrity

These features put Postgres into the category of databases referred to as *object-relational*. Note that this is distinct from those referred to as *object-oriented*, which in general are not as well suited to supporting the traditional relational database languages. So, although Postgres has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by Postgres.

2. A Short History of Postgres

The object-relational database management system now known as PostgreSQL (and briefly called Postgres95) is derived from the Postgres package written at the University of California at Berkeley. With over a decade of development behind it, PostgreSQL is the most advanced open-source database available anywhere, offering multi-version concurrency control, supporting almost all SQL constructs (including subselects, transactions, and user-defined types and functions), and having a wide range of language bindings available (including C, C++, Java, Perl, Tcl, and Python).

2.1. The Berkeley Postgres Project

Implementation of the Postgres DBMS began in 1986. The initial concepts for the system were presented in *The Design of Postgres* and the definition of the initial data model appeared in *The Postgres Data Model*. The design of the rule system at that time was described in *The Design of the Postgres Rules System*. The rationale and architecture of the storage manager were detailed in *The Postgres Storage System*.

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. We released Version 1, described in *The Implementation of Postgres*, to a few external users in June 1989. In response to a critique of the first rule system (*A Commentary on the Postgres Rules System*), the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*) 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 rewrite rule system. For the most part, 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 (http://www.illustra.com/) (since merged into Informix (http://www.informix.com/)) picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000 (http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html) scientific computing project in late 1992.

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 project officially ended with Version 4.2.

2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to Postgres. 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 v1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to Postgres v4.2. Apart from bug fixes, these were the major enhancements:

- The query language Postquel was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregates were re-implemented. Support for the GROUP BY query clause was also added. The `libpq` interface remained available for C programs.
- In addition to the monitor program, a new program (psql) was provided for interactive SQL queries using GNU `readline`.
- 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 backend.
- 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 Postgres Project.

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

Major enhancements in PostgreSQL include:

Table-level locking has been replaced with multi-version concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database stays available for queries.

Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.

Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.

Built-in types have been improved, including new wide-range date/time types and additional geometric type support.

Overall backend code speed has been increased by approximately 20-40%, and backend start-up time has decreased 80% since version 6.0 was released.

3. Documentation Resources

This manual set is organized into several parts:

Tutorial

An introduction for new users. Does not cover advanced features.

User’s Guide

Documents the SQL query language environment, including data types and functions.

Programmer’s Guide

Advanced information for application programmers. Topics include type and function extensibility, library interfaces, and application design issues.

Administrator’s Guide

Installation and server management information

Reference Manual

Reference pages for SQL command syntax and client and server programs

Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the *Programmer's Guide*.

In addition to this manual set, there are other resources to help you with Postgres installation and use:

man pages

The *Reference Manual*'s pages in the traditional Unix man format.

FAQs

Frequently Asked Questions (FAQ) lists document both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The PostgreSQL web site (http://www.postgresql.org) carries details on the latest release, upcoming features, and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The `<pgsql-general@postgresql.org>` (archive (http://www.postgresql.org/mhonarc/pgsql-general/)) mailing list is a good place to have user questions answered. Other mailing lists are available; consult the User's Lounge (http://www.postgresql.org/users-lounge/) section of the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open source effort. 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. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute it.

Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The `<pgsql-docs@postgresql.org>` (archive (http://www.postgresql.org/mhonarc/pgsql-docs/)) mailing list is the place to get going.

4. Terminology and Notation

The terms Postgres and PostgreSQL will be used interchangeably to refer to the software that accompanies this documentation.

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 documentation set does not have fixed presumptions about system administration procedures.

`/usr/local/pgsql/` is generally used as the root directory of the installation and `/usr/local/pgsql/data` as the directory with the database files. These directories may vary on your site, details can be derived in the *Administrator's Guide*.

In a command synopsis, brackets ("`[`" and "`]`") indicate an optional phrase or keyword. Anything in braces ("`{`" and "`}`") and containing vertical bars ("`|`") indicates that you must choose one.

Examples will show commands executed from various accounts and programs. Commands executed from a Unix shell may be preceeded with a dollar sign (`$`). Commands executed from particular user accounts such as root or postgres are specially flagged and explained. SQL commands may be preceeded with `=>` or will have no leading prompt, depending on the context.

> **Note:** The notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the documentation mailing list `<pgsql-docs@postgresql.org>`.

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 it 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 the 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. 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 server.

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 SQL is not a bug 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 query-language 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 start at 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 reproduceable, 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 "mid-size 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:** In case of fatal errors, the error message provided by the client might not contain all the information available. In that case, also look at the log output of the database server. If you do not keep your server 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 okay 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 concerned environment variables or configuration files that you changed from the default. Again, be exact. If you are using a pre-packaged 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 `postmaster --version` and `psql --version` should work. If the function or the options do not exist then your version is probably old enough. You can also look into the `README` file in the source directory or at the name of your distribution file or package name. If you run a pre-packaged version, such as RPMs, say so, including any subversion the package may have. If you are talking about a CVS snapshot, mention that, including its date and time.

If your version is older than 7.1 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new release, that is why we make new releases.

Platform information. This includes the kernel name and version, C library, processor, memory information. 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 Pentiums. If you have installation problems then information about compilers, make, etc. 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.

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 choose non-confusing terminology. The software package as such is called "PostgreSQL", sometimes "Postgres" for short. (Sometimes the abbreviation "Pgsql" is used but don’t do that.) When you are specifically talking about the backend server, mention that, do not just say "Postgres crashes". The interactive frontend is called "psql" and is for all intends and purposes completely separate from the backend.

5.3. Where to report bugs

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

Do not send bug reports to any of the user mailing lists, such as `<pgsql-sql@postgresql.org>` or `<pgsql-general@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@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 it, if the bug needs more review.

If you have a problem with the documentation, send email to the documentation mailing list `<pgsql-docs@postgresql.org>`. Mention the document, chapter, and sections in your problem report.

If your bug is a portability problem on a non-supported platform, send mail to `<pgsql-ports@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 email addresses are closed mailing lists. That is, you need to be subscribed to a list to be allowed to post on it. If you simply want to send mail but do not want to receive list traffic, you can subscribe and set your subscription option to `nomail`. For more information send mail to `<majordomo@postgresql.org>` with the single word `help` in the body of the message.

6. Y2K Statement

> **Author:** Written by Thomas Lockhart (`<lockhart@alumni.caltech.edu>`) on 1998-10-22. Updated 2000-03-31.

The PostgreSQL Global Development Group provides the PostgreSQL software code tree as a public service, without warranty and without liability for its behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).

- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.

- To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current *User's Guide* in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. "`70-01-01`" is interpreted as 1970-01-01, whereas "`69-01-01`" is interpreted as 2069-01-01.

- Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

Refer to The Gnu Project (http://www.gnu.org/software/year2000.html) and The Perl Institute (http://language.perl.com/news/y2k.html) for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

Chapter 1. SQL Syntax

A description of the general syntax of SQL.

1.1. 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).

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

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).

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 the *Reference Manual*.

1.1.1. 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 B.

SQL identifiers and key words must begin with a letter (`a-z`) or underscore (_). Subsequent characters in an identifier or key word can be letters, digits (`0-9`), or underscores, although the SQL standard will not define a key word that contains digits or starts or ends with an underscore.

The system uses no more than NAMEDATALEN-1 characters of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 32 so the maximum identifier length is 31 (but at the time the system is built, NAMEDATALEN can be changed in `src/include/postgres_ext.h`).

Identifier and key word names 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 other than a double quote itself. 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 Postgres, but `"Foo"` and `"FOO"` are different from these three and each other. [1]

1.1.2. Constants

There are four kinds of *implicitly typed constants* in Postgres: strings, bit strings, integers, and floating point numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. The implicit constants are described below; explicit constants are discussed afterwards.

1.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes ('), e.g., `'This is a string'`. SQL allows single quotes to be embedded in strings by typing two adjacent single quotes (e.g., `'Dianne''s horse'`). In Postgres single quotes may alternatively be escaped with a backslash (\, e.g., `'Dianne\'s horse'`).

C-style backslash escapes are also available: `\b` is a backspace, `\f` is a form feed, `\n` is a newline, `\r` is a carriage return, `\t` is a tab, and `\xxx`, where *xxx* is an octal number, is the character with the corresponding ASCII code. Any other character following a backslash is taken literally. Thus, to include a backslash in a string constant, type two backslashes.

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

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 in one constant. For example:

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

is equivalent to

```
SELECT 'foobar';
```

but

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

is not valid syntax.

1.1.2.2. Bit String Constants

Bit string constants look like 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`. Bit string constants can be continued across lines in the same way as regular string constants.

1.1.2.3. Integer Constants

Integer constants in SQL are sequences of decimal digits (0 though 9) with no decimal point. The range of legal values depends on which integer data type is used, but the plain `integer` type accepts values ranging from -2147483648 to +2147483647. (The optional plus or minus sign is actually a separate unary operator and not part of the integer constant.)

1.1.2.4. Floating Point Constants

Floating point constants are accepted in these general forms:

```
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where *digits* is one or more decimal digits. At least one digit must be before or after the decimal point, and after the `e` if you use that option. Thus, a floating point constant is distinguished from an integer constant by the presence of either the decimal point or the exponent clause (or both). There must not be a space or other characters embedded in the constant.

These are some examples of valid floating point constants:
3.5
4.
.001
5e2
1.925e-3

Floating point constants are of type `DOUBLE PRECISION`. `REAL` can be specified explicitly by using SQL string notation or Postgres type notation:

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

1.1.2.5. 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 value inside the string 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 may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is passed as an argument to a non-overloaded function), in which case it is automatically coerced.

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

```
typename ( value )
```

although this only works for types whose names are also valid as function names. (For example, `double precision` can't be used this way --- but the equivalent `float8` can.)

The `::`, `CAST()`, and function-call syntaxes can also be used to specify the type of arbitrary expressions, but the form `type 'string'` can only be used to specify the type of a literal constant.

1.1.2.6. Array constants

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. (For all built-in types, this is the comma character ",".) Each `val` is either a constant of the array element type, or a sub-array. 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 sub-arrays of integers.

Individual array elements can be placed between double-quote marks (`"`) to avoid ambiguity problems with respect to white space. Without quote marks, the array-value parser will skip leading white space.

(Array constants are actually only a special case of the generic type constants discussed in the previous section. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

1.1.3. Operators

An operator is a sequence of up to NAMEDATALEN-1 (31 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ‘ ? $

There are a few restrictions on operator names, however:

"$" (dollar) cannot be a single-character operator, although it can be part of a multi-character operator name.

-- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.

A multi-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 Postgres 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 left-unary operator named "@", you cannot write X*@Y; you must write X* @Y to ensure that Postgres reads it as two operator names not one.

1.1.4. 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 the positional parameters in the body of a function definition. In other contexts the dollar sign may be part of an operator name.

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 Chapter 6 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 Chapter 6.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.

The asterisk (*) has a special meaning when used in the **SELECT** command or with the COUNT aggregate function.

The period (.) is used in floating point constants, and to separate table and column names.

1.1.5. Comments

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

```
-- This is a standard SQL92 comment
```

Alternatively, C-style block comments can be used:

```
/* multi-line 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 SQL99 but unlike C, so that one can comment out larger blocks of code that may contain existing block comments.

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

1.2. Columns

A *column* is either a user-defined column of a given table or one of the following system-defined columns:

oid

: The unique identifier (object ID) of a row. This is a serial number that is added by Postgres to all rows automatically. OIDs are not reused and are 32-bit quantities.

tableoid

: The OID of the table containing this row. This attribute is particularly handy for queries that select from inheritance hierarchies, since without it, it's difficult to tell which individual table a row came from. The tableoid can be joined against the OID attribute of pg_class to obtain the table name.

xmin

: The identity (transaction ID) of the inserting transaction for this tuple. (Note: a tuple is an individual state of a row; each UPDATE of a row creates a new tuple 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 tuple. In practice, this is never nonzero for a visible tuple.

cmax

: The command identifier within the deleting transaction, or zero. Again, this is never nonzero for a visible tuple.

ctid

: The tuple ID of the tuple within its table. This is a pair (block number, tuple index within block) that identifies the physical location of the tuple. Note that although the ctid can be used to locate the tuple very quickly, a row's ctid will change each time it is updated or moved by VACUUM. Therefore ctid is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

For further information on the system attributes consult *Stonebraker, Hanson, Hong, 1987*. Transaction and command identifiers are 32-bit quantities.

1.3. 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; see Section 1.1.2.

A column reference

An operator invocation:

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

where `operator` follows the syntax rules of Section 1.1.3 or is one of the tokens AND, OR, and NOT. 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 4 describes the built-in operators.

`( expression )`

Parentheses are used to group subexpressions and override precedence.

A positional parameter reference, in the body of a function declaration.

A function call

An aggregate expression

A scalar subquery. This is an ordinary **SELECT** in parentheses that returns exactly one row with one column. It is an error to use a subquery that returns more than one row or more than one column in the context of a value expression.

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 4. An example is the `IS NULL` clause.

We have already discussed constants in Section 1.1.2. The following sections discuss the remaining options.

1.3.1. Column References

A column can be referenced in the form:

`correlation.columnname ‘[’subscript‘]’`

`correlation` is either the name of a table, an alias for a table defined by means of a FROM clause, or the keyword `NEW` or `OLD`. (NEW and OLD can only appear in the action portion of a rule, while other correlation names can be used in any SQL statement.) The correlation name can be omitted if the column name is unique across all the tables being used in the current query. If `column` is of an array type, then the optional `subscript` selects a specific element in the array. If no subscript is provided, then the whole array is selected. Refer to the description of the particular commands in the *PostgreSQL Reference Manual* for the allowed syntax in each case.

1.3.2. Positional Parameters

A positional parameter reference is used to indicate a parameter in an SQL function. Typically this is used in SQL function definition statements. The form of a parameter 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` will be replaced by the first function argument when the function is invoked.

1.3.3. Function Calls

The syntax for a function call is the name of a function (which is subject to the syntax rules for identifiers of Section 1.1.1), followed by its argument list enclosed in parentheses:

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

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

```
sqrt(2)
```

The list of built-in functions is in Chapter 4. Other functions may be added by the user.

1.3.4. 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`*)
`aggregate_name` (ALL *`expression`*)
`aggregate_name` (DISTINCT *`expression`*)
`aggregate_name` (*)

where *`aggregate_name`* is a previously defined aggregate, and *`expression`* is any expression that does not itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression yields a non-NULL value. (Actually, it is up to the aggregate function whether to ignore NULLs or not --- but all the standard ones do.) The second form is the same as the first, since `ALL` is the default. The third form invokes the aggregate for all distinct non-NULL values of the expression found in the input rows. The last form invokes the aggregate once for each input row regardless of NULL or non-NULL values; since no particular input value is specified, it is generally only useful for the `count()` aggregate function.

For example, `count(*)` yields the total number of input rows; `count(f1)` yields the number of input rows in which `f1` is non-NULL; `count(distinct f1)` yields the number of distinct non-NULL values of `f1`.

The predefined aggregate functions are described in Section 4.12. Other aggregate functions may be added by the user.

1.4. Lexical Precedence

The precedence and associativity of the operators is hard-wired into the parser. Most operators have the same precedence and are left-associative. This may lead to non-intuitive behavior; for example the Boolean operators "<" and ">" have a different precedence than the Boolean operators "<=" and ">=". Also, you will sometimes need to add parentheses when using combinations of binary and unary operators. For instance

```
SELECT 5 ! - 6;
```

will be parsed as

```
SELECT 5 ! (- 6);
```

because the parser has no idea -- until it is too late -- that ! is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write

```
SELECT (5 !) - 6;
```

This is the price one pays for extensibility.

Table 1-1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
::	left	Postgres-style typecast
[ ]	left	array element selection
.	left	table/column name separator
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		containment
OVERLAPS		time interval overlap
LIKE ILIKE		string pattern matching

Operator/Element	Associativity	Description
< >		less than, greater than
=	right	equality, assignment
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.

Notes

1. Postgres' folding of unquoted names to lower case 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.

Chapter 2. Queries

A *query* is the process of retrieving or the command to retrieve data from a database. In SQL the **SELECT** command is used to specify queries. The general syntax of the **SELECT** command is

```
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. The simplest 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 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, client libraries will offer functions to retrieve individual rows and columns.) 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 even make calculations on the columns before retrieving them; see Section 2.2. 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 numeric data type).

`FROM table1` is a particularly simple kind of table expression. 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();
```

2.1. Table Expressions

A *table expression* specifies 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. The derived table that is produced by all these transformations provides the input rows used to compute output rows as specified by the select list of column value expressions.

2.1.1. 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 may be a table name or a derived table such as a subquery, a table join, or complex combinations of these. If more than one table reference is listed in the FROM clause they are CROSS JOINed (see below) to form the derived table that may 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 supertable of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its subtable successors, unless the keyword 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.

2.1.1.1. 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 JOIN are supported.

Join Types

CROSS JOIN

```
T1 CROSS JOIN T2
```

For each combination of rows from *T1* and *T2*, the derived 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. A cross join is equivalent to an `INNER JOIN ON TRUE`.

> **Tip:** `FROM T1 CROSS JOIN T2` is equivalent to `FROM T1, T2`.

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 for all JOINs. 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 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 for them.

USING is a shorthand notation: it takes a comma-separated list of column names, which the joined tables must have in common, and forms a join condition specifying equality of each of these pairs of columns. Furthermore, the output of a JOIN USING has one column for each of the equated pairs of input columns, followed by all of the other columns from each table. Thus, `USING (a, b, c)` is equivalent to `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)` with the exception that if ON is used there will be two columns a, b, and c in the result, whereas with USING there will be only one of each.

Finally, NATURAL is a shorthand form of USING: it forms a USING list consisting of exactly those column names that appear in both input tables. As with USING, these columns appear only once in the output table.

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 returned with NULL values in columns of T2. Thus, the joined table unconditionally 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 returned with NULL values in columns of T1. This is the converse of a left join: the result table will unconditionally 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 returned 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 returned.

Joins of all types can be chained together or nested: either or both of *`T1`* and *`T2`* may be JOINed tables. Parentheses may be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

2.1.1.2. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and *must* be named using an AS clause. (See Section 2.1.1.3.)

```
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which can't be reduced to a plain join, arise when the subquery involves grouping or aggregation.

2.1.1.3. 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 further processing. This is called a *table alias*.

```
FROM table_reference AS alias
```

Here, *`alias`* can be any regular identifier. The alias becomes the new name of the table reference for the current query -- it is no longer possible to refer to the table by the original name. Thus

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;
```

is not valid SQL syntax. What will actually happen (this is a Postgres extension to the standard) is that an implicit table reference is added to the FROM clause, so the query is processed as if it were written as

```
SELECT * FROM my_table AS m, my_table AS my_table WHERE my_table.a > 5;
```

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.,

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
```

Additionally, an alias is required if the table reference is a subquery.

Parentheses are used to resolve ambiguities. The following statement will assign the alias `b` to the result of the join, unlike the previous example:

```
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

```
FROM table_reference alias
```

This form is equivalent to the previously treated one; the AS key word is noise.

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

In this form, in addition to renaming the table as described above, the columns of the table are also given temporary names for use by the surrounding query. 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, using any of these forms, the alias hides the original names 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.

2.1.1.4. Examples

```
FROM T1 INNER JOIN T2 USING (C)
FROM T1 LEFT OUTER JOIN T2 USING (C)
FROM (T1 RIGHT OUTER JOIN T2 ON (T1C1=T2C1)) AS DT1
FROM (T1 FULL OUTER JOIN T2 USING (C)) AS DT1 (DT1C1, DT1C2)

FROM T1 NATURAL INNER JOIN T2
FROM T1 NATURAL LEFT OUTER JOIN T2
FROM T1 NATURAL RIGHT OUTER JOIN T2
FROM T1 NATURAL FULL OUTER JOIN T2

FROM (SELECT * FROM T1) DT1 CROSS JOIN T2, T3
FROM (SELECT * FROM T1) DT1, T2, T3
```

Above are some examples of joined tables and complex derived tables. Notice how the AS clause renames or names a derived table and how the optional comma-separated list of column names that follows renames the columns. The last two FROM clauses produce the same derived table from T1, T2, and T3. The AS keyword was omitted in naming the subquery as DT1. The keywords OUTER and INNER are noise that can be omitted also.

2.1.2. WHERE clause

The syntax of the WHERE clause is

```
WHERE search_condition
```

where *search_condition* is any value expression as defined in Section 1.3 that returns a value of type `boolean`.

After the processing of the FROM clause is done, each row of the derived table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (that is, if the result is false or NULL) it is discarded. The search condition typically references at least some column in the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

> **Note:** Before the implementation of the JOIN syntax, it was necessary to put the join condition of an inner join in the WHERE 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 products. For outer joins there is no choice in any case: they must be done in the FROM clause. An outer join's ON/USING clause is *not* equivalent to a WHERE condition, because it determines the addition of rows (for unmatched input rows) as well as the removal of rows from the final result.

```
FROM FDT WHERE
    C1 > 5

FROM FDT WHERE
    C1 IN (1, 2, 3)
FROM FDT WHERE
    C1 IN (SELECT C1 FROM T2)
FROM FDT WHERE
    C1 IN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10)

FROM FDT WHERE
    C1 BETWEEN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10) AND 100

FROM FDT WHERE
    EXISTS (SELECT C1 FROM T2 WHERE C2 > FDT.C1)
```

In the examples above, 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 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. Qualifying the column name adds clarity even when it is not needed. This shows how the column naming scope of an outer query extends into its inner queries.

2.1.3. GROUP BY and HAVING clauses

After passing the WHERE filter, the derived input table may 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 rows in a table that share the same values in all the columns listed. The order in which the columns are listed does not matter (as opposed to an ORDER BY clause). The purpose is to reduce each group of rows sharing common values into one group row that is representative of all rows in the group. This is done to eliminate redundancy in the output and/or obtain aggregates that apply to these groups.

Once a table is grouped, columns that are not used in the grouping cannot be referenced except in aggregate expressions, since a specific value in those columns is ambiguous - which row in the group should it come from? The grouped-by columns can be referenced in select list column expressions since they have a known constant value per group. Aggregate functions on the ungrouped columns provide values that span the rows of a group, not of the whole table. For instance, a `sum(sales)` on a table grouped by product code gives the total sales for each product, not the total sales on all products. Aggregates computed on the ungrouped columns are representative of the group, whereas individual values of an ungrouped column are not.

Example:

```
SELECT pid, p.name, (sum(s.units) * p.price) AS sales
  FROM products p LEFT JOIN sales s USING ( pid )
  GROUP BY pid, p.name, p.price;
```

In this example, the columns pid, p.name, and p.price must be in the GROUP BY clause since they are referenced in the query select list. 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 group of sales of a product. For each product, a summary row is returned about all sales of the product.

In strict SQL, GROUP BY can only group by columns of the source table but Postgres extends this to also allow GROUP BY to group by select columns in the query select list. Grouping by value expressions instead of simple column names is also allowed.

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING
boolean_expression
```

If a table has been grouped using a GROUP BY clause, but then only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from a grouped table. Postgres allows a HAVING clause to be used without a GROUP BY, in which case it acts like another WHERE clause, but the point in using HAVING that way is not clear. A good rule of thumb is that a HAVING condition should refer to the results of aggregate functions. A restriction that does not involve an aggregate is more efficiently expressed in the WHERE clause.

Example:

```
SELECT pid    AS "Products",
       p.name AS "Over 5000",
       (sum(s.units) * (p.price - p.cost)) AS "Past Month Profit"
  FROM products p LEFT JOIN sales s USING ( pid )
  WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
  GROUP BY pid, 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, while the HAVING clause restricts the output to groups with total gross sales over 5000.

2.2. 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. 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 1.3). 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 2.1.1.3. 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.b, tbl1.c FROM ...
```

(see also Section 2.1.2).

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 retrieved 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 could be constant arithmetic expressions as well, for instance.

2.2.1. Column Labels

The entries in the select list can be assigned names for further processing. The further processing in this case is an optional sort specification and the client application (e.g., column headers for display). For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified via AS, the system assigns a default 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.

Note: The naming of output columns here is different from that done in the FROM clause (see Section 2.1.1.3). This pipeline will in fact allow you to rename the same column twice, but the name chosen in the select list is the one that will be passed on.

2.2.2. DISTINCT

After the select list has been processed, the result table may optionally be subject to the elimination of duplicates. The DISTINCT key word is written directly after the SELECT to enable this:

```
SELECT DISTINCT select_list ...
```

(Instead of DISTINCT the word ALL can be used to select the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. NULLs 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 subselects in FROM the construct can be avoided, but it is very often the most convenient alternative.

2.3. Combining Queries

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
```

query1 and *query2* are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 UNION query2 UNION query3
```

which really says

```
(query1 UNION query2) UNION query3
```

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 all duplicate rows, in the sense of DISTINCT, unless ALL is specified.

INTERSECT returns all rows that are both in the result of *query1* and in the result of *query2*. Duplicate rows are eliminated unless ALL is specified.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. Again, duplicates are eliminated unless ALL is specified.

In order to calculate the union, intersection, or difference of two queries, the two queries must be union compatible, which means that they both return the same number of columns, and that the corresponding columns have compatible data types, as described in Section 5.5.

2.4. Sorting Rows

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 random 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 column1 [ASC | DESC] [,
column2 [ASC | DESC] ...]
```

column1, etc., refer to select list columns. These can be either the output name of a column (see Section 2.2.1) or the number of a column. Some examples:

```
SELECT a, b FROM table1 ORDER BY a;
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, sum(b) FROM table1 GROUP BY a ORDER BY 1;
```

As an extension to the SQL standard, Postgres also allows ordering by arbitrary expressions:

```
SELECT a, b FROM table1 ORDER BY a + b;
```

References to column names in the FROM clause that are renamed in the select list are also allowed:

```
SELECT a AS b FROM table1 ORDER BY a;
```

But these extensions do not work in queries involving UNION, INTERSECT, or EXCEPT, and are not portable to other DBMSes.

Each column specification may be followed by an optional ASC or DESC to set the sort direction. ASC is default. Ascending order puts smaller values first, where smaller is defined in terms of the < operator. Similarly, descending order is determined with the > operator.

If more than one sort column is specified, the later entries are used to sort rows that are equal under the order imposed by the earlier sort specifications.

2.5. LIMIT and OFFSET

```
SELECT select_list FROM table_expression [ORDER BY sort_spec] [LIMIT {
number | ALL }] [OFFSET number]
```

LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query. If a limit count is given, no more than that many rows will be returned. If an offset is given, that many rows will be skipped before starting to return rows.

When using LIMIT, it is a good idea 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 may 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 a query plan, 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.

Chapter 3. Data Types

Postgres has a rich set of native data types available to users. Users may add new types to Postgres using the **CREATE TYPE** command.

Table 3-1 shows all general-purpose data types available to users. Most of the alternative names listed in the Aliases column are the names used internally by Postgres for historical reasons. In addition, some internally used or deprecated types are available, but they are not documented here. Many of the built-in types have obvious external formats. However, several types are either unique to Postgres, such as open and closed paths, or have several possibilities for formats, such as the date and time types.

Table 3-1. Data Types

Type Name	Aliases	Description
`bigint`	`int8`	signed eight-byte integer
`bit`		fixed-length bit string
`bit varying(`*n*`)`	`varbit(`*n*`)`	variable-length bit string
`boolean`	`bool`	logical Boolean (true/false)
`box`		rectangular box in 2D plane
`character(`*n*`)`	`char(`*n*`)`	fixed-length character string
`character varying(`*n*`)`	`varchar(`*n*`)`	variable-length character string
`cidr`		IP network address
`circle`		circle in 2D plane
`date`		calendar date (year, month, day)
`double precision`	`float8`	double precision floating-point number
`inet`		IP host address
`integer`	`int`, `int4`	signed four-byte integer
`interval`		general-use time span
`line`		infinite line in 2D plane
`lseg`		line segment in 2D plane
`macaddr`		MAC address
`money`		US-style currency
`numeric(`*p*`, `*s*`)`	`decimal(`*p*`, `*s*`)`	exact numeric with selectable precision
`oid`		object identifier
`path`		open and closed geometric path in 2D plane
`point`		geometric point in 2D plane

Type Name	Aliases	Description
`polygon`		closed geometric path in 2D plane
`real`	`float4`	single precision floating-point number
`smallint`	`int2`	signed two-byte integer
`serial`		autoincrementing four-byte integer
`text`		variable-length character string
`time [ without time zone ]`		time of day
`time with time zone`		time of day, including time zone
`timestamp [ with time zone ]`		date and time

> **Compatibility:** The following types (or spellings thereof) are specified by SQL: `bit`, `bit varying`, `boolean`, `char`, `character`, `character varying`, `varchar`, `date`, `double precision`, `integer`, `interval`, `numeric`, `decimal`, `real`, `smallint`, `time`, `timestamp` (both with or without time zone).

Most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking. Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

3.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating point numbers and fixed-precision decimals.

Table 3-2. Numeric Types

Type Name	Storage	Description	Range
smallint	2 bytes	Fixed-precision	-32768 to +32767
integer	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
bigint	8 bytes	Very large range fixed-precision	about 18 decimal places
decimal	variable	User-specified precision	no limit
numeric	variable	User-specified precision	no limit
real	4 bytes	Variable-precision	6 decimal places
double precision	8 bytes	Variable-precision	15 decimal places

Type Name	Storage	Description	Range
serial	4 bytes	Identifier or cross-reference	0 to +2147483647

The syntax of constants for the numeric types is described in Section 1.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 4 for more information.

The `bigint` type may not be available on all platforms since it relies on compiler support for eight-byte integers.

3.1.1. The Serial Type

The `serial` type is a special-case type constructed by Postgres from other existing components. It is typically used to create unique identifiers for table entries. In the current implementation, specifying

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

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
    (colname integer DEFAULT nextval('tablename_colname_seq');
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

> **Caution**
>
> The implicit sequence created for the `serial` type will *not* be automatically removed when the table is dropped.

Implicit sequences supporting the `serial` are not automatically dropped when a table containing a serial type is dropped. So, the following commands executed in order will likely fail:

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

The sequence will remain in the database until explicitly dropped using **DROP SEQUENCE**.

3.2. Monetary Type

Deprecated: The `money` type is deprecated. Use `numeric` or `decimal` instead, in combination with the `to_char` function. The money type may become a locale-aware layer over the `numeric` type in a future release.

The `money` type stores U.S.-style currency with fixed decimal point representation. If Postgres is compiled with locale support then the `money` type uses locale-specific output formatting.

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 in the latter form.

Table 3-3. Monetary Types

Type Name	Storage	Description	Range
money	4 bytes	Fixed-precision	-21474836.48 to +21474836.47

3.3. Character Types

SQL defines two primary character types: `character` and `character varying`. Postgres supports these types, in addition to the more general `text` type, which unlike `character varying` does not require an explicit declared upper limit on the size of the field.

Refer to Section 1.1.2.1 for information about the syntax of string literals, and to Chapter 4 for information about available operators and functions.

Table 3-4. Character Types

Type Name	Storage	Recommendation	Description
character(n), char(n)	(4+n) bytes	SQL-compatible	Fixed-length blank padded
character varying(n), varchar(n)	(4+n) bytes	SQL-compatible	Variable-length with limit
text	(4+n) bytes	Most flexible	Variable unlimited length

> **Note:** Although the type `text` is not SQL-compliant, many other RDBMS packages have it as well.

There are two other fixed-length character types in Postgres. The `name` type exists *only* for storage of internal catalog names and is not intended for use by the general user. Its length is currently defined as 32 bytes (31 characters plus terminator) but should be referenced using the macro NAMEDATALEN. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length may 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. It is internally used in the system catalogs as a poor-man's enumeration type.

Table 3-5. Specialty Character Type

Type Name	Storage	Description
"char"	1 byte	Single character internal type
name	32 bytes	Thirty-one character internal type

3.4. Date/Time Types

Postgres supports the full set of SQL date and time types.

Table 3-6. Date/Time Types

Type	Description	Storage	Earliest	Latest	Resolution
`timestamp`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`timestamp [with time zone]`	date and time with time zone	8 bytes	1903 AD	2037 AD	1 microsecond / 14 digits
`interval`	for time intervals	12 bytes	-178000000 years	178000000 years	1 microsecond
`date`	dates only	4 bytes	4713 BC	32767 AD	1 day
`time [without time zone]`	times of day only	4 bytes	00:00:00.00	23:59:59.99	1 microsecond
`time with time zone`	times of day only	4 bytes	00:00:00.00+12	23:59:59.99-12	1 microsecond

Note: To ensure compatibility to earlier versions of Postgres we also continue to provide `datetime` (equivalent to `timestamp`) and `timespan` (equivalent to `interval`), however support for these is now restricted to having an implicit translation to `timestamp` and `interval`. The types `abstime` and `reltime` are lower precision types which are used internally. You are discouraged from using any of these types in new applications and are encouraged to move any old ones over when appropriate. Any or all of these internal types might disappear in a future release.

3.4.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO-8601, SQL-compatible, traditional Postgres, and others. The ordering of month and day in date input can be ambiguous, therefore a setting exists to specify how it should be interpreted in ambiguous cases.

The command `SET DateStyle TO 'US'` or `SET DateStyle TO 'NonEuropean'` specifies the variant "month before day", the command `SET DateStyle TO 'European'` sets the variant "day before month". The `ISO` style is the default but this default can be changed at compile time or at run time.

See Appendix A for the exact parsing rules of date/time input and for the recognized time zones.

Remember that any date or time input needs to be enclosed into single quotes, like text strings. Refer to Section 1.1.2.5 for more information. SQL requires the following syntax

```
type 'value'
```

but Postgres is more flexible.

3.4.1.1. date

The following are possible inputs for the `date` type.

Table 3-7. Date Input

Example	Description
January 8, 1999	Unambiguous
1999-01-08	ISO-8601 format, preferred
1/8/1999	US; read as August 1 in European mode
8/1/1999	European; read as August 1 in US mode
1/18/1999	US; read as January 18 in any mode
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
1999.008	Year and day of year
99008	Year and day of year
January 8, 99 BC	Year 99 before the Common Era

Table 3-8. Month Abbreviations

Month	Abbreviations
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun

Month	Abbreviations
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Note: The month `May` has no explicit abbreviation, for obvious reasons.

Table 3-9. Day of the Week Abbreviations

Day	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

3.4.1.2. time [without time zone]

Per SQL99, this type can be referenced as `time` and as `time without time zone`.

The following are valid `time` inputs.

Table 3-10. 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
z	Same as 00:00:00
zulu	Same as 00:00:00

Example	Description
allballs	Same as 00:00:00

3.4.1.3. time with time zone

This type is defined by SQL92, but the definition exhibits fundamental deficiencies that render the type nearly useless. In most cases, a combination of `date`, `time`, and `timestamp` should provide a complete range of date/time functionality required by any application.

`time with time zone` accepts all input also legal for the `time` type, appended with a legal time zone, as follows:

Table 3-11. Time With Time Zone Input

Example	Description
04:05:06.789-8	ISO-8601
04:05:06-08:00	ISO-8601
04:05-08:00	ISO-8601
040506-08	ISO-8601

Refer to Table 3-12 for more examples of time zones.

3.4.1.4. timestamp

Valid input for the `timestamp` type consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06 -8:00
```

is a valid `timestamp` value that is ISO-compliant. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

Table 3-12. Time Zone Input

Time Zone	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

3.4.1.5. interval

`interval`s can be specified with the following syntax:

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Direction]
```

where: `Quantity` is ..., `-1`, `0`, `1`, `2`, ...; `Unit` is `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; `Direction` can be `ago` or empty.

3.4.1.6. Special values

The following SQL-compatible functions can be used as date or time input for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`.

Postgres also supports several special constants for convenience.

Table 3-13. Special Date/Time Constants

Constant	Description
current	Current transaction time, deferred
epoch	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	Later than other valid times
-infinity	Earlier than other valid times
invalid	Illegal entry
now	Current transaction time
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday

`'now'` is resolved when the value is inserted, `'current'` is resolved every time the value is retrieved. So you probably want to use `'now'` in most applications. (Of course you *really* want to use `CURRENT_TIMESTAMP`, which is equivalent to `'now'`.)

3.4.2. Date/Time Output

Output formats can be set to one of the four styles ISO-8601, SQL (Ingres), traditional Postgres, and German, using the **SET DateStyle**. The default is the ISO format.

Table 3-14. Date/Time Output Styles

Style Specification	Description	Example
'ISO'	ISO-8601 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

The output of the `date` and `time` styles is of course only the date or time part in accordance with the above examples.

The SQL style has European and non-European (US) variants, which determines whether month follows day or vice versa. (See also above at Date/Time Input, how this setting affects interpretation of input values.)

Table 3-15. Date Order Conventions

Style Specification	Description	Example
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

`interval` output looks like the input format, except that units like `week` or `century` are converted to years and days. In ISO mode the output looks like

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

There are several ways to affect the appearance of date/time types:

- The PGDATESTYLE environment variable used by the backend directly on postmaster start-up.
- The PGDATESTYLE environment variable used by the frontend libpq on session start-up.
- **SET DATESTYLE** SQL command.

3.4.3. Time Zones

Postgres endeavors to be compatible with SQL92 definitions for typical usage. However, the SQL92 standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type does not have an associated time zone, the `time` type can or does. Time zones in the real world can have no meaning unless associated with a date as well as a time since the offset may vary through the year with daylight savings time boundaries.

The default time zone is specified as a constant integer offset from GMT/UTC. It is not possible to adapt to daylight savings 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 recommend *not* using the SQL92 type TIME WITH TIME ZONE (though it is supported by Postgres for legacy applications and for compatibility with other RDBMS implementations). Postgres assumes local time for any type containing only date or time. Further, time zone support is derived from the underlying operating system time zone capabilities, and hence can handle daylight savings time and other expected behavior.

Postgres obtains time zone support from the underlying operating system for dates between 1902 and 2038 (near the typical date limits for Unix-style systems). Outside of this range, all dates are assumed to be specified and used in Universal Coordinated Time (UTC).

All dates and times are stored internally in UTC, traditionally known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to affect the time zone behavior:

The TZ environment variable is used by the backend directly on postmaster start-up as the default time zone.

The PGTZ environment variable set at the client used by libpq to send time zone information to the backend upon connection.

The SQL command **SET TIME ZONE** sets the time zone for the session.

The SQL92 qualifier on

```
timestamp AT TIME ZONE 'zone'
```

where `zone` can be specified as a text time zone (e.g. `'PST'`) or as an interval (e.g. `INTERVAL '-08:00'`).

Note: If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

Note: If the compiler option USE_AUSTRALIAN_RULES is set then EST refers to Australia Eastern Standard Time, which has an offset of +10:00 hours from UTC.

3.4.4. Internals

Postgres uses Julian dates for all date/time calculations. They have the nice property of correctly predicting/calculating any date more recent than 4713BC to far into the future, using the assumption that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistent enough to warrant coding into a date/time handler.

3.5. Boolean Type

Postgres provides the SQL99 type `boolean`. `boolean` can have one of only two states: true or false . A third state, unknown , is represented by the SQL NULL state.

Valid literal values for the true state are:

```
TRUE
't'
'true'
'y'
'yes'
'1'
```

For the false state, the following values can be used:

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

Using the key words `TRUE` and `FALSE` is preferred (and SQL-compliant).

Example 3-1. 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
```

Example 3-1 shows that `boolean` values are output using the letters `t` and `f`.

> **Tip:** Values of the `boolean` type cannot be cast directly to other types (e.g., `CAST (`*`boolval`*` AS integer)` does not work). This can be accomplished using the `CASE` expression: `CASE WHEN `*`boolval`*` THEN 'value if true' ELSE 'value if false' END`. See also Section 4.10.

`boolean` uses 1 byte of storage.

3.6. Geometric Types

Geometric types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Table 3-16. Geometric Types

Geometric Type	Storage	Representation	Description
point	16 bytes	(x,y)	Point in space
line	32 bytes	((x1,y1),(x2,y2))	Infinite line
lseg	32 bytes	((x1,y1),(x2,y2))	Finite line segment
box	32 bytes	((x1,y1),(x2,y2))	Rectangular box
path	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
path	4+32n bytes	[(x1,y1),...]	Open path
polygon	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
circle	24 bytes	<(x,y),r>	Circle (center and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

3.6.1. Point

Points are the fundamental two-dimensional building block for geometric types.

`point` is specified using the following syntax:

```
( x , y )
  x , y
```

where the arguments are

x

The x-axis coordinate as a floating point number.

y

The y-axis coordinate as a floating point number.

3.6.2. Line Segment

Line segments (`lseg`) are represented by pairs of points.

`lseg` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

($x1$,$y1$)
($x2$,$y2$)

The end points of the line segment.

3.6.3. Box

Boxes are represented by pairs of points that are opposite corners of the box.

`box` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

($x1$,$y1$)
($x2$,$y2$)

Opposite corners of the box.

Boxes are output using the first syntax. The corners are reordered on input to store the upper right corner, then the lower left corner. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored.

3.6.4. Path

Paths are represented by connected sets of points. Paths can be "open", where the first and last points in the set are not connected, and "closed", where the first and last point are connected. Functions `popen(p)` and `pclose(p)` are supplied to force a path to be open or closed, and functions `isopen(p)` and `isclosed(p)` are supplied to test for either type in a query.

`path` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(x,y)

End points of the line segments comprising the path. A leading square bracket ("[") indicates an open path, while a leading parenthesis ("(") indicates a closed path.

Paths are output using the first syntax.

3.6.5. Polygon

Polygons are represented by sets of points. Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

`polygon` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

3.6.6. Circle

Circles are represented by a center point and a radius.

`circle` is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where the arguments are

(*x*,*y*)

Center of the circle.

r

Radius of the circle.

Circles are output using the first syntax.

3.7. Network Address Data Types

Postgres offers data types to store IP and MAC addresses. It is preferable to use these types over plain text types, because these types offer input error checking and several specialized operators and functions.

Table 3-17. Network Address Data Types

Name	Storage	Description	Range
cidr	12 bytes	IP networks	valid IPv4 networks
inet	12 bytes	IP hosts and networks	valid IPv4 hosts or networks
macaddr	6 bytes	MAC addresses	customary formats

IP v6 is not supported, yet.

3.7.1. `inet`

The `inet` type holds an IP host address, and optionally the identity of the subnet it is in, all in one field. The subnet identity is represented by the number of bits in the network part of the address (the netmask). If the netmask is 32, then the value does not indicate a subnet, only a single host. Note that if you want to accept networks only, you should use the `cidr` type rather than `inet`.

The input format for this type is *`x.x.x.x/y`* where *`x.x.x.x`* is an IP address and *`y`* is the number of bits in the netmask. If the *`/y`* part is left off, then the netmask is 32, and the value represents just a single host. On display, the *`/y`* portion is suppressed if the netmask is 32.

3.7.2. `cidr`

The `cidr` type holds an IP network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying classless networks is *`x.x.x.x/y`* where *`x.x.x.x`* is the network and *`y`* is the number of bits in the netmask. If *`y`* is omitted, it is calculated using assumptions from the older classful numbering system, except that it will be at least large enough to include all of the octets written in the input.

Here are some examples:

Table 3-18. `cidr` Type Input Examples

CIDR Input	CIDR Displayed	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
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

3.7.3. 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.

Tip: If you do not like the output format for inet or cidr values, try the host(), text(), and abbrev() functions.

3.7.4. macaddr

The macaddr type stores MAC addresses, i.e., Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in various customary formats, including '08002b:010203', '08002b-010203', '0800.2b01.0203', '08-00-2b-01-02-03', and '08:00:2b:01:02:03', which would all specify the same address. Upper and lower case is accepted for the digits a through f. Output is always in the latter of the given forms.

The directory contrib/mac in the Postgres source distribution contains tools that can be used to map MAC addresses to hardware manufacturer names.

3.8. 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(*x*) and BIT VARYING(*x*); the *x* specifies the maximum length. BIT type data is automatically padded with 0's on the right to the maximum length, BIT VARYING is of variable length. BIT without length is equivalent to BIT(1), BIT VARYING means unlimited length. Input data that is longer than the allowed length will be truncated. Refer to Section 1.1.2.2 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Chapter 4.

Some examples:

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
SELECT SUBSTRING(b FROM 1 FOR 2) FROM test;
```

Chapter 4. Functions and Operators

Postgres provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in the *Programmer's Guide*. The psql commands **\df** and **\do** can be used to show the list of all actually available functions and operators, respectively.

If you are concerned about portability then take 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. However, many other RDBMS packages provide a lot of the same or similar functions, and some of the ones provided in Postgres have in fact been inspired by other implementations.

4.1. Logical Operators

The usual logical operators are available:

AND
OR
NOT

SQL uses a three-valued Boolean logic where NULL 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

4.2. Comparison Operators

Table 4-1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or `!=`	not equal

> **Note:** The `!=` operator is converted to <> in the parser stage. It is not possible to implement `!=` and <> operators that do different things.

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `boolean`; expressions like `1 < 2 < 3` are not valid (because there is no < operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special BETWEEN construct is available.

```
a BETWEEN x AND y
```

is equivalent to

```
a >= x AND a <= y
```

Similarly,

```
a NOT BETWEEN x AND y
```

is equivalent to

```
a < x OR a > y
```

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not NULL, use the constructs

```
expression IS NULL
expression IS NOT NULL
```

Do *not* use `expression = NULL` because NULL is not equal to NULL. (NULL represents an unknown value, so it is not known whether two unknown values are equal.) Postgres presently converts `x = NULL` clauses to `x IS NULL` to allow some broken client applications (such as Microsoft Access) to work, but this may be discontinued in a future release.

4.3. Mathematical Functions and Operators

Table 4-2. Mathematical Operators

<table>
<tr><th>Name</th><th>Description</th><th>Example</th><th>Result</th></tr>
<tr><td>+</td><td>Addition</td><td>2 + 3</td><td>5</td></tr>
<tr><td>-</td><td>Subtraction</td><td>2 - 3</td><td>-1</td></tr>
<tr><td>*</td><td>Multiplication</td><td>2 * 3</td><td>6</td></tr>
<tr><td>/</td><td>Division (integer division truncates results)</td><td>4 / 2</td><td>2</td></tr>
<tr><td>%</td><td>Modulo (remainder)</td><td>5 % 4</td><td>1</td></tr>
<tr><td>^</td><td>Exponentiation</td><td>2.0 ^ 3.0</td><td>8.0</td></tr>
<tr><td>|/</td><td>Square root</td><td>|/ 25.0</td><td>5.0</td></tr>
<tr><td>||/</td><td>Cube root</td><td>||/ 27.0</td><td>3</td></tr>
<tr><td>!</td><td>Factorial</td><td>5 !</td><td>120</td></tr>
<tr><td>!!</td><td>Factorial (prefix operator)</td><td>!! 5</td><td>120</td></tr>
<tr><td>@</td><td>Absolute value</td><td>@ -5.0</td><td>5.0</td></tr>
<tr><td>&</td><td>Binary AND</td><td>91 & 15</td><td>11</td></tr>
<tr><td>|</td><td>Binary OR</td><td>32 | 3</td><td>35</td></tr>
<tr><td>#</td><td>Binary XOR</td><td>17 # 5</td><td>20</td></tr>
<tr><td>~</td><td>Binary NOT</td><td>~1</td><td>-2</td></tr>
<tr><td><<</td><td>Binary shift left</td><td>1 << 4</td><td>16</td></tr>
<tr><td>>></td><td>Binary shift right</td><td>8 >> 2</td><td>2</td></tr>
</table>

The binary operators are also available for the bit string types `BIT` and `BIT VARYING`.

Table 4-3. Bit String Binary Operators

<table>
<tr><th>Example</th><th>Result</th></tr>
<tr><td>B'10001' & B'01101'</td><td>00001</td></tr>
<tr><td>B'10001' | B'01101'</td><td>11101</td></tr>
<tr><td>B'10001' # B'01101'</td><td>11110</td></tr>
<tr><td>~ B'10001'</td><td>01110</td></tr>
<tr><td>B'10001' << 3</td><td>01000</td></tr>
<tr><td>B'10001' >> 2</td><td>00100</td></tr>
</table>

Bit string arguments to `&`, `|`, and # must be of equal length. When bit shifting, the original length of the string is preserved, as shown here.

Table 4-4. Mathematical Functions

Function	Return Type	Description	Example	Result
abs(*x*)	(same as x)	absolute value	abs(-17.4)	17.4
cbrt(`dp`)	`dp`	cube root	cbrt(27.0)	3.0
ceil(`numeric`)	`numeric`	smallest int not less than	ceil(-42.8)	-42
degrees(`dp`)	`dp`	radians to degrees	degrees(0.5)	28.647...
exp(`dp`)	`dp`	exponential	exp(1.0)	2.7182...
floor(`numeric`)	`numeric`	largest integer not greater than	floor(-42.8)	-43
ln(`dp`)	`dp`	natural logarithm	ln(2.0)	0.693...
log(`dp`)	`dp`	base 10 logarithm	log(100.0)	2.0
log(*b* `numeric`, *x* `numeric`)	`numeric`	logarithm to base *b*	log(2.0, 64.0)	6.0
mod(*y*, *x*)	(same as args)	remainder of *y*/*x*	mod(9,4)	1
pi()	`dp`	Pi constant	pi()	3.141...
pow(*e* `dp`, `n dp`)	`dp`	n raised to *e*	pow(9.0, 3.0)	729.0
radians(`dp`)	`dp`	degrees to radians	radians(45.0)	0.785...
random()	`dp`	a pseudo-random value between 0.0 to 1.0	random()	
round(`dp`)	`dp`	round to nearest integer	round(42.4)	42
round(*v* `numeric`, *s* `integer`)	`numeric`	round to s places	round(42.4382, 2)	42.44
sqrt(`dp`)	`dp`	square root	sqrt(2.0)	1.414...
trunc(`dp`)	`dp`	truncate -> zero	trunc(42.8)	42
trunc(`numeric`, *s* `integer`)	`numeric`	truncate to *s* places	round(42.4382, 2)	42.43

In the table above, "dp" indicates `double precision`. The functions `exp`, `ln`, `log`, `pow`, `round` (1 argument), `sqrt`, and `trunc` (1 argument) are also available for the type `numeric` in place of `double precision`. Functions returning a `numeric` result take `numeric` input arguments, unless otherwise specified. Many of these functions are implemented on top of the host system's C library and behavior in boundary cases could therefore vary depending on the operating system.

Table 4-5. Trigonometric Functions

Function	Description
acos(*x*)	inverse cosine
asin(*x*)	inverse sine
atan(*x*)	inverse tangent
atan2(*x*, *y*)	inverse tangent of *y*/*x*
cos(*x*)	cosine
cot(*x*)	cotangent
sin(*x*)	sine
tan(*x*)	tangent

All trigonometric functions have arguments and return values of type `double precision`.

4.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of all the types `CHARACTER`, `CHARACTER VARYING`, and `TEXT`. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of the automatic padding when using the `CHARACTER` type. Generally the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions also exist natively for bit string types.

SQL defines some string functions with a special syntax where certain keywords rather than commas are used to separate the arguments. Details are in Table 4-6. These functions are also implemented using the regular syntax for function invocation. (See Table 4-7.)

Table 4-6. SQL String Functions and Operators

Function	Returns	Description	Example	Result
string `||` *string*	`text`	string concatenation	'Postgre' || 'SQL'	PostgreSQL
char_length(*string*) or character_length(*string*)	`integer`	length of string	char_length('jose')	4
lower(*string*)	`text`	Convert string to lower case.	lower('TOM')	tom
octet_length(*string*)	`integer`	number of bytes in string	octet_length('jose')	4
position(*substr* in *str*)	`integer`	location of specified substring	position('om' in 'Thomas')	3
substring(*string* [from `integer`] [for `integer`])	`text`	extract substring	substring('Thomas' from 2 for 3)	oma

Function	Returns	Description	Example	Result
trim([leading | trailing | both] [*characters*] from *string*)	text	Removes the longest string containing only the *characters* (a space by default) from the beginning/end/both ends of the *string*.	trim(both 'x' from 'xTomx')	Tom
upper(*string*)	text	Convert string to upper case.	upper('tom')	TOM

Additional string manipulation functions are available and are listed below. Some of them are used internally to implement the SQL string functions listed above.

Table 4-7. Other String Functions

Function	Returns	Description	Example	Result
ascii(text)	integer	Returns the ASCII code of the first character of the argument.	ascii('x')	120
btrim(*string* text, *trim* text)	text	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*.	btrim('xyxtrimyyx', 'xy')	trim
chr(integer)	text	Returns the character with the given ASCII code.	chr(65)	A
initcap(text)	text	Converts first letter of each word (whitespace separated) to upper case.	initcap('hi thomas')	Hi Thomas
lpad(*string* text, *length* integer [, *fill* text])	text	Fills up 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, *trim* text)	text	Removes the longest string containing only characters from *trim* from the start of the string.	ltrim('zzzytrim', 'xyz')	trim
repeat(text, integer)	text	Repeat text a number of times.	repeat('Pg', 4)	PgPgPgPg

Function	Returns	Description	Example	Result
rpad(*string* text, *length* integer [, *fill* text])	text	Fills up 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, *trim* text)	text	Removes the longest string containing only characters from *trim* from the end of the string.	rtrim('trimxxxx','x')	trim
strpos(*string*, *substring*)	text	Locates specified substring. (same as position(*substring* in *string*), but note the reversed argument order)	strpos('high','ig')	2
substr(*string*, *from* [, *count*])	text	Extracts specified substring. (same as substring(*string* from *from* for *count*))	substr('alphabet', 3, 2)	ph
to_ascii(text [, *encoding*])	text	Converts text from multibyte encoding to ASCII.	to_ascii('Karel')	Karel
translate(*string* text, *from* text, *to* text)	text	Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set.	translate('12345', '14', 'ax')	a23x5

The `to_ascii` function supports conversion from LATIN1, LATIN2, WIN1250 (CP1250) only.

4.5. Pattern Matching

There are two separate approaches to pattern matching provided by Postgres: the SQL `LIKE` operator and POSIX-style regular expressions.

> **Tip:** If you have pattern matching needs that go beyond this, or want to make pattern-driven substitutions or translations, consider writing a user-defined function in Perl or Tcl.

4.5.1. Pattern Matching with LIKE

```
string LIKE pattern [ ESCAPE escape-character ]
string NOT LIKE pattern [ ESCAPE escape-character ]
```

Every *pattern* defines a set of strings. The `LIKE` expression returns true if the *string* is contained in the set of strings represented by *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 underscore, 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 string 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 matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore 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 may be selected by using the `ESCAPE` clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in the query. You can avoid this by selecting a different escape character with `ESCAPE`.

The keyword 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 Postgres extension.

The operator `~~` is equivalent to `LIKE`, and `~~*` corresponds to `ILIKE`. There are also `!~~` and `!~~*` operators that represent `NOT LIKE` and `NOT ILIKE`. All of these are also Postgres-specific.

4.5.2. POSIX Regular Expressions

Table 4-8. Regular Expression Match Operators

Operator	Description	Example
`~`	Matches regular expression, case sensitive	'thomas' ~ '.*thomas.*'
`~*`	Matches regular expression, case insensitive	'thomas' ~* '.*Thomas.*'
`!~`	Does not match regular expression, case sensitive	'thomas' !~ '.*Thomas.*'
`!~*`	Does not match regular expression, case insensitive	'thomas' !~* '.*vadim.*'

POSIX regular expressions provide a more powerful means for pattern matching than the `LIKE` function. 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.

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of **egrep**; 1003.2 calls these extended REs) and obsolete REs (roughly those of **ed**; 1003.2 basic REs). Postgres implements the modern form.

A (modern) RE is one or more non-empty *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is one or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single `*`, `+`, `?`, or *bound*. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a sequence of 0 or 1 matches of the atom.

A *bound* is `{` followed by an unsigned decimal integer, possibly followed by `,` possibly followed by another unsigned decimal integer, always followed by `}`. The integers must lie between 0 and RE_DUP_MAX (255) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *`i`* and no comma matches a sequence of exactly *`i`* matches of the atom. An atom followed by a bound containing one integer *`i`* and a comma matches a sequence of *`i`* or more matches of the atom. An atom followed by a bound containing two integers *`i`* and *`j`* matches a sequence of *`i`* through *`j`* (inclusive) matches of the atom.

> **Note:** A repetition operator (`?`, `*`, `+`, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow `^` or `|`.

An *atom* is a regular expression enclosed in `()` (matching a match for the regular expression), an empty set of `()` (matching the null string), a *bracket expression* (see below), `.` (matching any single character), `^` (matching the null string at the beginning of the input string), `$` (matching the null string at the end of the input string), a `\` followed by one of the characters `^.[$()|*+?{\` (matching that character taken as an ordinary character), a `\` followed by any other character (matching that character taken as an ordinary character, as if the `\` had not been present), or a single character with no other significance (matching that character). A `{` followed by a character other than a digit is an ordinary character, not the beginning of a bound. It is illegal to end an RE with `\`.

Note that the backslash (`\`) already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in the query.

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 (but see below) 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, and portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (following a possible `^`). 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 and some combinations using `[` (see next paragraphs), all other special characters, including `\`, lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-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 a single element of the bracket expression's list. A bracket expression containing a multi-character collating element can thus 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`.

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 may not 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. Standard character class names are: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit. These stand for the character classes defined in ctype. A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions [[:<:]] and [[:>:]] match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype) or an 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.

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, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, bb* matches the three middle characters of abbbc, (wee|week)(knights|nights) 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 the null 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, so that (e.g.) [x] becomes [xX] and [^x] becomes [^xX].

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. Bounded repetitions are implemented by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, ((((a{1,100}){1,100}){1,100}){1,100}){1,100} will (eventually) run almost any existing machine out of swap space. [1]

4.6. Formatting Functions

Author: Written by Karel Zak (<zakkr@zf.jcu.cz>) on 2000-01-24

The Postgres 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. 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 4-9. Formatting Functions

Function	Returns	Description	Example
to_char(timestamp, text)	text	convert timestamp to string	to_char(timestamp ’now’, ’HH12:MI:SS’)
to_char(int, text)	text	convert int4/int8 to string	to_char(125, ’999’)
to_char(double precision, text)	text	convert real/double precision to string	to_char(125.8, ’999D9’)
to_char(numeric, text)	text	convert numeric to string	to_char(numeric ’-125.8’, ’999D99S’)
to_date(text, text)	date	convert string to date	to_date(’05 Dec 2000’, ’DD Mon YYYY’)
to_timestamp(text, text)	timestamp	convert string to timestamp	to_timestamp(’05 Dec 2000’, ’DD Mon YYYY’)
to_number(text, text)	numeric	convert string to numeric	to_number(’12,454.8-’, ’99G999D9S’)

In an output template string, there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string template patterns identify the parts of the input data string to be looked at and the values to be found there.

Table 4-10. Template patterns for date/time conversions

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)
SSSS	seconds past midnight (0-86399)
AM or A.M. or PM or P.M.	meridian indicator (upper case)
am or a.m. or pm or p.m.	meridian indicator (lower case)
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)

Pattern	Description
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
BC or B.C. or AD or A.D.	year indicator (upper case)
bc or b.c. or ad or a.d.	year indicator (lower case)
MONTH	full upper case month name (blank-padded to 9 chars)
Month	full mixed case 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)
Mon	abbreviated mixed case month name (3 chars)
mon	abbreviated lower case month name (3 chars)
MM	month number (01-12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full mixed case 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)
Dy	abbreviated mixed case day name (3 chars)
dy	abbreviated lower case day name (3 chars)
DDD	day of year (001-366)
DD	day of month (01-31)
D	day of week (1-7; SUN=1)
W	week of month (1-5): first week start on the first day of the month
WW	week number of year (1-53): starts on the first day of the year
IW	ISO week number in year: first week has the first Thursday of year
CC	century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman Numerals (I-XII; I=January) - upper case
rm	month in Roman Numerals (I-XII; I=January) - lower case
TZ	timezone name - upper case
tz	timezone name - lower case

Certain modifiers may be applied to any template pattern to alter its behavior. For example, `FMMonth` is the `Month` pattern with the `FM` prefix.

Table 4-11. Template pattern modifiers for date/time conversions

Modifier	Description	Example
`FM` prefix	fill mode (suppress padding blanks and zeroes)	FMMonth
`TH` suffix	add upper-case ordinal number suffix	DDTH
`th` suffix	add lower-case ordinal number suffix	DDth
`FX` prefix	FiXed format global option (see below)	FX Month DD Day
`SP` suffix	spell mode (not yet implemented)	DDSP

Usage notes:

`FM` suppresses leading zeroes or trailing blanks that would otherwise be added to make the output of a pattern be fixed-width.

`to_timestamp` and `to_date` skip multiple blank spaces in the input string if the `FX` option is not used. `FX` must be specified as the first item in the template; for example `to_timestamp('2000    JUN','YYYY    MON')` is right, but `to_timestamp('2000    JUN','FXYYYY MON')` returns an error, because `to_timestamp` expects one blank space only.

If a backslash (`\`) is desired in a string constant, a double backslash (`\\`) must be entered; for example `'\\HH\\MI\\SS'`. This is true for any string constant in Postgres.

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 pattern keywords. For example, in `'"Hello Year: "YYYY'`, the `YYYY` will be replaced by year data, but the single `Y` will not be.

If you want to have a double quote in the output you must precede it with a backslash, for example `'\\"YYYY Month\\"'`.

`YYYY` conversion from string to timestamp or date is restricted if you use a year 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 year 20000): `to_date('200001131', 'YYYYMMDD')` will be interpreted as a 4-digit year; better is to use a non-digit separator after the year, like `to_date('20000-1131', 'YYYY-MMDD')` or `to_date('20000Nov31', 'YYYYMonDD')`.

Table 4-12. Template patterns for numeric conversions

Pattern	Description
9	value with the specified number of digits
0	value with leading zeros
. (period)	decimal point
, (comma)	group (thousand) separator
PR	negative value in angle brackets
S	negative value with minus sign (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	convert to ordinal number
V	shift *n* digits (see notes)
EEEE	scientific numbers (not supported yet)

Usage notes:

A sign formatted using 'SG', 'PL' or 'MI' is not an anchor in the number; for example, to_char(-12, 'S9999') produces `'  -12'`, but to_char(-12, 'MI9999') produces `'-  12'`. The Oracle implementation does not allow the use of `MI` ahead of `9`, but rather requires that `9` precede `MI`.

`9` specifies a value with the same number of digits as there are `9`s. If a digit is not available use blank space.

`TH` does not convert values less than zero and does not convert decimal numbers.

`PL`, `SG`, and `TH` are Postgres extensions.

`V` effectively multiplies the input values by `10^`*n*, where *n* is the number of digits following `V`. `to_char` does not support the use of `V` combined with a decimal point. (E.g., `99.9V99` is not allowed.)

Table 4-13. `to_char` Examples

Input	Output
to_char(now(),’Day, DD HH12:MI:SS’)	`'Tuesday , 06 05:39:18'`
to_char(now(),’FMDay, FMDD HH12:MI:SS’)	`'Tuesday, 6 05:39:18'`
to_char(-0.1,’99.99’)	`' -.10'`
to_char(-0.1,’FM9.99’)	`'-.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,’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,’PL999’)	`'+485'`
to_char(485,’SG999’)	`'+485'`
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'`

Input	Output
to_char(12.4,'99V999')	`' 12400'`
to_char(12.45, '99V9')	`' 125'`

4.7. Date/Time Functions

Table 4-14 shows the available functions for date/time value processing. The basic arithmetic operators (+, *, etc.) are also available. For formatting functions, refer to Section 4.6. You should be familiar with the background information on date/time data types (see Section 3.4).

Table 4-14. Date/Time Functions

Name	Returns	Description	Example	Result
age(timestamp)	interval	subtract from today	age(timestamp '1957-06-13')	43 years 8 mons 3 days
age(timestamp, timestamp)	interval	subtract arguments	age('2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
current_date	date	Today's date		
current_time	time	Time of day		
current_timestamp	timestamp	Date and time now		
date_part(text, timestamp)	double precision	Get subfield (equivalent to `extract`)	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Get subfield (equivalent to `extract`)	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Truncate date to specified units	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00+00
extract(identifier from timestamp)	double precision	Get subfield	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(identifier from interval)	double precision	Get subfield from interval value	extract(month from interval '2 years 3 months')	3
isfinite(timestamp)	boolean	Test for finite time stamp (neither invalid nor infinity)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Test for finite interval	isfinite(interval '4 hours')	true

Name	Returns	Description	Example	Result
now()	timestamp	Current date and time (equivalent to `current_timestamp`)		
timeofday()	text	High-precision date and time	timeofday()	Wed Feb 21 17:01:13.000126 2001 EST
timestamp(date)	timestamp	Date to timestamp	timestamp(date '2000-12-25')	2000-12-25 00:00:00
timestamp(date, time)	timestamp	Date and time to timestamp	timestamp(date '1998-02-24',time '23:07')	1998-02-24 23:07:00

4.7.1. `EXTRACT`, `date_part`

`EXTRACT (`*`field`*` FROM `*`source`*`)`

The `extract` function retrieves sub-fields from date/time values, such as year or hour. *`source`* is a value expression that evaluates to type `timestamp` or `interval`. (Expressions of type `date` or `time` will be cast to `timestamp` and can therefore be used as well.) *`field`* is an identifier or string that selects what field to extract from the source value. The `extract` function returns values of type `double precision`. The following are valid values:

century

The year field divided by 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

Note that the result for the century field is simply the year field divided by 100, and not the conventional definition which puts most years in the 1900's in the twentieth century.

day

The day (of the month) field (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16
```

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 (0 - 6; Sunday is 0) (for `timestamp` values only)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

doy

The day of the year (1 - 365/366) (for `timestamp` values only)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

epoch

For `date` and `timestamp` values, the number of seconds since 1970-01-01 00:00:00 (Result may be negative.); for `interval` values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 982352320

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

hour

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

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 year field divided by 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
```

Note that the result for the millennium field is simply the year field divided by 1000, and not the conventional definition which puts years in the 1900's in the second millennium.

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
```

minute

The minutes field (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

month

For `timestamp` values, 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 day is in (for `timestamp` values only)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

second

The seconds field, including fractional parts (0 - 59[2])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

week

From a `timestamp` value, calculate the number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

year

The year field

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

The `extract` function is primarily intended for computational processing. For formatting date/time values for display, see Section 4.6.

The `date_part` function is modeled on the traditional Ingres equivalent to the SQL-function `extract`:

```
date_part('field', source)
```

Note that here the *`field`* value needs to be a string. The valid field values for `date_part` are the same as for `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes')
Result: 4
```

4.7.2. `date_trunc`

The function `date_trunc` is conceptually similar to the `trunc` function for numbers.

```
date_trunc('field', source)
```

`source` is a value expression of type `timestamp` (values of type `date` and `time` are cast automatically). *`field`* selects to which precision to truncate the time stamp value. The return value is of type `timestamp` with all fields that are less than the selected one set to zero (or one, for day and month).

Valid values for *`field`* are:

microseconds
milliseconds
second
minute
hour
day
month
year
decade
century
millennium

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00+00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00+00
```

4.7.3. Current Date/Time

The following functions are available to obtain the current date and/or time:

```
CURRENT_TIME
CURRENT_DATE
CURRENT_TIMESTAMP
```

Note that because of the requirements of the SQL standard, these functions must not be called with trailing parentheses.

```
SELECT CURRENT_TIME;
19:07:32

SELECT CURRENT_DATE;
2001-02-17
```

```
SELECT CURRENT_TIMESTAMP;
2001-02-17 19:07:32-05
```

The function `now()` is the traditional Postgres equivalent to `CURRENT_TIMESTAMP`.

There is also `timeofday()`, which returns current time to higher precision than the `CURRENT_TIMESTAMP` family does:

```
SELECT timeofday();
 Sat Feb 17 19:07:32.000126 2001 EST
```

`timeofday()` uses the operating system call `gettimeofday(2)`, which may have resolution as good as microseconds (depending on your platform); the other functions rely on `time(2)` which is restricted to one-second resolution. For historical reasons, `timeofday()` returns its result as a text string rather than a timestamp value.

It is quite important to realize that `CURRENT_TIMESTAMP` and related functions all return the time as of the start of the current transaction; their values do not increment while a transaction is running. But `timeofday()` returns the actual current time.

All the date/time datatypes also accept the special literal value `now` to specify the current date and time. Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

Note: You do not want to use the third form when specifying a DEFAULT value while creating a table. 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.

4.8. 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.

Table 4-15. Geometric Operators

Operator	Description	Usage
+	Translation	box '((0,0),(1,1))' + point '(2.0,0)'
-	Translation	box '((0,0),(1,1))' - point '(2.0,0)'
*	Scaling/rotation	box '((0,0),(1,1))' * point '(2.0,0)'
/	Scaling/rotation	box '((0,0),(2,2))' / point '(2.0,0)'
#	Intersection	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Number of points in polygon	# '((1,0),(0,1),(-1,0))'
##	Point of closest proximity	point '(0,0)' ## lseg '((2,0),(0,2))'

Operator	Description	Usage
&&	Overlaps?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	Overlaps to left?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Overlaps to right?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<->	Distance between	circle '((0,0),1)' <-> circle '((5,0),1)'
<<	Left of?	circle '((0,0),1)' << circle '((5,0),1)'
<^	Is below?	circle '((0,0),1)' <^ circle '((0,5),1)'
>>	Is right of?	circle '((5,0),1)' >> circle '((0,0),1)'
>^	Is above?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Intersects or overlaps	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))';
?-	Is horizontal?	point '(1,0)' ?- point '(0,0)'
?-|	Is perpendicular?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
@-@	Length or circumference	@-@ path '((0,0),(1,0))'
?|	Is vertical?	point '(0,1)' ?| point '(0,0)'
?||	Is parallel?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@	Contained or on	point '(1,1)' @ circle '((0,0),2)'
@@	Center of	@@ circle '((0,0),10)'
~=	Same as	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Table 4-16. Geometric Functions

Function	Returns	Description	Example
area(object)	double precision	area of item	area(box '((0,0),(1,1))')
box(box, box)	box	intersection box	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(object)	point	center of item	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	length of item	length(path '((-1,0),(1,0))')
pclose(path)	path	convert path to closed	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	int4	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open path	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')

Function	Returns	Description	Example
width(box)	double precision	horizontal size	width(box '((0,0),(1,1))')

Table 4-17. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	circle to box	box(circle '((0,0),2.0)')
box(point, point)	box	points to box	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polygon to box	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	to circle	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	point to circle	circle(point '(0,0)', 2.0)
lseg(box)	lseg	box diagonal to lseg	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	points to lseg	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	polygon to path	path(polygon '((0,0),(1,1),(2,0))')
point(circle)	point	center	point(circle '((0,0),2.0)')
point(lseg, lseg)	point	intersection	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	center	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	12 point polygon	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	12-point polygon	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	*npts* polygon	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	path to polygon	polygon(path '((0,0),(1,1),(2,0))')

4.9. Network Address Type Functions

Table 4-18. `cidr` and `inet` Operators

Operator	Description	Usage
<	Less than	inet '192.168.1.5' < inet '192.168.1.6'
<=	Less than or equal	inet '192.168.1.5' <= inet '192.168.1.5'
=	Equals	inet '192.168.1.5' = inet '192.168.1.5'
>=	Greater or equal	inet '192.168.1.5' >= inet '192.168.1.5'
>	Greater	inet '192.168.1.5' > inet '192.168.1.4'
<>	Not equal	inet '192.168.1.5' <> inet '192.168.1.4'
<<	is contained within	inet '192.168.1.5' << inet '192.168.1/24'
<<=	is contained within or equals	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	contains	inet'192.168.1/24' >> inet '192.168.1.5'
>>=	contains or equals	inet '192.168.1/24' >>= inet '192.168.1/24'

All of the operators for `inet` can be applied to `cidr` values as well. The operators `<<`, `<<=`, `>>`, `>>=` test for subnet inclusion: they consider only the network parts of the two addresses, ignoring any host part, and determine whether one network part is identical to or a subnet of the other.

Table 4-19. `cidr` and `inet` Functions

Function	Returns	Description	Example	Result
broadcast(inet)	inet	broadcast address for network	broadcast('192.168.1.5/24')	192.168.1.255/24
host(inet)	text	extract IP address as text	host('192.168.1.5/24')	192.168.1.5
masklen(inet)	integer	extract netmask length	masklen('192.168.1.5/24')	24
netmask(inet)	inet	construct netmask for network	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	extract network part of address	network('192.168.1.5/24')	192.168.1.0/24
text(inet)	text	extract IP address and masklen as text	text(inet '192.168.1.5')	192.168.1.5/32
abbrev(inet)	text	extract abbreviated display as text	abbrev(cidr '10.1.0.0/16')	10.1/16

All of the functions for `inet` can be applied to `cidr` values as well. The `host`(), `text`(), and `abbrev`() functions are primarily intended to offer alternative display formats.

Table 4-20. `macaddr` Functions

Function	Returns	Description	Example	Result
trunc(macaddr)	macaddr	set last 3 bytes to zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

The function `trunc(macaddr)` returns a MAC address with the last 3 bytes set to 0. This can be used to associate the remaining prefix with a manufacturer. The directory `contrib/mac` in the source distribution contains some utilities to create and maintain such an association table.

The `macaddr` type also supports the standard relational operators (`>`, `<=`, etc.) for lexicographical ordering.

4.10. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in Postgres.

> **Tip:** If your needs go beyond the capabilities of these conditional expressions you might want to consider writing a stored procedure in a more expressive programming language.

CASE

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other languages. CASE clauses can be used wherever an expression is valid. *condition* is an expression that returns a `boolean` result. If the result is true then the value of the CASE expression is *result*. If the result is false any subsequent WHEN clauses are searched in the same manner. If no WHEN *condition* is true then the value of the case expression is the *result* in the ELSE clause. If the ELSE clause is omitted and no condition matches, 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 coercible to a single output type. See Section 5.5 for more detail.

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

This simple CASE expression is a specialized variant of the general form above. The *expression* is computed and compared to all the *value*s in the WHEN clauses until one is found that is equal. If no match is found, the *result* in the ELSE clause (or NULL) 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
```

COALESCE

`COALESCE(value[, ...])`

The `COALESCE` function returns the first of its arguments that is not NULL. This is often useful to substitute a default value for NULL values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

NULLIF

`NULLIF(value1, value2)`

The `NULLIF` function returns NULL if and only if *value1* and *value2* are equal. Otherwise it returns *value1*. This can be used to perform the inverse operation of the `COALESCE` example given above:

```
SELECT NULLIF(value, '(none)') ...
```

> **Tip:** `COALESCE` and `NULLIF` are just shorthand for CASE expressions. They are actually converted into CASE expressions at a very early stage of processing, and subsequent processing thinks it is dealing with CASE. Thus an incorrect `COALESCE` or `NULLIF` usage may draw an error message that refers to CASE.

4.11. Miscellaneous Functions

Table 4-21. Miscellaneous Functions

Name	Return Type	Description
current_user	name	user name of current execution context
session_user	name	session user name
user	name	equivalent to `current_user`

The `session_user` is the user that initiated a database connection and is fixed for the duration of that connection. The `current_user` is the user identifier that is applicable for permission checking. Currently it is always equal to the session user, but in the future there might be setuid functions and other facilities to allow the current user to change temporarily. In Unix parlance, the session user is the real user and the current user is the effective user .

Note that these functions have special syntactic status in SQL; they must be called without trailing parentheses.

> **Deprecated:** The function `getpgusername()` is an obsolete equivalent of `current_user`.

4.12. Aggregate Functions

Author: Written by Isaac Wilcox `<isaac@azartmedia.com>` on 2000-06-16

Aggregate functions compute a single result value from a set of input values. The special syntax considerations for aggregate functions are explained in Section 1.3.4. Consult the *PostgreSQL Tutorial* for additional introductory information.

Table 4-22. Aggregate Functions

Function	Description	Notes
AVG(*expr*)	the average (arithmetic mean) of all input values	Finding the average value is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`, `interval`. The result is of type `numeric` for any integer type input, `double precision` for floating point input, otherwise the same as the input data type.
COUNT(*)	number of input values	The return value is of type `integer`.
COUNT(*expr*)	Counts the input values for which the value of *expression* is not NULL.	
MAX(*expr*)	the maximum value of *expression* across all input values	Available for all numeric, string, and date/time types. The result has the same type as the input expression.
MIN(*expr*)	the minimum value of *expression* across all input values	Available for all numeric, string, and date/time types. The result has the same type as the input expression.
STDDEV(*expr*)	the sample standard deviation of the input values	Finding the standard deviation is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`. The result is of type `double precision` for floating point input, otherwise `numeric`.
SUM(*expr*)	sum of *expression* across all input values	Summation is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`, `interval`. The result is of type `numeric` for any integer type input, `double precision` for floating point input, otherwise the same as the input data type.
VARIANCE(*expr*)	the sample variance of the input values	The variance is the square of the standard deviation. The supported data types are the same.

It should be noted that except for `COUNT`, these functions return NULL when no rows are selected. In particular, `SUM` of no rows returns NULL, not zero as one might expect.

Notes

1. This was written in 1994, mind you. The numbers have probably changed, but the problem persists.
2. 60 if leap seconds are implemented by the operating system

Chapter 5. Type Conversion

SQL queries can, intentionally or not, require mixing of different data types in the same expression. Postgres has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by Postgres can affect the results of a query. When necessary, these results can be tailored by a user or programmer using *explicit* type coercion.

This chapter introduces the Postgres type conversion mechanisms and conventions. Refer to the relevant sections in the User's Guide and Programmer's Guide for more information on specific data types and allowed functions and operators.

The Programmer's Guide has more details on the exact algorithms used for implicit type conversion and coercion.

5.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. Postgres has an extensible type system that is much more general and flexible than other RDBMS implementations. Hence, most type conversion behavior in Postgres should be governed by general rules rather than by ad-hoc heuristics to allow mixed-type expressions to be meaningful, even with user-defined types.

The Postgres scanner/parser decodes lexical elements into only five fundamental categories: integers, floats, strings, names, and keywords. Most extended types are first tokenized into strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in Postgres to start the parser down the correct path. For example, the query

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

has two strings, of type `text` and `point`. If a type is not specified for a string, 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 Postgres parser:

Operators

Postgres allows expressions with left- and right-unary (one argument) operators, as well as binary (two argument) operators.

Function calls

Much of the Postgres type system is built around a rich set of functions. Function calls have one or more arguments which, for any specific query, must be matched to the functions available in the system catalog. Since Postgres 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.

Query targets

SQL INSERT and UPDATE statements place the results of expressions into a table. The expressions in the query must be matched up with, and perhaps converted to, the types of the target columns.

UNION and CASE constructs

Since all select results from a UNION 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 coerced to a common type so that the CASE expression as a whole has a known output type.

Many of the general type conversion rules use simple conventions built on the Postgres function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL92 standard native types such as `smallint`, `integer`, and `float`.

The Postgres parser uses the convention that all type conversion functions take a single argument of the source type and are named with the same name as the target type. Any function meeting these criteria is considered to be a valid conversion function, and may be used by the parser as such. This simple assumption gives the parser the power to explore type conversion possibilities without hardcoding, allowing extended user-defined types to use these same features transparently.

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are several basic *type categories* defined: boolean, numeric, string, bitstring, datetime, timespan, geometric, network, and user-defined. Each category, with the exception of user-defined, has a *preferred type* which is preferentially selected when there is ambiguity. In the user-defined category, each type is its own preferred type. Ambiguous expressions (those with multiple candidate parsing solutions) can often be resolved when there are multiple possible built-in types, but they will raise an error when there are multiple choices for user-defined types.

5.1.1. Guidelines

All type conversion rules are designed with several principles in mind:

Implicit conversions should never have surprising or unpredictable outcomes.

User-defined types, of which the parser has no a-priori knowledge, should be "higher" in the type hierarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).

User-defined types are not related. Currently, Postgres does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions in the catalog.

There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion functions into the query.

Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines an explicit function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

5.2. Operators

Operator Type Resolution

1. Check for an exact match in the pg_operator system catalog.
    a. If one argument of a binary operator is `unknown` type, then assume it is the same type as the other argument for this check. Other cases involving `unknown` will never find a match at this step.
2. Look for the best match.
    a. Make a list of all operators of the same name for which the input types match or can be coerced to match. (`unknown` literals are assumed to be coercible to anything for this purpose.) If there is only one, use it; else continue to the next step.
    b. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.
    c. Run through all candidates and keep those with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible 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 at the most positions where type coercion 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 "string" category if any candidate accepts that category (this bias towards string is appropriate since an unknown-type literal does look 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. Also note whether any of the candidates accept a preferred datatype within the selected category. Now discard operator candidates that do not accept the selected type category; furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.
    f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

5.2.1. Examples

5.2.1.1. Exponentiation Operator

There is only one exponentiation operator defined in the catalog, and it takes arguments of type `double precision`. The scanner assigns an initial type of `int4` to both arguments of this query expression:

```
tgl=> select 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
tgl=> select CAST(2 AS double precision) ^ CAST(3 AS double precision) AS
"Exp";
 Exp
-----
   8
(1 row)
```

or

```
tgl=> select 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

> **Note:** This last form has the least overhead, since no functions are called to do implicit type conversion. This is not an issue for small queries, but may have an impact on the performance of queries involving large tables.

5.2.1.2. String Concatenation

A string-like syntax is used for working with string types as well as for working with complex extended types. Strings with unspecified type are matched with likely operator candidates.

One unspecified argument:

```
tgl=> 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 of type `text`.

Concatenation on unspecified types:

```
tgl=> 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 bitstring-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 literals to.

5.2.1.3. Factorial

This example illustrates an interesting result. Traditionally, the factorial operator is defined for integers only. The Postgres operator catalog has only one entry for factorial, taking an integer operand. If given a non-integer numeric argument, Postgres will try to convert that argument to an integer for evaluation of the factorial.

```
tgl=> select (4.3 !);
 ?column?
----------
       24
(1 row)
```

Note: Of course, this leads to a mathematically suspect result, since in principle the factorial of a non-integer is not defined. However, the role of a database is not to teach mathematics, but to be a tool for data manipulation. If a user chooses to take the factorial of a floating point number, Postgres will try to oblige.

5.3. Functions

Function Call Type Resolution

1. Check for an exact match in the pg_proc system catalog. (Cases involving `unknown` will never find a match at this step.)
2. Look for the best match.
   a. Make a list of all functions of the same name with the same number of arguments for which the input types match or can be coerced to match. (`unknown` literals are assumed to be coercible to anything for this purpose.) If there is only one, use it; else continue to the next step.
   b. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.
   c. Run through all candidates and keep those with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible 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 at the most positions where type coercion 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 "string" category if any candidate accepts that category (this bias towards string is appropriate since an unknown-type literal does look 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. Also note whether any of the candidates accept a preferred datatype within the selected category. Now discard operator candidates that do not accept the selected type category; furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.
   f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.
3. If no best match could be identified, see whether the function call appears to be a trivial type coercion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some datatype. Furthermore, the function argument must be either an unknown-type literal or a type that is binary-compatible with the named datatype. When these conditions are met, the function argument is coerced to the named datatype.

5.3.1. Examples

5.3.1.1. Factorial Function

There is only one factorial function defined in the pg_proc catalog. So the following query automatically converts the `int2` argument to `int4`:

```
tgl=> select int4fac(int2 ’4’);
 int4fac
---------
      24
(1 row)
```

and is actually transformed by the parser to

```
tgl=> select int4fac(int4(int2 ’4’));
 int4fac
---------
      24
(1 row)
```

5.3.1.2. Substring Function

There are two `substr` functions declared in pg_proc. However, only one takes two arguments, of types `text` and `int4`.

If called with a string constant of unspecified type, the type is matched up directly with the only candidate function type:

```
tgl=> 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 coerce it to become `text`:

```
tgl=> select substr(varchar ’1234’, 3);
 substr
--------
     34
(1 row)
```

which is transformed by the parser to become

```
tgl=> select substr(text(varchar ’1234’), 3);
 substr
--------
     34
(1 row)
```

> **Note:** Actually, the parser is aware 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 explicit type conversion call is really inserted in this case.

And, if the function is called with an `int4`, the parser will try to convert that to `text`:

```
tgl=> select substr(1234, 3);
 substr
--------
     34
(1 row)
```

actually executes as

```
tgl=> select substr(text(1234), 3);
 substr
--------
     34
(1 row)
```

This succeeds because there is a conversion function text(int4) in the system catalog.

5.4. Query Targets

Query Target Type Resolution

1. Check for an exact match with the target.
2. Otherwise, try to coerce the expression to the target type. This will succeed if the two types are known binary-compatible, or if there is a conversion function. 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. If the target is a fixed-length type (e.g. `char` or `varchar` declared with a length) then try to find a sizing function for the target type. A sizing function is a function of the same name as the type, taking two arguments of which the first is that type and the second is an integer, and returning the same type. If one is found, it is applied, passing the column's declared length as the second parameter.

5.4.1. Examples

5.4.1.1. `varchar` Storage

For a target column declared as `varchar(4)` the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v varchar(4));
CREATE
tgl=> INSERT INTO vv SELECT ’abc’ || ’def’;
INSERT 392905 1
tgl=> SELECT * FROM vv;
  v
------
 abcd
(1 row)
```

What's 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 coerced to varchar to match the target column type. (But, since the parser knows that text and varchar are binary-compatible, this coercion is implicit and does not insert any real function call.) Finally, the sizing function `varchar(varchar,int4)` is found in the system catalogs and applied to the operator's result and the stored column length. This type-specific function performs the desired truncation.

5.5. UNION and CASE Constructs

The UNION and CASE 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. CASE uses the identical algorithm to match up its result expressions.

UNION and CASE Type Resolution

1. If all inputs are of type `unknown`, resolve as type `text` (the preferred type for string category). Otherwise, ignore the `unknown` inputs while choosing the type.
2. If the non-unknown inputs are not all of the same type category, fail.
3. If one or more non-unknown inputs are of a preferred type in that category, resolve as that type.
4. Otherwise, resolve as the type of the first non-unknown input.
5. Coerce all inputs to the selected type.

5.5.1. Examples

5.5.1.1. Underspecified Types

```
tgl=> SELECT text ’a’ AS "Text" UNION SELECT ’b’;
 Text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal ’b’ will be resolved as type text.

5.5.1.2. Simple UNION

```
tgl=> SELECT 1.2 AS "Double" UNION SELECT 1;
 Double
--------
      1
    1.2
(2 rows)
```

5.5.1.3. Transposed UNION

Here the output type of the union is forced to match the type of the first/top clause in the union:

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT CAST(’2.2’ AS REAL);
 All integers
--------------
            1
            2
(2 rows)
```

Since `REAL` is not a preferred type, the parser sees no reason to select it over `INTEGER` (which is what the 1 is), and instead falls back on the use-the-first-alternative rule. This example demonstrates that the preferred-type mechanism doesn't encode as much information as we'd like. Future versions of Postgres may support a more general notion of type preferences.

Chapter 6. Arrays

Postgres allows columns of a table to be defined as variable-length multi-dimensional arrays. Arrays of any built-in type or user-defined type can be created. To illustrate their use, we create this table:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

The above query will create a table named `sal_emp` with a `text` string (`name`), a one-dimensional array of type `integer` (`pay_by_quarter`), which shall represent the employee's salary by quarter, and a two-dimensional array of `text` (`schedule`), which represents the employee's weekly schedule.

Now we do some **INSERT**s; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know C, this is not unlike the syntax for initializing structures.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

Now, we can run some queries on `sal_emp`. First, we show how to access a single element of an array at a time. 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)
```

Postgres uses the *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 subscript* : *upper subscript* for one or more array dimensions. 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"},{""}}
(1 row)
```

We could also have written

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

with the same result. An array subscripting operation is taken to represent an array slice if any of the subscripts are written in the form *lower* : *upper*. A lower bound of 1 is assumed for any subscript where only one value is specified.

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or 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';
```

An array can be enlarged by assigning to an element adjacent to those already present, or by assigning to a slice that is adjacent to or overlaps the data already present. For example, if an array value currently has 4 elements, it will have five elements after an update that assigns to array[5]. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

The syntax for **CREATE TABLE** allows fixed-length arrays to be defined:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

However, the current implementation does not enforce the array size limits --- the behavior is the same as for arrays of unspecified length.

Actually, the current implementation does not enforce the declared number of dimensions either. Arrays of a particular base type are all considered to be of the same type, regardless of size or number of dimensions.

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][1:1]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps not so convenient for programs.

To search for a value in an array, you must check each value of the array. This can be done by hand (if you know the size of the array):

```
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. Although it is not part of the primary PostgreSQL distribution, in the contributions directory, there is an extension to PostgreSQL that defines new functions and operators for iterating over array values. Using this, the above query could be:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1:4] *= 10000;
```

To search the entire array (not just specified columns), you could use:

```
SELECT * FROM sal_emp WHERE pay_by_quarter *= 10000;
```

In addition, you could find rows where the array had all values equal to 10 000 with:

```
SELECT * FROM sal_emp WHERE pay_by_quarter **= 10000;
```

To install this optional module, look in the `contrib/array` directory of the PostgreSQL source distribution.

> **Tip:** Arrays are not lists; using arrays in the manner described in the previous paragraph is often a sign of database misdesign. The array field should generally be split off into a separate table. Tables can obviously be searched easily.

Chapter 7. Indices

Indices 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 indices also add overhead to the database system as a whole, so they should be used sensibly.

7.1. Introduction

The classical example for the need of an index is if there is a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application requires a lot of queries of the form

```
SELECT content FROM test1 WHERE id = constant;
```

Ordinarily, the system would have to scan the entire `test1` table row by row to find all matching entries. If there are a lot of rows in `test1` and only a few rows (possibly zero or one) returned by the query, then this is clearly an inefficient method. If the system were instructed to maintain an index on the `id` column, then it could 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 books of non-fiction: 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, and would not have to read the entire book to find the interesting location. As it is the task of the author to anticipate the items that the readers are most likely to look up, it is the task of the database programmer to foresee which indexes would be of advantage.

The following command would be used to create the 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. Indices can be added and removed from tables at any time.

Once the index is created, no further intervention is required: the system will use the index when it thinks it would be more efficient than a sequential table scan. But you may have to run the **VACUUM ANALYZE** command regularly to update statistics to allow the query planner to make educated decisions. Also read Chapter 11 for information about how to find out whether an index is used and when and why the planner may choose to *not* use an index.

Indices can also benefit **UPDATE**s and **DELETE**s with search conditions. Note that a query or data manipulation commands can only use at most one index per table. Indices can also be used in table join methods. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.

When an index is created, it has to be kept synchronized with the table. This adds overhead to data manipulation operations. Therefore indices that are non-essential or do not get used at all should be removed.

7.2. Index Types

Postgres provides several index types: B-tree, R-tree, and Hash. Each index type is more appropriate for a particular query type because of the algorithm it uses. By default, the **CREATE INDEX** command will create a B-tree index, which fits the most common situations. In particular, the Postgres query optimizer will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: <, <=, =, >=, >

R-tree indices are especially suited for spacial data. To create an R-tree index, use a command of the form

```
CREATE INDEX name ON table USING RTREE (column);
```

The Postgres query optimizer will consider using an R-tree index whenever an indexed column is involved in a comparison using one of these operators: `<<`, `&<`, `&>`, `>>`, `@`, `~=`, `&&` (Refer to Section 4.8 about the meaning of these operators.)

The query optimizer will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX name ON table USING HASH (column);
```

> **Note:** Because of the limited utility of hash indices, a B-tree index should generally be preferred over a hash index. We do not have sufficient evidence that hash indices are actually faster than B-trees even for = comparisons. Moreover, hash indices require coarser locks; see Section 9.7.

The B-tree index is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree index method implements standard R-trees using Guttman's quadratic split algorithm. The hash index is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

7.3. Multi-Column Indices

An index can be defined on more than one column. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(Say, you keep you your `/dev` directory in a database...) and you frequently make queries like

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it may 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 implementation supports multi-column indices. Up to 16 columns may be specified. (This limit can be altered when building Postgres; see the file `config.h`.)

The query optimizer can use a multi-column index for queries that involve the first *n* consecutive columns in the index (when used with appropriate operators), up to the total number of columns specified in the index definition. For example, an index on `(a, b, c)` can be used in queries involving all of `a`, `b`, and `c`, or in queries involving both `a` and `b`, or in queries involving only `a`, but not in other combinations. (In a query involving `a` and `c` the optimizer might choose to use the index for `a` only and treat `c` like an ordinary unindexed column.)

Multi-column indexes can only be used if the clauses involving the indexed columns are joined with `AND`. For instance,

```
SELECT name FROM test2 WHERE major = constant OR minor = constant;
```

cannot make use of the index `test2_mm_idx` defined above to look up both columns. (It can be used to look up only the `major` column, however.)

Multi-column indices should be used sparingly. Most of the time, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are almost certainly inappropriate.

7.4. Unique Indices

Indexes may 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 [, ...]);
```

Only B-tree indices can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values will not be allowed. NULL values are not considered equal.

PostgreSQL automatically creates unique indices when a table is declared with a unique constraint or a primary key, on the columns that make up the primary key or unique columns (a multi-column index, if appropriate), to enforce that constraint. A unique index can be added to a table at any later time, to add a unique constraint. (But a primary key cannot be added after table creation.)

7.5. Functional Indices

For a *functional index*, an index is defined on the result of a function applied to one or more columns of a single table. Functional indices can be used to obtain fast access to data based on the result of function calls.

For example, a common way to do case-insensitive comparisons is to use the `lower`:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

In order for that query to be able to use an index, it has to be defined on the result of the `lower(column)` operation:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

The function in the index definition can take more than one argument, but they must be table columns, not constants. Functional indices are always single-column (namely, the function result) even if the function uses more than one input field; there cannot be multi-column indices that contain function calls.

> **Tip:** The restrictions mentioned in the previous paragraph can easily be worked around by defining custom functions to use in the index definition that call the desired function(s) internally.

7.6. Operator Classes

An index definition may specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. 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. There are also some operator classes with special purposes:

The operator classes `box_ops` and `bigbox_ops` both support R-tree indices on the `box` data type. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating point exceptions from doing multiplication, addition, and subtraction on very large floating point coordinates. If the field on which your rectangles lie is about 20 000 units square or larger, you should use `bigbox_ops`.

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    FROM pg_am am, pg_amop amop,
         pg_opclass opc, pg_operator opr
    WHERE amop.amopid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY acc_name, ops_name, ops_comp;
```

7.7. Keys

Author: Written by Herouth Maoz (`<herouth@oumail.openu.ac.il>`). This originally appeared on the User's Mailing List on 1998-03-02 in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

What's the difference between:

PRIMARY KEY(fields,...) and
UNIQUE (fields,...)

- Is this an alias?
- If PRIMARY KEY is already unique, then why
  is there another kind of key named UNIQUE?

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named Life Science, the the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, within the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from Life Science to Biology, it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

Primary key:
Is used for identifying the row and relating to it.
Is impossible (or hard) to update.
Should not allow NULLs.

Unique field(s):
Are used as an alternative access to the row.
Are updatable, so long as they are kept unique.
NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indices are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. Postgres does allow non-unique indices, but indices used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementation aid that each RDBMS offers you, in order to cause commonly used queries to be done more efficiently. Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON table COLUMNS cols
```

(This is not an existing command, just an example.)

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields that is not unique as a secondary key, you really don't have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you'll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary MEMSTORE command, or an intelligent RDBMS that creates indices without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

7.8. Partial Indices

Author: This is from a reply to a question on the email list by Paul M. Aoki (<aoki@CS.Berkeley.EDU>) on 1998-08-11.

Note: Partial indices are not currently supported by PostgreSQL, but they were once supported by its predecessor Postgres, and much of the code is still there. We hope to revive support for this feature someday.

A *partial index* is an index built over a subset of a table; the subset is defined by a predicate. Postgres supported partial indices with arbitrary predicates. I believe IBM's DB2 for AS/400 supports partial indices using single-clause predicates.

The main motivation for partial indices is this: if all of the queries you ask that can profitably use an index fall into a certain range, why build an index over the whole table and suffer the associated space/time costs? (There are other reasons too; see *Stonebraker, M, 1989b* for details.)

The machinery to build, update and query partial indices isn't too bad. The hairy parts are index selection (which indices do I build?) and query optimization (which indices do I use?); i.e., the parts that involve deciding what predicate(s) match the workload/query in some useful way. For those who are into database theory, the problems are basically analogous to the corresponding materialized view problems, albeit with different cost parameters and formulae. These are, in the general case, hard problems for the standard ordinal SQL types; they're super-hard problems with black-box extension types, because the selectivity estimation technology is so crude.

Check *Stonebraker, M, 1989b*, *Olson, 1993*, and *Seshardri, 1995* for more information.

Chapter 8. Inheritance

Let's create two tables. The capitals table contains state capitals which are also cities. Naturally, the capitals table should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native Postgres type for variable length ASCII strings. The type of the attribute population is `float`, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, 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 descendants.

> **Note:** The inheritance hierarchy is a actually a directed acyclic graph.

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

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

which returns:

```
+----------+----------+
|name      | altitude |
+----------+----------+
|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 altitude of 500ft or higher:

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

+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
```

Here the ONLY before cities indicates that the query should be run over only cities 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.

In some cases you may wish to know which table a particular tuple 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.altitude
    FROM cities c
    WHERE c.altitude > 500;
```

which returns:

```
+---------+----------+----------+
|tableoid |name      | altitude |
+---------+----------+----------+
|37292    |Las Vegas | 2174     |
+---------+----------+----------+
|37280    |Mariposa  | 1953     |
+---------+----------+----------+
|37280    |Madison   | 845      |
+---------+----------+----------+
```

If you do a join with pg_class you can see the actual table name:

```
    SELECT p.relname, c.name, c.altitude
    FROM cities c, pg_class p
    WHERE c.altitude > 500 and c.tableoid = p.oid;
```

which returns:

```
+---------+----------+----------+
|relname  |name      | altitude |
+---------+----------+----------+
|capitals |Las Vegas | 2174     |
|cities   |Mariposa  | 1953     |
|cities   |Madison   | 845      |
+---------+----------+----------+
```

Deprecated: In previous versions of Postgres, the default was not to get access to child tables. This was found to be error prone and is also in violation of SQL99. Under the old syntax, to get the sub-tables you append "*" to the table name. For example

```
SELECT * from cities*;
```

You can still explicitly specify scanning child tables by appending "*", as well as explicitly specify not scanning child tables by writing ONLY. But beginning in version 7.1, the default behavior for an undecorated table name is to scan its child tables too, whereas before the default was not to do so. To get the old default behavior, set the configuration option `SQL_Inheritance` to off, e.g.,

```
SET SQL_Inheritance TO OFF;
```

or add a line in your `postgresql.conf` file.

Chapter 9. Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) is an advanced technique for improving database performance in a multi-user environment. Vadim Mikheev (`<vadim@krs.ru>`) provided the implementation for Postgres.

9.1. Introduction

Unlike most other database systems which use locks for concurrency control, Postgres maintains data consistency by using a multiversion model. This means that while querying a database each transaction sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing *transaction isolation* for each database session.

The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading.

9.2. Transaction Isolation

The ANSI/ISO SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

dirty reads

: A transaction reads data written by concurrent uncommitted transaction.

non-repeatable reads

: 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.

The four isolation levels and the corresponding behaviors are described below.

Table 9-1. ANSI/ISO SQL Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Postgres offers the read committed and serializable isolation levels.

9.3. Read Committed Isolation Level

Read Committed is the default isolation level in Postgres. When a transaction runs on this isolation level, a **SELECT** query sees only data committed before the query began and never sees either uncommitted data or changes committed during query execution by concurrent transactions. (However, the **SELECT** does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) Notice that two successive **SELECT**s can see different data, even though they are within a single transaction, when other transactions commit changes during execution of the first **SELECT**.

If a target row found by a query while executing an **UPDATE** statement (or **DELETE** or **SELECT FOR UPDATE**) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of commit (and if the row still exists; i.e. was not deleted by the other transaction), the query will be re-executed for this row to check that the new row version still satisfies the query search condition. If the new row version satisfies the query search condition then the row will be updated (or deleted or marked for update). Note that the starting point for the update will be the new row version; moreover, after the update the doubly-updated row is visible to subsequent **SELECT**s in the current transaction. Thus, the current transaction is able to see the effects of the other transaction for this specific row.

The partial transaction isolation provided by Read Committed level is adequate for many applications, and this level is fast and simple to use. However, for applications that do complex queries and updates, it may be necessary to guarantee a more rigorously consistent view of the database than Read Committed level provides.

9.4. Serializable Isolation Level

Serializable provides the highest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. However, applications using this level must be prepared to retry transactions due to serialization failures.

When a transaction is on the serializable level, a **SELECT** query sees only data committed before the transaction began and never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the **SELECT** does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) This is different from Read Committed in that the **SELECT** sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction.

If a target row found by a query while executing an **UPDATE** statement (or **DELETE** or **SELECT FOR UPDATE**) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of a concurrent transaction commit, a serializable transaction will be rolled back with the message

```
ERROR:  Can’t serialize access due to concurrent update
```

because a serializable transaction cannot modify rows changed by other transactions after the serializable transaction began.

When the application receives this error message, it should abort the current transaction and then retry the whole transaction from the beginning. The second time through, the transaction sees 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 may need to be retried --- read-only transactions never have serialization conflicts.

Serializable transaction level provides a rigorous guarantee that each transaction sees a wholly consistent view of the database. However, the application has to be prepared to retry transactions when concurrent updates make it impossible to sustain the illusion of serial execution, and the cost of redoing complex transactions may be significant. So this level is recommended only when update queries contain logic sufficiently complex that they may give wrong answers in Read Committed level.

9.5. Data consistency checks at the application level

Because readers in Postgres don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, if a row is returned by **SELECT** it doesn't mean that the row still exists at the time it is returned (i.e. sometime after the current transaction began); the row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

Another way to think about it is that each transaction sees a snapshot of the database contents, and concurrently executing transactions may very well see different snapshots. So the whole concept of "now" is somewhat suspect anyway. This is not normally a big problem if the client applications are isolated from each other, but if the clients can communicate via channels outside the database then serious confusion may ensue.

To ensure the current existence of a row and protect it against concurrent updates one must use **SELECT FOR UPDATE** or an appropriate **LOCK TABLE** statement. (**SELECT FOR UPDATE** locks just the returned rows against concurrent updates, while **LOCK TABLE** protects the whole table.) This should be taken into account when porting applications to Postgres from other environments.

> **Note:** Before version 6.5 Postgres used read-locks and so the above consideration is also the case when upgrading to 6.5 (or higher) from previous Postgres versions.

9.6. Locking and Tables

Postgres provides various lock modes to control concurrent access to data in tables. Some of these lock modes are acquired by Postgres automatically before statement execution, while others are provided to be used by applications. All lock modes acquired in a transaction are held for the duration of the transaction.

9.6.1. Table-level locks

AccessShareLock

A read-lock mode acquired automatically on tables being queried.

Conflicts with AccessExclusiveLock only.

RowShareLock

Acquired by **SELECT FOR UPDATE** and **LOCK TABLE** for `IN ROW SHARE MODE` statements.

Conflicts with ExclusiveLock and AccessExclusiveLock modes.

RowExclusiveLock

Acquired by **UPDATE**, **DELETE**, **INSERT** and **LOCK TABLE** for `IN ROW EXCLUSIVE MODE` statements.

Conflicts with ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareLock

Acquired by **CREATE INDEX** and **LOCK TABLE** table for `IN SHARE MODE` statements.

Conflicts with RowExclusiveLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareRowExclusiveLock

Acquired by **LOCK TABLE** for `IN SHARE ROW EXCLUSIVE MODE` statements.

Conflicts with RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ExclusiveLock

Acquired by **LOCK TABLE** table for `IN EXCLUSIVE MODE` statements.

Conflicts with RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

AccessExclusiveLock

Acquired by **ALTER TABLE**, **DROP TABLE**, **VACUUM** and **LOCK TABLE** statements.

Conflicts with all modes (AccessShareLock, RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock).

Note: Only AccessExclusiveLock blocks **SELECT** (without `FOR UPDATE`) statement.

9.6.2. Row-level locks

These locks are acquired when rows are being updated (or deleted or marked for update). Row-level locks don't affect data querying. They block writers to *the same row* only.

Postgres doesn't remember any information about modified rows in memory and so has no limit to the number of rows locked at one time. However, locking a row may cause a disk write; thus, for example, **SELECT FOR UPDATE** will modify selected rows to mark them and so will result in disk writes.

In addition to table and row locks, short-term 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 tuple is fetched or updated. Application writers normally need not be concerned with page-level locks, but we mention them for completeness.

9.7. Locking and Indices

Though Postgres provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in Postgres.

The various index types are handled as follows:

GiST and R-Tree indices

Share/exclusive index-level locks are used for read/write access. Locks are released after statement is done.

Hash indices

Share/exclusive page-level locks are used for read/write access. Locks are released after page is processed.

Page-level locks provide better concurrency than index-level ones but are subject to deadlocks.

Btree indices

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index tuple is fetched/inserted.

Btree indices provide the highest concurrency without deadlock conditions.

In short, btree indices are the recommended index type for concurrent applications.

Chapter 10. Managing a Database

> **Note:** This section is currently a thinly disguised copy of the Tutorial. Needs to be augmented. - thomas 1998-01-12

Although the *site administrator* is responsible for overall management of the Postgres installation, some databases within the installation may be managed by another person, designated the *database administrator*. This assignment of responsibilities occurs when a database is created. A user may be assigned explicit privileges to create databases and/or to create new users. A user assigned both privileges can perform most administrative task within Postgres, but will not by default have the same operating system privileges as the site administrator.

The Database Administrator's Guide covers these topics in more detail.

10.1. Database Creation

Databases are created by the **create database** issued from within Postgres. createdb is a command-line utility provided to give the same functionality from outside Postgres.

The Postgres backend must be running for either method to succeed, and the user issuing the command must be the Postgres *superuser* or have been assigned database creation privileges by the superuser.

To create a new database named `mydb` from the command line, type

```
% createdb mydb
```

and to do the same from within psql type

```
=> CREATE DATABASE mydb;
```

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: Permission denied.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length.

10.2. Alternate Database Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access actually occurs through the database backend, so that any location specified must be accessible by the backend.

Alternate database locations are created and referenced by an environment variable which gives the absolute path to the intended storage location. This environment variable must have been defined before the postmaster was started and the location it points to must be writable by the postgres administrator account. Consult with the site administrator regarding preconfigured alternate database locations. Any valid environment variable name may be used to reference an alternate location, although using variable names with a prefix of PGDATA is recommended to avoid confusion and conflict with other variables.

> **Note:** In previous versions of Postgres, it was also permissable to use an absolute path name to specify an alternate storage location. Although the environment variable style of specification is to be preferred since it allows the site administrator more flexibility in managing disk storage, it is also possible to use an absolute path to specify an alternate location. The administrator's guide discusses how to enable this feature.

For security and integrity reasons, any path or environment variable specified has some additional path fields appended. Alternate database locations must be prepared by running initlocation.

To create a data storage area using the environment variable PGDATA2 (for this example set to `/alt/postgres`), ensure that `/alt/postgres` already exists and is writable by the Postgres administrator account. Then, from the command line, type

```
% initlocation PGDATA2
Creating Postgres database system directory /alt/postgres/data
Creating Postgres database system directory /alt/postgres/data/base
```

To create a database in the alternate storage area PGDATA2 from the command line, use the following command:

```
% createdb -D PGDATA2 mydb
```

and to do the same from within psql type

```
=> CREATE DATABASE mydb WITH LOCATION = 'PGDATA2';
```

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: permission denied
```

If the specified location does not exist or the database backend does not have permission to access it or to write to directories under it, you will see the following:

```
ERROR:  The database path '/no/where' is invalid. This may be due to a
character that is not allowed or because the chosen path isn't permitted
for databases.
```

10.3. Accessing a Database

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

- running the PostgreSQL interactive terminal psql which allows you to interactively enter, edit, and execute SQL commands.
- writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in *The PostgreSQL Programmer's Guide*.

You might want to start up psql, to try out the examples in this manual. It can be activated for the mydb database by typing the command:

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

mydb=>
```

This prompt indicates that psql is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, "`\`". For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the "`\g`" is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to Unix, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type **\?** at the psql prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by "`--`". Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "`/* ... */`".

10.4. Destroying a Database

If you are the owner of the database mydb, you can destroy it using the following Unix command:

```
% dropdb mydb
```

This action physically removes all of the Unix files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 11. Performance Tips

Query performance can be affected by many things. Some of these can be manipulated by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning Postgres performance.

11.1. Using EXPLAIN

Author: Written by Tom Lane, from e-mail dated 2000-03-27.

Postgres devises a *query plan* for each query it is given. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance. You can use the **EXPLAIN** command to see what query plan the system creates for any query. Unfortunately, plan-reading is an art that deserves a tutorial, and I haven't had time to write one. Here is some quick & dirty explanation.

The numbers that are currently quoted by EXPLAIN are:

- Estimated start-up cost (time expended before output scan can start, e.g., time to do the sorting in a SORT node).
- Estimated total cost (if all tuples are retrieved, which they may not be --- a query with a LIMIT will stop short of paying the total cost, for example).
- Estimated number of rows output by this plan node (again, without regard for any LIMIT).
- Estimated average width (in bytes) of rows output by this plan node.

The costs are measured in units of disk page fetches. (CPU effort estimates are converted into disk-page units using some fairly arbitrary fudge-factors. If you want to experiment with these factors, see the list of run-time configuration parameters in the *Administrator's Guide*.)

It's important to note 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/optimizer cares about. In particular, the cost does not consider the time spent transmitting result tuples to the frontend --- which could be a pretty dominant factor in the true elapsed time, but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same tuple set, we trust.)

Rows output is a little tricky because it is *not* the number of rows processed/scanned by the query --- it is usually less, reflecting the estimated selectivity of any WHERE-clause constraints that are being applied at this node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query (again, without considering the effects of LIMIT).

Average width is pretty bogus because the thing really doesn't have any idea of the average length of variable-length columns. I'm thinking about improving that in the future, but it may not be worth the trouble, because the width isn't used for very much.

Here are some examples (using the regress test database after a vacuum analyze, and almost-7.0

sources):

```
regression=# explain select * from tenk1;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

This is about as straightforward as it gets. If you do

```
select * from pg_class where relname = 'tenk1';
```

you'll find out that tenk1 has 233 disk pages and 10000 tuples. So the cost is estimated at 233 block reads, defined as 1.0 apiece, plus 10000 * cpu_tuple_cost which is currently 0.01 (try **show cpu_tuple_cost**).

Now let's modify the query to add a qualification clause:

```
regression=# explain select * from tenk1 where unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1000 width=148)
```

The estimate of output rows has gone down because of the WHERE clause. (This estimate is uncannily accurate because tenk1 is a particularly simple case --- the unique1 column has 10000 distinct values ranging from 0 to 9999, so the estimator's linear interpolation between min and max column values is dead-on.) 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 to reflect the extra CPU time spent checking the WHERE condition.

Modify the query to restrict the qualification even more:

```
regression=# explain select * from tenk1 where unique1 < 100;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..89.35 rows=100
width=148)
```

and you will see that if we make the WHERE condition selective enough, the planner will eventually decide that an indexscan is cheaper than a sequential scan. This plan will only have to visit 100 tuples because of the index, so it wins despite the fact that each individual fetch is expensive.

Add another condition to the qualification:

```
regression=# explain select * from tenk1 where unique1 < 100 and
regression-# stringu1 = 'xxx';
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..89.60 rows=1
width=148)
```

The added clause "stringu1 = 'xxx'" reduces the output-rows estimate, but not the cost because we still have to visit the same set of tuples.

Let's try joining two tables, using the fields we have been discussing:

```
regression=# explain select * from tenk1 t1, tenk2 t2 where t1.unique1 <
100
regression-# and t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..144.07 rows=100 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
             (cost=0.00..89.35 rows=100 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
             (cost=0.00..0.53 rows=1 width=148)
```

In this nested-loop join, the outer scan is the same indexscan we had in the example before last, and so its cost and row count are the same because we are applying the "unique1 < 100" WHERE clause at that node. The "t1.unique2 = t2.unique2" clause isn't relevant yet, so it doesn't affect the outer scan's row count. For the inner scan, the current outer-scan tuple's unique2 value is plugged into the inner indexscan to produce an indexqual like "t2.unique2 = *constant*". So we get the same inner-scan plan and costs that we'd get from, say, "explain select * from tenk2 where unique2 = 42". The loop node's costs are then set on the basis of the outer scan's cost, plus one repetition of the inner scan for each outer tuple (100 * 0.53, here), plus a little CPU time for join processing.

In this example the loop's output row count is the same as the product of the two scans' row counts, but that's not true in general, because in general you can have WHERE clauses that mention both relations and so can only be applied at the join point, not to either input scan. For example, if we added "WHERE ... AND t1.hundred < t2.hundred", that'd decrease the output row count of the join node, but not change either input scan.

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the winner, using the enable/disable flags for each plan type. (This is a crude tool, but useful. See also Section 11.2.)

```
regression=# set enable_nestloop = off;
SET VARIABLE
regression=# explain select * from tenk1 t1, tenk2 t2 where t1.unique1 <
100
regression-# and t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=89.60..574.10 rows=100 width=296)
  ->  Seq Scan on tenk2 t2
               (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=89.35..89.35 rows=100 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
               (cost=0.00..89.35 rows=100 width=148)
```

This plan proposes to extract the 100 interesting rows of tenk1 using ye same olde indexscan, stash them into an in-memory hash table, and then do a sequential scan of tenk2, probing into the hash table for possible matches of "t1.unique2 = t2.unique2" at each tenk2 tuple. The cost to read tenk1 and set up the hash table is entirely start-up cost for the hash join, since we won't get any tuples out until we can start reading tenk2. The total time estimate for the join also includes a pretty hefty charge for CPU time to probe the hash table 10000 times. Note, however, that we are NOT charging 10000 times 89.35; the hash table setup is only done once in this plan type.

11.2. Controlling the Planner with Explicit JOINs

Beginning with Postgres 7.1 it is possible to control the query planner to some extent by using 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 clause a.id = b.id, and then joins C to this joined table, using the other WHERE clause. 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 WHERE clause to allow optimization of the join. (All joins in the Postgres 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 may 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 may take an annoyingly long time. When there are too many input tables, the Postgres planner will switch from exhaustive search to a *genetic* probabilistic search through a limited number of possibilities. (The switchover threshold is set by the GEQO_THRESHOLD run-time parameter described in the *Administrator's Guide*.) 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 much 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 Postgres 7.1, the planner treats all explicit JOIN syntaxes as constraining the join order, even though it is not logically necessary to make such a constraint for inner joins. Therefore, although all of these queries give the same result:

```
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);
```

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.

You do not need to constrain the join order completely in order to cut search time, because it's OK to use JOIN operators in a plain FROM list. For example,

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

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.

If you have a mix of outer and inner joins in a complex query, you might not want to constrain the planner's search for a good ordering of inner joins inside an outer join. You can't do that directly in the JOIN syntax, but you can get around the syntactic limitation by using subselects. For example,

```
SELECT * FROM d LEFT JOIN
        (SELECT * FROM a, b, c WHERE ...) AS ss
        ON (...);
```

Here, joining D must be the last step in the query plan, but the planner is free to consider various join orders for A,B,C.

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.

11.3. Populating a Database

One may need to do a large number of table insertions when first populating a database. Here are some tips and techniques for making that as efficient as possible.

11.3.1. Disable Auto-commit

Turn off auto-commit and just do one commit at the end. Otherwise Postgres is doing a lot of work for each record added. In general when you are doing bulk inserts, you want to turn off some of the database features to gain speed.

11.3.2. Use COPY FROM

Use **COPY FROM STDIN** to load all the records in one command, instead of a series of INSERT commands. This reduces parsing, planning, etc overhead a great deal. If you do this then it's not necessary to fool around with autocommit, since it's only one command anyway.

11.3.3. Remove Indices

If you are loading a freshly created table, the fastest way is to create the table, bulk-load with COPY, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each record is loaded.

If you are augmenting an existing table, you can **DROP INDEX**, load the table, then recreate the index. Of course, the database performance for other users may be adversely affected during the time that the index is missing.

Appendix A. Date/Time Support

A.1. Time Zones

Postgres must have internal tabular information for time zone decoding, since there is no *nix standard system interface to provide access to general, cross-timezone information. The underlying OS *is* used to provide time zone information for *output*.

Table A-1. Postgres Recognized Time Zones

Time Zone	Offset from UTC	Description
NZDT	+13:00	New Zealand Daylight Time
IDLE	+12:00	International Date Line, East
NZST	+12:00	New Zealand Std Time
NZT	+12:00	New Zealand Time
AESST	+11:00	Australia Eastern Summer Std Time
ACSST	+10:30	Central Australia Summer Std Time
CADT	+10:30	Central Australia Daylight Savings Time
SADT	+10:30	South Australian Daylight Time
AEST	+10:00	Australia Eastern Std Time
EAST	+10:00	East Australian Std Time
GST	+10:00	Guam Std Time, USSR Zone 9
LIGT	+10:00	Melbourne, Australia
ACST	+09:30	Central Australia Std Time
SAST	+09:30	South Australia Std Time
CAST	+09:30	Central Australia Std Time
AWSST	+9:00	Australia Western Summer Std Time
JST	+9:00	Japan Std Time,USSR Zone 8
KST	+9:00	Korea Standard Time
WDT	+9:00	West Australian Daylight Time
MT	+8:30	Moluccas Time
AWST	+8:00	Australia Western Std Time
CCT	+8:00	China Coastal Time
WADT	+8:00	West Australian Daylight Time
WST	+8:00	West Australian Std Time

Time Zone	Offset from UTC	Description
JT	+7:30	Java Time
WAST	+7:00	West Australian Std Time
IT	+3:30	Iran Time
BT	+3:00	Baghdad Time
EETDST	+3:00	Eastern Europe Daylight Savings Time
CETDST	+2:00	Central European Daylight Savings Time
EET	+2:00	Eastern Europe, USSR Zone 1
FWT	+2:00	French Winter Time
IST	+2:00	Israel Std Time
MEST	+2:00	Middle Europe Summer Time
METDST	+2:00	Middle Europe Daylight Time
SST	+2:00	Swedish Summer Time
BST	+1:00	British Summer Time
CET	+1:00	Central European Time
DNT	+1:00	Dansk Normal Tid
FST	+1:00	French Summer Time
MET	+1:00	Middle Europe Time
MEWT	+1:00	Middle Europe Winter Time
MEZ	+1:00	Middle Europe Zone
NOR	+1:00	Norway Standard Time
SET	+1:00	Seychelles Time
SWT	+1:00	Swedish Winter Time
WETDST	+1:00	Western Europe Daylight Savings Time
GMT	0:00	Greenwich Mean Time
WET	0:00	Western Europe
WAT	-1:00	West Africa Time
NDT	-2:30	Newfoundland Daylight Time
ADT	-03:00	Atlantic Daylight Time
NFT	-3:30	Newfoundland Standard Time
NST	-3:30	Newfoundland Standard Time
AST	-4:00	Atlantic Std Time (Canada)
EDT	-4:00	Eastern Daylight Time

Time Zone	Offset from UTC	Description
CDT	-5:00	Central Daylight Time
EST	-5:00	Eastern Standard Time
CST	-6:00	Central Std Time
MDT	-6:00	Mountain Daylight Time
MST	-7:00	Mountain Standard Time
PDT	-7:00	Pacific Daylight Time
PST	-8:00	Pacific Std Time
YDT	-8:00	Yukon Daylight Time
HDT	-9:00	Hawaii/Alaska Daylight Time
YST	-9:00	Yukon Standard Time
AHST	-10:00	Alaska-Hawaii Std Time
CAT	-10:00	Central Alaska Time
NT	-11:00	Nome Time
IDLW	-12:00	International Date Line, West

A.1.1. Australian Time Zones

Australian time zones and their naming variants account for fully one quarter of all time zones in the Postgres time zone lookup table. There are two naming conflicts with common time zones defined in the United States, `CST` and `EST`.

If the compiler option USE_AUSTRALIAN_RULES is set then `CST`, `EST`, and `SAT` will be interpreted using Australian conventions. Without this option, `SAT` is interpreted as a noise word indicating "`Saturday`".

Table A-2. Postgres Australian Time Zones

Time Zone	Offset from UTC	Description
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time
SAT	+9:30	South Australian Std Time

A.1.2. Date/Time Input Interpretation

The date/time types are all decoded using a common set of routines.

Date/Time Input Interpretation

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.
   a. If the token contains a colon (":"), this is a time string.
   b. If the token contains a dash ("-"), slash ("/"), or dot ("."), this is a date string which may have a text month.
   c. If the token is numeric only, then it is either a single field or an ISO-8601 concatenated date (e.g. "19990113" for January 13, 1999) or time (e.g. 141516 for 14:15:16).
   d. If the token starts with a plus ("+") or minus ("-"), then it is either a time zone or a special field.
2. If the token is a text string, match up with possible strings.
   a. Do a binary-search table lookup for the token as either a special string (e.g. `today`), day (e.g. `Thursday`), month (e.g. `January`), or noise word (e.g. `on`).

      Set field values and bit mask for fields. For example, set year, month, day for `today`, and additionally hour, minute, second for `now`.
   b. If not found, do a similar binary-search table lookup to match the token with a time zone.
   c. If not found, throw an error.
3. The token is a number or number field.
   a. If there are more than 4 digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g. `19990118`). 8 and 6 digits are interpreted as year, month, and day, while 7 and 5 digits are interpreted as year, day of year, respectively.
   b. If the token is three digits and a year has already been decoded, then interpret as day of year.
   c. If four or more digits, then interpret as a year.
   d. If in European date mode, and if the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.
   e. If the month field has not yet been read, and if the value is less than or equal to 12, then interpret as a month.
   f. If the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.
   g. If two digits or four or more digits, then interpret as a year.
   h. Otherwise, throw an error.
4. If BC has been specified, negate the year and offset by one for internal storage (there is no year zero in the Gregorian calendar, so numerically 1BC becomes year zero).
5. If BC was not specified, and if the year field was two digits in length, then adjust the year to 4 digits. If the field was less than 70, then add 2000; otherwise, add 1900.

Tip: Gregorian years 1-99AD may be entered by using 4 digits with leading zeros (e.g. 0099 is 99AD). Previous versions of Postgres accepted years with three digits and with single digits, but as of version 7.0 the rules have been tightened up to reduce the possibility of ambiguity.

A.2. History of Units

Note: Contributed by José Soares (`<jose@sferacarta.com>`)

The Julian Day was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from the Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). Astronomers have used the Julian period to assign a unique number to every day since 1 January 4713 BC. This is the so-called Julian Day (JD). JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

"Julian Day" is different from "Julian Date". The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use until the 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of about 1 day in 128 years. The accumulating calendar error prompted Pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent.

In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar only years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek orthodox countries didn't change until the start of this century. The reform was observed by Great Britain and Dominions (including what is now the USA) in 1752. Thus 2 Sep 1752 was followed by 14 Sep 1752. This is why Unix systems have cal produce the following:

```
% cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Note: SQL92 states that Within the definition of a datetime literal, the datetime values are constrained by the natural rules for dates and times according to the Gregorian calendar .Dates between 1752-09-03 and 1752-09-13, although eliminated in some countries by Papal fiat, conform to natural rules and are hence valid dates.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century BC. Legend has it that the Emperor Huangdi invented the calendar in 2637 BC. The People's Republic of China uses the Gregorian calendar for civil purposes. Chinese calendar is used for determining festivals.

Appendix B. SQL Key Words

Table B-1 lists all tokens that are key words in the SQL standard and in PostgreSQL 7.1. Background information can be found in Section 1.1.1.

SQL distinguishes between *reserved* and *non-reserved* key words. Reserved key words are the only real key words; they are never allowed as identifiers. Non-reserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most non-reserved key words are actually the names of built-in tables and functions specified by SQL and the concept of non-reserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the PostgreSQL parser life is a bit more complicated. There are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Most SQL reserved key words are not completely reserved in PostgreSQL, but can be used as column label (as in `SELECT 55 AS CHECK`, even though CHECK is a reserved key word).

In Table B-1 in the column for PostgreSQL we classify as non-reserved those key words that are explicitly known to the parser but are allowed in most or all contexts where an identifier is expected. Labeled reserved are those tokens that are only allowed as AS column label names (and perhaps in very few other contexts). The token AS is the only exception: it cannot even be used as a column label. As a general rule, if you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

It is important to understand before studying Table B-1 that the fact that a key word is not reserved in PostgreSQL does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table B-1. SQL Key Words

Key Word	PostgreSQL	SQL 99	SQL 92
ABORT	reserved		
ABS		non-reserved	
ABSOLUTE	non-reserved	reserved	reserved
ACCESS	non-reserved		
ACTION	non-reserved	reserved	reserved
ADA		non-reserved	non-reserved
ADD	non-reserved	reserved	reserved
ADMIN		reserved	
AFTER	non-reserved	reserved	
AGGREGATE	non-reserved	reserved	
ALIAS		reserved	
ALL	reserved	reserved	reserved
ALLOCATE		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
ALTER	non-reserved	reserved	reserved
ANALYSE	reserved		
ANALYZE	reserved		
AND	reserved	reserved	reserved
ANY	reserved	reserved	reserved
ARE		reserved	reserved
ARRAY		reserved	
AS	reserved	reserved	reserved
ASC	reserved	reserved	reserved
ASENSITIVE		non-reserved	
ASSERTION		reserved	reserved
ASSIGNMENT		non-reserved	
ASYMMETRIC		non-reserved	
AT	non-reserved	reserved	reserved
ATOMIC		non-reserved	
AUTHORIZATION		reserved	reserved
AVG		non-reserved	reserved
BACKWARD	non-reserved		
BEFORE	non-reserved	reserved	
BEGIN	non-reserved	reserved	reserved
BETWEEN	reserved	non-reserved	reserved
BINARY	reserved	reserved	
BIT	reserved	reserved	reserved
BITVAR		non-reserved	
BIT_LENGTH		non-reserved	reserved
BLOB		reserved	
BOOLEAN		reserved	
BOTH	reserved	reserved	reserved
BREADTH		reserved	
BY	non-reserved	reserved	reserved
C		non-reserved	non-reserved
CACHE	non-reserved		
CALL		reserved	
CALLED		non-reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
CARDINALITY		non-reserved	
CASCADE	non-reserved	reserved	reserved
CASCADED		reserved	reserved
CASE	reserved	reserved	reserved
CAST	reserved	reserved	reserved
CATALOG		reserved	reserved
CATALOG_NAME		non-reserved	non-reserved
CHAIN	non-reserved	non-reserved	
CHAR	reserved	reserved	reserved
CHARACTER	reserved	reserved	reserved
CHARACTERISTICS	non-reserved		
CHARACTER_LENGTH		non-reserved	reserved
CHARACTER_SET_CATALOG		non-reserved	non-reserved
CHARACTER_SET_NAME		non-reserved	non-reserved
CHARACTER_SET_SCHEMA		non-reserved	non-reserved
CHAR_LENGTH		non-reserved	reserved
CHECK	reserved	reserved	reserved
CHECKED		non-reserved	
CHECKPOINT	non-reserved		
CLASS		reserved	
CLASS_ORIGIN		non-reserved	non-reserved
CLOB		reserved	
CLOSE	non-reserved	reserved	reserved
CLUSTER	reserved		
COALESCE	reserved	non-reserved	reserved
COBOL		non-reserved	non-reserved
COLLATE	reserved	reserved	reserved
COLLATION		reserved	reserved
COLLATION_CATALOG		non-reserved	non-reserved
COLLATION_NAME		non-reserved	non-reserved
COLLATION_SCHEMA		non-reserved	non-reserved
COLUMN	reserved	reserved	reserved
COLUMN_NAME		non-reserved	non-reserved
COMMAND_FUNCTION		non-reserved	non-reserved

Key Word	PostgreSQL	SQL 99	SQL 92
COMMAND_FUNCTION_CODE		non-reserved	
COMMENT	non-reserved		
COMMIT	non-reserved	reserved	reserved
COMMITTED	non-reserved	non-reserved	non-reserved
COMPLETION		reserved	
CONDITION_NUMBER		non-reserved	non-reserved
CONNECT		reserved	reserved
CONNECTION		reserved	reserved
CONNECTION_NAME		non-reserved	non-reserved
CONSTRAINT	reserved	reserved	reserved
CONSTRAINTS	non-reserved	reserved	reserved
CONSTRAINT_CATALOG		non-reserved	non-reserved
CONSTRAINT_NAME		non-reserved	non-reserved
CONSTRAINT_SCHEMA		non-reserved	non-reserved
CONSTRUCTOR		reserved	
CONTAINS		non-reserved	
CONTINUE		reserved	reserved
CONVERT		non-reserved	reserved
COPY	reserved		
CORRESPONDING		reserved	reserved
COUNT		non-reserved	reserved
CREATE	non-reserved	reserved	reserved
CREATEDB	non-reserved		
CREATEUSER	non-reserved		
CROSS	reserved	reserved	reserved
CUBE		reserved	
CURRENT		reserved	reserved
CURRENT_DATE	reserved	reserved	reserved
CURRENT_PATH		reserved	
CURRENT_ROLE		reserved	
CURRENT_TIME	reserved	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved	reserved
CURRENT_USER	reserved	reserved	reserved
CURSOR	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
CURSOR_NAME		non-reserved	non-reserved
CYCLE	non-reserved	reserved	
DATA		reserved	non-reserved
DATABASE	non-reserved		
DATE		reserved	reserved
DATETIME_INTERVAL_CODE		non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION		non-reserved	non-reserved
DAY	non-reserved	reserved	reserved
DEALLOCATE		reserved	reserved
DEC	reserved	reserved	reserved
DECIMAL	reserved	reserved	reserved
DECLARE	non-reserved	reserved	reserved
DEFAULT	reserved	reserved	reserved
DEFERRABLE	reserved	reserved	reserved
DEFERRED	non-reserved	reserved	reserved
DEFINED		non-reserved	
DEFINER		non-reserved	
DELETE	non-reserved	reserved	reserved
DELIMITERS	non-reserved		
DEPTH		reserved	
DEREF		reserved	
DESC	reserved	reserved	reserved
DESCRIBE		reserved	reserved
DESCRIPTOR		reserved	reserved
DESTROY		reserved	
DESTRUCTOR		reserved	
DETERMINISTIC		reserved	
DIAGNOSTICS		reserved	reserved
DICTIONARY		reserved	
DISCONNECT		reserved	reserved
DISPATCH		non-reserved	
DISTINCT	reserved	reserved	reserved
DO	reserved		
DOMAIN		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
DOUBLE	non-reserved	reserved	reserved
DROP	non-reserved	reserved	reserved
DYNAMIC		reserved	
DYNAMIC_FUNCTION		non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE		non-reserved	
EACH	non-reserved	reserved	
ELSE	reserved	reserved	reserved
ENCODING	non-reserved		
END	reserved	reserved	reserved
END-EXEC		reserved	reserved
EQUALS		reserved	
ESCAPE	non-reserved	reserved	reserved
EVERY		reserved	
EXCEPT	reserved	reserved	reserved
EXCEPTION		reserved	reserved
EXCLUSIVE	non-reserved		
EXEC		reserved	reserved
EXECUTE	non-reserved	reserved	reserved
EXISTING		non-reserved	
EXISTS	reserved	non-reserved	reserved
EXPLAIN	reserved		
EXTEND	reserved		
EXTERNAL		reserved	reserved
EXTRACT	reserved	non-reserved	reserved
FALSE	reserved	reserved	reserved
FETCH	non-reserved	reserved	reserved
FINAL		non-reserved	
FIRST		reserved	reserved
FLOAT	reserved	reserved	reserved
FOR	reserved	reserved	reserved
FORCE	non-reserved		
FOREIGN	reserved	reserved	reserved
FORTRAN		non-reserved	non-reserved
FORWARD	non-reserved		

Key Word	PostgreSQL	SQL 99	SQL 92
FOUND		reserved	reserved
FREE		reserved	
FROM	reserved	reserved	reserved
FULL	reserved	reserved	reserved
FUNCTION	non-reserved	reserved	
G		non-reserved	
GENERAL		reserved	
GENERATED		non-reserved	
GET		reserved	reserved
GLOBAL	reserved	reserved	reserved
GO		reserved	reserved
GOTO		reserved	reserved
GRANT	non-reserved	reserved	reserved
GRANTED		non-reserved	
GROUP	reserved	reserved	reserved
GROUPING		reserved	
HANDLER	non-reserved		
HAVING	reserved	reserved	reserved
HIERARCHY		non-reserved	
HOLD		non-reserved	
HOST		reserved	
HOUR	non-reserved	reserved	reserved
IDENTITY		reserved	reserved
IGNORE		reserved	
ILIKE	reserved		
IMMEDIATE	non-reserved	reserved	reserved
IMPLEMENTATION		non-reserved	
IN	reserved	reserved	reserved
INCREMENT	non-reserved		
INDEX	non-reserved		
INDICATOR		reserved	reserved
INFIX		non-reserved	
INHERITS	non-reserved		
INITIALIZE		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
INITIALLY	reserved	reserved	reserved
INNER	reserved	reserved	reserved
INOUT	reserved	reserved	
INPUT		reserved	reserved
INSENSITIVE	non-reserved	non-reserved	reserved
INSERT	non-reserved	reserved	reserved
INSTANCE		non-reserved	
INSTANTIABLE		non-reserved	
INSTEAD	non-reserved		
INT		reserved	reserved
INTEGER		reserved	reserved
INTERSECT	reserved	reserved	reserved
INTERVAL	non-reserved	reserved	reserved
INTO	reserved	reserved	reserved
INVOKER		non-reserved	
IS	reserved	reserved	reserved
ISNULL	reserved		
ISOLATION	non-reserved	reserved	reserved
ITERATE		reserved	
JOIN	reserved	reserved	reserved
K		non-reserved	
KEY	non-reserved	reserved	reserved
KEY_MEMBER		non-reserved	
KEY_TYPE		non-reserved	
LANCOMPILER	non-reserved		
LANGUAGE	non-reserved	reserved	reserved
LARGE		reserved	
LAST		reserved	reserved
LATERAL		reserved	
LEADING	reserved	reserved	reserved
LEFT	reserved	reserved	reserved
LENGTH		non-reserved	non-reserved
LESS		reserved	
LEVEL	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
LIKE	reserved	reserved	reserved
LIMIT	reserved	reserved	
LISTEN	reserved		
LOAD	reserved		
LOCAL	reserved	reserved	reserved
LOCALTIME		reserved	
LOCALTIMESTAMP		reserved	
LOCATION	non-reserved		
LOCATOR		reserved	
LOCK	reserved		
LOWER		non-reserved	reserved
M		non-reserved	
MAP		reserved	
MATCH	non-reserved	reserved	reserved
MAX		non-reserved	reserved
MAXVALUE	non-reserved		
MESSAGE_LENGTH		non-reserved	non-reserved
MESSAGE_OCTET_LENGTH		non-reserved	non-reserved
MESSAGE_TEXT		non-reserved	non-reserved
METHOD		non-reserved	
MIN		non-reserved	reserved
MINUTE	non-reserved	reserved	reserved
MINVALUE	non-reserved		
MOD		non-reserved	
MODE	non-reserved		
MODIFIES		reserved	
MODIFY		reserved	
MODULE		reserved	reserved
MONTH	non-reserved	reserved	reserved
MORE		non-reserved	non-reserved
MOVE	reserved		
MUMPS		non-reserved	non-reserved
NAME		non-reserved	non-reserved
NAMES	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
NATIONAL	non-reserved	reserved	reserved
NATURAL	reserved	reserved	reserved
NCHAR	reserved	reserved	reserved
NCLOB		reserved	
NEW	reserved	reserved	
NEXT	non-reserved	reserved	reserved
NO	non-reserved	reserved	reserved
NOCREATEDB	non-reserved		
NOCREATEUSER	non-reserved		
NONE	non-reserved	reserved	
NOT	reserved	reserved	reserved
NOTHING	non-reserved		
NOTIFY	non-reserved		
NOTNULL	reserved		
NULL	reserved	reserved	reserved
NULLABLE		non-reserved	non-reserved
NULLIF	reserved	non-reserved	reserved
NUMBER		non-reserved	non-reserved
NUMERIC	reserved	reserved	reserved
OBJECT		reserved	
OCTET_LENGTH		non-reserved	reserved
OF	non-reserved	reserved	reserved
OFF	reserved	reserved	
OFFSET	reserved		
OIDS	non-reserved		
OLD	reserved	reserved	
ON	reserved	reserved	reserved
ONLY	reserved	reserved	reserved
OPEN		reserved	reserved
OPERATION		reserved	
OPERATOR	non-reserved		
OPTION	non-reserved	reserved	reserved
OPTIONS		non-reserved	
OR	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
ORDER	reserved	reserved	reserved
ORDINALITY		reserved	
OUT	reserved	reserved	
OUTER	reserved	reserved	reserved
OUTPUT		reserved	reserved
OVERLAPS	reserved	non-reserved	reserved
OVERLAY		non-reserved	
OVERRIDING		non-reserved	
OWNER	non-reserved		
PAD		reserved	reserved
PARAMETER		reserved	
PARAMETERS		reserved	
PARAMETER_MODE		non-reserved	
PARAMETER_NAME		non-reserved	
PARAMETER_ORDINAL_POSITION		non-reserved	
PARAMETER_SPECIFIC_CATALOG		non-reserved	
PARAMETER_SPECIFIC_NAME		non-reserved	
PARAMETER_SPECIFIC_SCHEMA		non-reserved	
PARTIAL	non-reserved	reserved	reserved
PASCAL		non-reserved	non-reserved
PASSWORD	non-reserved		
PATH	non-reserved	reserved	
PENDANT	non-reserved		
PLI		non-reserved	non-reserved
POSITION	reserved	non-reserved	reserved
POSTFIX		reserved	
PRECISION	reserved	reserved	reserved
PREFIX		reserved	
PREORDER		reserved	
PREPARE		reserved	reserved
PRESERVE		reserved	reserved
PRIMARY	reserved	reserved	reserved
PRIOR	non-reserved	reserved	reserved
PRIVILEGES	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
PROCEDURAL	non-reserved		
PROCEDURE	non-reserved	reserved	reserved
PUBLIC	reserved	reserved	reserved
READ	non-reserved	reserved	reserved
READS		reserved	
REAL		reserved	reserved
RECURSIVE		reserved	
REF		reserved	
REFERENCES	reserved	reserved	reserved
REFERENCING		reserved	
REINDEX	non-reserved		
RELATIVE	non-reserved	reserved	reserved
RENAME	non-reserved		
REPEATABLE		non-reserved	non-reserved
RESET	reserved		
RESTRICT	non-reserved	reserved	reserved
RESULT		reserved	
RETURN		reserved	
RETURNED_LENGTH		non-reserved	non-reserved
RETURNED_OCTET_LENGTH		non-reserved	non-reserved
RETURNED_SQLSTATE		non-reserved	non-reserved
RETURNS	non-reserved	reserved	
REVOKE	non-reserved	reserved	reserved
RIGHT	reserved	reserved	reserved
ROLE		reserved	
ROLLBACK	non-reserved	reserved	reserved
ROLLUP		reserved	
ROUTINE		reserved	
ROUTINE_CATALOG		non-reserved	
ROUTINE_NAME		non-reserved	
ROUTINE_SCHEMA		non-reserved	
ROW	non-reserved	reserved	
ROWS		reserved	reserved
ROW_COUNT		non-reserved	non-reserved

Key Word	PostgreSQL	SQL 99	SQL 92
RULE	non-reserved		
SAVEPOINT		reserved	
SCALE		non-reserved	non-reserved
SCHEMA	non-reserved	reserved	reserved
SCHEMA_NAME		non-reserved	non-reserved
SCOPE		reserved	
SCROLL	non-reserved	reserved	reserved
SEARCH		reserved	
SECOND	non-reserved	reserved	reserved
SECTION		reserved	reserved
SECURITY		non-reserved	
SELECT	reserved	reserved	reserved
SELF		non-reserved	
SENSITIVE		non-reserved	
SEQUENCE	non-reserved	reserved	
SERIAL	non-reserved		
SERIALIZABLE	non-reserved	non-reserved	non-reserved
SERVER_NAME		non-reserved	non-reserved
SESSION	non-reserved	reserved	reserved
SESSION_USER	reserved	reserved	reserved
SET	non-reserved	reserved	reserved
SETOF	reserved		
SETS		reserved	
SHARE	non-reserved		
SHOW	reserved		
SIMILAR		non-reserved	
SIMPLE		non-reserved	
SIZE		reserved	reserved
SMALLINT		reserved	reserved
SOME	reserved	reserved	reserved
SOURCE		non-reserved	
SPACE		reserved	reserved
SPECIFIC		reserved	
SPECIFICTYPE		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
SPECIFIC_NAME		non-reserved	
SQL		reserved	reserved
SQLCODE			reserved
SQLERROR			reserved
SQLEXCEPTION		reserved	
SQLSTATE		reserved	reserved
SQLWARNING		reserved	
START	non-reserved	reserved	
STATE		reserved	
STATEMENT	non-reserved	reserved	
STATIC		reserved	
STDIN	non-reserved		
STDOUT	non-reserved		
STRUCTURE		reserved	
STYLE		non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved
SUBLIST		non-reserved	
SUBSTRING	reserved	non-reserved	reserved
SUM		non-reserved	reserved
SYMMETRIC		non-reserved	
SYSID	non-reserved		
SYSTEM		non-reserved	
SYSTEM_USER		reserved	reserved
TABLE	reserved	reserved	reserved
TABLE_NAME		non-reserved	non-reserved
TEMP	non-reserved		
TEMPLATE	non-reserved		
TEMPORARY	non-reserved	reserved	reserved
TERMINATE		reserved	
THAN		reserved	
THEN	reserved	reserved	reserved
TIME	non-reserved	reserved	reserved
TIMESTAMP	non-reserved	reserved	reserved
TIMEZONE_HOUR	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
TIMEZONE_MINUTE	non-reserved	reserved	reserved
TO	reserved	reserved	reserved
TOAST	non-reserved		
TRAILING	reserved	reserved	reserved
TRANSACTION	reserved	reserved	reserved
TRANSACTIONS_COMMITTED		non-reserved	
TRANSACTIONS_ROLLED_BACK		non-reserved	
TRANSACTION_ACTIVE		non-reserved	
TRANSFORM		non-reserved	
TRANSFORMS		non-reserved	
TRANSLATE		non-reserved	reserved
TRANSLATION		reserved	reserved
TREAT		reserved	
TRIGGER	non-reserved	reserved	
TRIGGER_CATALOG		non-reserved	
TRIGGER_NAME		non-reserved	
TRIGGER_SCHEMA		non-reserved	
TRIM	reserved	non-reserved	reserved
TRUE	reserved	reserved	reserved
TRUNCATE	non-reserved		
TRUSTED	non-reserved		
TYPE	non-reserved	non-reserved	non-reserved
UNCOMMITTED		non-reserved	non-reserved
UNDER		reserved	
UNION	reserved	reserved	reserved
UNIQUE	reserved	reserved	reserved
UNKNOWN		reserved	reserved
UNLISTEN	non-reserved		
UNNAMED		non-reserved	non-reserved
UNNEST		reserved	
UNTIL	non-reserved		
UPDATE	non-reserved	reserved	reserved
UPPER		non-reserved	reserved
USAGE		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
USER	reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG		non-reserved	
USER_DEFINED_TYPE_NAME		non-reserved	
USER_DEFINED_TYPE_SCHEMA		non-reserved	
USING	reserved	reserved	reserved
VACUUM	reserved		
VALID	non-reserved		
VALUE		reserved	reserved
VALUES	non-reserved	reserved	reserved
VARCHAR	reserved	reserved	reserved
VARIABLE		reserved	
VARYING	non-reserved	reserved	reserved
VERBOSE	reserved		
VERSION	non-reserved		
VIEW	non-reserved	reserved	reserved
WHEN	reserved	reserved	reserved
WHENEVER		reserved	reserved
WHERE	reserved	reserved	reserved
WITH	non-reserved	reserved	reserved
WITHOUT	non-reserved	reserved	
WORK	non-reserved	reserved	reserved
WRITE		reserved	reserved
YEAR	non-reserved	reserved	reserved
ZONE	non-reserved	reserved	reserved

Bibliography

Selected references and readings for SQL and Postgres.

Some white papers and technical reports from the original Postgres development team are available at the University of California, Berkeley, Computer Science Department web site (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/)

SQL Reference Books

The Practical SQL Handbook , Bowman et al, 1996 , *Using Structured Query Language* , 3, Judith Bowman, Sandra Emerson, and Marcy Darnovsky, 0-201-44787-8, 1996, Addison-Wesley, 1996.

A Guide to the SQL Standard , Date and Darwen, 1997 , *A user's guide to the standard database language SQL* , 4, C. J. Date and Hugh Darwen, 0-201-96426-0, 1997, Addison-Wesley, 1997.

An Introduction to Database Systems , Date, 1994 , 6, C. J. Date, 1, 1994, Addison-Wesley, 1994.

Understanding the New SQL , Melton and Simon, 1993 , *A complete guide*, Jim Melton and Alan R. Simon, 1-55860-245-3, 1993, Morgan Kaufmann, 1993.

Abstract

Accessible reference for SQL features.

Principles of Database and Knowledge : Base Systems , Ullman, 1988 , Jeffrey D. Ullman, 1, Computer Science Press , 1988 .

PostgreSQL-Specific Documentation

The PostgreSQL Administrator's Guide , The Administrator's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Developer's Guide , The Developer's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide , The Programmer's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction , The Tutorial , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide , The User's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

Enhancement of the ANSI SQL Implementation of PostgreSQL , Simkovics, 1998 , Stefan Simkovics, O.Univ.Prof.Dr.. Georg Gottlob, November 29, 1998, Department of Information Systems, Vienna University of Technology .

Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into Postgres. Prepared as a Master's Thesis with the support of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr at Vienna University of Technology.

The Postgres95 User Manual , Yu and Chen, 1995 , A. Yu and J. Chen, The POSTGRES Group , Sept. 5, 1995, University of California, Berkeley CA.

Proceedings and Articles

Partial indexing in POSTGRES: research project , Olson, 1993 , Nels Olson, 1993, UCB Engin T7.49.1993 O676, University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System , Ong and Goh, 1990 , L. Ong and J. Goh, April, 1990, ERL Technical Memorandum M90/33, University of California, Berkeley CA.

The Postgres Data Model , Rowe and Stonebraker, 1987 , L. Rowe and M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

Generalized partial indexes (http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z) , Seshardri, 1995 , P. Seshadri and A. Swami, March 1995, Eleventh International Conference on Data Engineering, 1995, Cat. No.95CH35724, IEEE Computer Society Press.

The Design of Postgres , Stonebraker and Rowe, 1986 , M. Stonebraker and L. Rowe, May 1986, Conference on Management of Data, Washington DC, ACM-SIGMOD, 1986.

The Design of the Postgres Rules System, Stonebraker, Hanson, Hong, 1987 , M. Stonebraker, E. Hanson, and C. H. Hong, Feb. 1987, Conference on Data Engineering, Los Angeles, CA, IEEE, 1987.

The Postgres Storage System , Stonebraker, 1987 , M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

A Commentary on the Postgres Rules System , Stonebraker et al, 1989, M. Stonebraker, M. Hearst, and S. Potamianos, Sept. 1989, Record 18(3), SIGMOD, 1989.

The case for partial indexes (DBMS) (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf) , Stonebraker, M, 1989b, M. Stonebraker, Dec. 1989, Record 18(no.4):4-11, SIGMOD, 1989.

The Implementation of Postgres , Stonebraker, Rowe, Hirohama, 1990 , M. Stonebraker, L. A. Rowe, and M. Hirohama, March 1990, Transactions on Knowledge and Data Engineering 2(1), IEEE.

On Rules, Procedures, Caching and Views in Database Systems , Stonebraker et al, ACM, 1990 , M. Stonebraker and et al, June 1990, Conference on Management of Data, ACM-SIGMOD.