The Postgres SQL Queries learner, 12beta

- 24 September 2019

Edited and abridged by Gregory Lorriman 2019

PostgreSQL is Copyright © 1996-2019 by the PostgreSQL Global Development Group.

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

This document also Copyright 2019 Gregory Lorriman

# Contents

[Contents 2](#_Toc22831007)

[Legal Notice 5](#_Toc22831008)

[Introduction 6](#_Toc22831009)

[Queries 7](#_Toc22831010)

[Overview 7](#_Toc22831011)

[Table expressions 9](#_Toc22831012)

[7.2.1. The FROM Clause 9](#_Toc22831013)

[7.2.1.1. Joining Tables 9](#_Toc22831014)

[7.2.1.2. Table And Column Aliases 15](#_Toc22831015)

[7.2.1.3. Subqueries 16](#_Toc22831016)

[7.2.1.4. Table Functions (advanced) 16](#_Toc22831017)

[7.2.1.5. LATERAL Subqueries (advanced) 17](#_Toc22831018)

[7.2.2. The WHERE Clause 18](#_Toc22831019)

[7.2.3. The GROUP BY and HAVING Clauses 19](#_Toc22831020)

[7.2.4. GROUPING SETS, CUBE, and ROLLUP 22](#_Toc22831021)

[7.2.5. Window Function Processing (advanced) 25](#_Toc22831022)

[Select Lists 25](#_Toc22831023)

[7.3.1. Select-List Items 25](#_Toc22831024)

[7.3.2. Column Labels 26](#_Toc22831025)

[7.3.3. DISTINCT 26](#_Toc22831026)

[Combining Queries (UNION, INTERSECT, EXCEPT) 27](#_Toc22831027)

[Sorting Rows 27](#_Toc22831028)

[LIMIT and OFFSET 29](#_Toc22831029)

[VALUES Lists 29](#_Toc22831030)

[WITH Queries (Common Table Expressions) 30](#_Toc22831031)

[7.8.1. SELECT in WITH 30](#_Toc22831032)

[Recursion (advanced) 31](#_Toc22831033)

[4.1. Lexical Structure 32](#_Toc22831034)

[4.1.1. Identifiers and Key Words 32](#_Toc22831035)

[4.1.2. Constants 33](#_Toc22831036)

[4.1.2.1. String Constants 33](#_Toc22831037)

[4.1.2.2. String Constants With C-Style Escapes 33](#_Toc22831038)

[Table 4.1. Backslash Escape Sequences 33](#_Toc22831039)

[4.1.2.4. Dollar-Quoted String Constants 34](#_Toc22831040)

[4.1.2.5. Bit-String Constants 34](#_Toc22831041)

[4.1.2.6. Numeric Constants 34](#_Toc22831042)

[4.1.2.7. Constants Of Other Types 35](#_Toc22831043)

[4.1.3. Operators 35](#_Toc22831044)

[4.1.5. Comments 35](#_Toc22831045)

[4.1.6. Operator Precedence 36](#_Toc22831046)

[Table 4.2. Operator Precedence (highest to lowest) 36](#_Toc22831047)

[4.2. Value Expressions 37](#_Toc22831048)

[4.2.1. Column References 37](#_Toc22831049)

[4.2.2. Positional Parameters 37](#_Toc22831050)

[4.2.3. Subscripts 37](#_Toc22831051)

[4.2.4. Field Selection 38](#_Toc22831052)

[4.2.5. Operator Invocations 38](#_Toc22831053)

[4.2.6. Function Calls 39](#_Toc22831054)

[4.2.7. Aggregate Expressions 39](#_Toc22831055)

[Ordered Set Aggregates 40](#_Toc22831056)

[4.2.8. Window Function Calls (advanced SQL) 41](#_Toc22831057)

[4.2.9. Type Casts (Advanced SQL) 44](#_Toc22831058)

[4.2.10. Collation Expressions (advanced SQL, usually for administrators) 45](#_Toc22831059)

[4.2.12. Array Constructors (Advanced SQL) 46](#_Toc22831060)

[4.2.13. Row Constructors (Advanced SQL) 48](#_Toc22831061)

[4.2.14. Expression Evaluation Rules 48](#_Toc22831062)

[Selected Types 50](#_Toc22831063)

[Table of numeric types 51](#_Toc22831064)

[Table 8.2. Numeric Types 51](#_Toc22831065)

[8.1.1. Integer Types 51](#_Toc22831066)

[8.1.2. Arbitrary Precision Numbers (*numeric* type) 51](#_Toc22831067)

[8.1.3. Floating-Point Types 53](#_Toc22831068)

[8.1.4. Serial Types 54](#_Toc22831069)

[8.2. Monetary Types 54](#_Toc22831070)

[Table 8.3. Monetary Types 54](#_Toc22831071)

[8.3. Character Types 55](#_Toc22831072)

[Table 8.4. Character Types 55](#_Toc22831073)

[Table 8.5. Special Character Types 57](#_Toc22831074)

[8.4. Binary Data Types (advanced SQL) 57](#_Toc22831075)

[Table 8-6. Binary Data Types 57](#_Toc22831076)

[Table 8-7. bytea Literal Escaped Octets (to aid recognition in 3rd party query SQL) 57](#_Toc22831077)

[8.5. Date/Time Types 57](#_Toc22831078)

[Table 8-9. Date/Time Types 58](#_Toc22831079)

[Table 8-10. Date Input 60](#_Toc22831080)

[Table 8-11. Time Input 60](#_Toc22831081)

[Table 8-12. Time Zone Input 60](#_Toc22831082)

[Table 8-16. ISO 8601 Interval Unit Abbreviations 65](#_Toc22831083)

[Table 8-17. Interval Input 66](#_Toc22831084)

[Table 8-18. Interval Output Style Examples 67](#_Toc22831085)

# Legal Notice

PostgreSQL is Copyright © 1996-2019 by the PostgreSQL Global Development Group.

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

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

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

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

In addition, and in respect of modifications to the original copyrighted documentation made by Gregory Lorriman, the following notice must be included with all copies of this modified documentation :

ALL ADDITIONS TO THE SOURCE DOCUMENTATION ARE COPYRIGHT GREGORY LORRIMAN 2019. Permission to use, copy, modify, and distribute additions to this documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notices and this paragraph and the following three paragraphs appear in all copies.

GREGORY LORRIMAN GIVES NO ASSURANCE OF CORRECTNESS OR FITNESS FOR PURPOSE OF THIS NON-PROFESSIONALLY MODIFIED DOCUMENTATION.

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

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

# Introduction

(A rapid introduction to Postgres SQL and database can be profitably gained here:

<http://postgresguide.com/> )

This is an intermediate-level guide for people wishing to query an SQL database, perhaps to extract data for a spreadsheet.

The intention is an accelerated learning of practical SQL for queriers/analysts not database administrators or application developers. As such, 90% of the postgres documentation has been left out leaving only that pertinent to querying.

After reading this document, you should then familiarise yourself with builtin functions, here: <https://www.postgresql.org/docs/12/functions.html> to avoid undue extra work when composing your queries.

This learner guide goes in to detail while briefly overviewing and avoiding detail on a few advanced topics, marked ‘advanced’.

**Notes on this fork of the PostgreSQL documentation:**

It is a subset of the official ‘PostgreSQL 12.0 Beta Documentation’ with some small additions and editing, the release version of which is available here: <https://www.postgresql.org/docs/12/index.html>

To avoid undue burdens to the learner, advanced SQL query features are touched on with examples for the purpose of recognition but the bulk of their explanatory text has been removed. These sections are marked with ‘advanced’. Advanced SQL is not necessary for most queries, and is usually of more interest to the experienced SQL user not the learner.

**Original intro text, edited :**

The information is arranged so that a novice user can follow it start to end to gain a full understanding of the topics without having to refer forward too many times. The chapters are intended to be self-contained, so that advanced users can read the chapters individually as they choose. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should see [**Part VI**](https://www.postgresql.org/docs/12/reference.html).

# Queries

## Overview

(Sections marked ‘Advanced’ have been severely cut-down. Refer to the original documentation to read details on these, here <https://www.postgresql.org/docs/12/index.html> .)

The process of retrieving or the command to retrieve data from a database is called a *query*. In SQL the [**SELECT**](https://www.postgresql.org/docs/12/sql-select.html) command is used to specify queries.

(Databases are a collection of tables, with each table consisting of rows and columns of data. Unlike spreadsheets they are highly regular and tightly managed for efficiency, integrity and performance for multiple client access, while spreadsheets are designed for adhoc design flexibility combined with a fundamental simplicity and accessibility, for one person. SQL databases meanwhile are only simple in the sense of having row and column tables.)

A typical result of a query is to produce a temporary/virtual table of rows and columns from a concrete source table. But some types of SELECT command can also produce a table with 1 column and 1 row, just a single value.

The general syntax of the SELECT command is:

[WITH ***with\_queries***] SELECT ***select\_list*** FROM ***table\_expression*** [***sort\_specification***] ;

Square brackets denote optional parts. Termination with a semi-colon is not needed for a simple query.

Put another way :

SELECT *chosen\_columns* FROM *source\_table(s)* WHERE *expression\_to\_match\_rows*;

*chosen\_columns* : column names from the source table to be included in the result table

*source\_table\_names*: the table(s) to fetch rows from

*expression\_to\_match\_rows*: some kind of comparison to choose which rows go into the result table

Notes: column names are also known as ‘fieldnames’ in database lingo. Fieldname/column ‘expressions’ are typically merely column names from the source tables to be included in the result table, but powerfully can also be new, calculated expressions based off the source fields, example below. The WHERE clause is optional and left out if you want all the source rows in the result.

Simple example:

SELECT name, price

FROM product\_table

In a table with only 3 products, might result in:

name | price

------------------------------

stapler | $40

matches | $0.20

printer | $500

Example (with calculation, renaming, and comparison) :

SELECT name as product\_name, price , (price \* 20%) as tax

FROM product\_table

WHERE price<$50 ;

The result of the second example would be:

product\_name | price | tax

------------------------------

stapler | $40 | $8

matches | $0.20 | $0.04

The following sections describe the details of the select list, the table expression, and the sort specification. WITH queries are treated last since they are an advanced feature.

A simple kind of query has the form:

SELECT \* FROM table1;

Where ‘\*’ is the character used in standard SQL to include all the fields from the source table in the resulting temporary table.

Assuming that there is a table called table1, this command would retrieve all rows and all user-defined columns from table1. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, while client libraries will offer functions to extract individual values from the query result.) The select list specification \* means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if table1 has columns named a, b, and c (and perhaps others) you can make the following query:

SELECT a, b + c FROM table1;

(assuming that b and c are of a numerical data type). See [**Section 7.3**](https://www.postgresql.org/docs/12/queries-select-lists.html) for more details.

FROM table1 is a simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the SELECT command as a calculator:

SELECT 3 \* 4;

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

SELECT random();

## Table expressions

A *table expression* computes a table. The table expression contains a FROM clause that is optionally followed by WHERE, GROUP BY, and HAVING clauses. Trivial table expressions simply refer to a table on disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. All these transformations produce a virtual/temporary.

## 7.2.1. The FROM Clause

The [FROM **Clause**](https://www.postgresql.org/docs/12/sql-select.html#SQL-FROM) derives a table from one or more other tables given in a comma-separated table reference list.

FROM ***table\_reference*** [, ***table\_reference*** [, ...]]

A table reference can be a table name (possibly schema-qualified), or a derived table such as a subquery, a JOIN construct, or complex combinations of these. The result of the FROM list is an intermediate/temporary table that can then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses, to produce the final temporary table.

(Advanced: When a table is the parent of a table inheritance hierarchy, the table reference produces rows/data of not only that table but all of its descendant tables, unless the key word ONLY precedes the table name. However, the reference produces only the columns that appear in the named table — extra columns in sub/descendent tables are ignored.)

## 7.2.1.1. Joining Tables

Introduction

A joined table is a table derived from two other tables joined together according to the rules of the particular join type.

(Joining is not the same as combining similarly structured tables using UNION, INTERSECT, EXCEPT. See section 7.3.3)

Joining is a crucial query ability in a relational/SQL DBMS, but requires a small amount of RDBMS theory and practice to understand.

As an example of how RDBMes conceptually link tables: typically an Employee table will have a ‘id’ column of unique numbers so that each employee/row can be uniquely identified (often the ‘id‘ is even printed on letters as a reference number etc). An Addresses table may likewise have a unique id to identify individual address rows, if needed, but more importantly it will also have a column named employee\_id storing the employee’s id from the Employee table, which therefore links each address row to an employee row. This is how ‘relations’ are typically constructed in RDBMSs.

SQL knows nothing of your database design intentions, however, so queries combining two tables require a ‘join’ clause to explicitly match/link columns from two tables, ie, spelling out that the Employee.id column matches the Addresses.employee\_id column to link the rows, and so to produce a result of both employee data and their addresses in one (temporary) table.

Now on to the details…..

The default (and most common) join is ‘inner’, but in the Postgres docs ‘cross join’, the least common, is given first.

The general syntax of a table joining is

***T1*** ***join\_type*** ***T2*** [ ***join\_condition*** ]

Joins can be chained together, or nested: either or both ***T1*** and ***T2*** can be themselves the result of a join. Parentheses can be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

**Join Types**

Cross join

***T1*** CROSS JOIN ***T2***

For every possible combination of rows from ***T1*** and ***T2*** (i.e., a Cartesian product), the joined table will contain a row consisting of all columns in ***T1*** followed by all columns in ***T2***. If the tables have N and M rows respectively, the joined table will have N \* M rows. Not often useful.

Practical example of a cross-join:

SELECT specId, week\_number

FROM standard\_reports CROSS JOIN weeks

FROM ***T1*** CROSS JOIN ***T2*** is equivalent to FROM ***T1***, ***T2***.

Qualified joins :

***T1*** INNER JOIN ***T2*** ON ***boolean\_expression***

***T1*** INNER JOIN ***T2*** USING ( ***join column list*** )

***T1*** NATURAL INNER JOIN ***T2***

INNER is the default (and most commonly useful).

The words INNER and OUTER are optional in all forms. INNER is the default; LEFT, RIGHT, and FULL imply an outer join.

The *join condition* is specified in the ON or USING clause, or implicitly by the word NATURAL (explained below). The join condition determines which rows from the two source tables are considered to “match”.

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.

Example:

SELECT name, postcode

FROM Employees INNER JOIN Addresses

ON Employees.id=Addresses.employee\_id

The resulting table will not include employees who do not have an address as the join condition must match to be included in the result table. To include employees without an address, use an OUTER join, as follows…

LEFT OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.

Example:

SELECT name, postcode

FROM Employees LEFT OUTER JOIN Addresses

ON Employees.id=Addresses.employee\_id

..will result in a table that also includes employees with no address and for which the postcode will be ‘null’.

RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will always have a row for each row in T2.

Whether to use LEFT or RIGHT is purely down to the order in which the tables are referenced in the query; there is no conceptual difference.

FULL OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from ***T1*** and ***T2*** match if the ON expression evaluates to true.

The USING clause is a convenience when both sides of the join use the same name for the joining column(s). It takes a comma-separated list of the shared column names and forms a join condition that uses an equality comparison for each one. For example, joining ***T1*** and ***T2*** with USING (a, b) produces the equivalent to ON ***T1***.a = ***T2***.a AND ***T1***.b = ***T2***.b.

Furthermore, the output of JOIN USING suppresses redundant columns: there is no need to print both of the matched columns, since they must have equal values. While JOIN ON produces all columns from ***T1*** followed by all columns from ***T2***, JOIN USING produces one output column for each of the listed column pairs (in the listed order), followed by any remaining columns from ***T1***, followed by any remaining columns from ***T2***.

Finally, NATURAL is a shorthand form of USING: it forms a USING list consisting of all column names that appear in both input tables. As with USING, these columns appear only once in the output table. If there are no common column names, NATURAL JOIN behaves like JOIN ... ON TRUE, producing a cross-product join.

**Note**

USING is reasonably safe from column changes in the joined relations since only the listed columns are combined. NATURAL is risky for anything other than one-off queries, as column names could change.

To put this together, assume we have tables t1:

num | name

-----+------

1 | a

2 | b

3 | c

and t2:

num | value

-----+-------

1 | xxx

3 | yyy

5 | zzz

then we get the following results for the various joins:

=> **SELECT \* FROM t1 CROSS JOIN t2;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

1 | a | 3 | yyy

1 | a | 5 | zzz

2 | b | 1 | xxx

2 | b | 3 | yyy

2 | b | 5 | zzz

3 | c | 1 | xxx

3 | c | 3 | yyy

3 | c | 5 | zzz

(9 rows)

=> **SELECT \* FROM t1 INNER JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

3 | c | 3 | yyy

(2 rows)

=> **SELECT \* FROM t1 INNER JOIN t2 USING (num);**

num | name | value

-----+------+-------

1 | a | xxx

3 | c | yyy

(2 rows)

=> **SELECT \* FROM t1 NATURAL INNER JOIN t2;**

num | name | value

-----+------+-------

1 | a | xxx

3 | c | yyy

(2 rows)

=> **SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

2 | b | |

3 | c | 3 | yyy

(3 rows)

=> **SELECT \* FROM t1 LEFT JOIN t2 USING (num);**

num | name | value

-----+------+-------

1 | a | xxx

2 | b |

3 | c | yyy

(3 rows)

=> **SELECT \* FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

3 | c | 3 | yyy

| | 5 | zzz

(3 rows)

=> **SELECT \* FROM t1 FULL JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

2 | b | |

3 | c | 3 | yyy

| | 5 | zzz

(4 rows)

The join condition specified with ON can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

=> **SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

2 | b | |

3 | c | |

(3 rows)

Notice that placing the restriction in the WHERE clause produces a different result:

=> **SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

(1 row)

This is because a restriction placed in the ON clause is processed *before* the join, while a restriction placed in the WHERE clause is processed *after* the join. That does not matter with inner joins, but it matters with outer joins.

## 7.2.1.2. Table And Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a *table alias*.

To create a table alias, write

FROM ***table\_reference*** AS ***alias***

or

FROM ***table\_reference*** ***alias***

The AS key word is optional. ***alias*** can be any identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

SELECT \* FROM some\_very\_long\_table\_name s JOIN another\_fairly\_long\_name a ON s.id = a.num;

The alias becomes the new name of the table reference so far as the current query is concerned — it is not allowed to refer to the table by the original name elsewhere in the query. Thus, this is not valid:

SELECT \* FROM my\_table AS m WHERE my\_table.a > 5; -- wrong

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

SELECT \* FROM people AS mother JOIN people AS child ON mother.id = child.mother\_id;

Additionally, an alias is required if the table reference is a subquery (see [**Section 7.2.1.3**](https://www.postgresql.org/docs/12/queries-table-expressions.html#QUERIES-SUBQUERIES)).

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias b to the second instance of my\_table, but the second statement assigns the alias to the result of the join:

SELECT \* FROM my\_table AS a CROSS JOIN my\_table AS b ...

SELECT \* FROM (my\_table AS a CROSS JOIN my\_table) AS b ...

Another form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

FROM ***table\_reference*** [AS] ***alias*** ( ***column1*** [, ***column2*** [, ...]] )

If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, the alias hides the original name(s) within the JOIN. For example:

SELECT a.\* FROM my\_table AS a JOIN your\_table AS b ON ...

is valid SQL, but:

SELECT a.\* FROM (my\_table AS a JOIN your\_table AS b ON ...) AS c

is not valid; the table alias a is not visible outside the alias c.

## 7.2.1.3. Subqueries

A subquery is a query within an overall query. The subquery is executed first and may create a virtual/temporary table which can then be joined to in the overall query (but can be used for other purposes, not only joins).

Subqueries that result in a virtual table must be enclosed in parentheses and *must* be assigned a table alias name (as in [**Section 7.2.1.2**](https://www.postgresql.org/docs/12/queries-table-expressions.html#QUERIES-TABLE-ALIASES)). For a contrived example:

FROM (SELECT \* FROM table1) AS alias\_name

This example is equivalent to FROM table1 AS alias\_name. Other cases, which cannot be reduced to a plain join, arise when the subquery involves grouping or aggregation.

A subquery can also be a VALUES list:

FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))

AS names(first, last)

Again, a table alias is required. Assigning alias names to the columns of the VALUES list is optional, but is good practice. For more information see [**Section 7.7**](https://www.postgresql.org/docs/12/queries-values.html).

## 7.2.1.4. Table Functions (advanced)

Table functions are functions that produce a set of rows, rather like a virtual table from a subquery, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view\*, or subquery in the FROM clause of a query. Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as columns of a table, view, or subquery. You can wrap a query, that would otherwise have been a subquery, in a function, and use it where you would have written a subquery.

(\*a *view* is a permanent virtual table, often read-only, that is defined by a live query, and so the view table changes with changes to the underlying tables. They are created by developers and database admins, but you may also have permission to create views for your convenience. It behaves like a concrete/normal table as far as we are concerned for data querying purposes. Normal (non-view) queries that we create produce temporary virtual tables, which are not live, which are read and then immediately dissolved on closing the query. )

If column aliases are not supplied, then for a function returning a base data type, the column name is also the same as the function name. For a function returning a composite type, the result columns get the names of the individual attributes of the type.

Some examples:

CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$

SELECT \* FROM foo WHERE fooid = $1;

$$ LANGUAGE SQL;

[the $1 is substituted with the int value when the function is called. Had there been a second ‘parameter’ it would have been named $2]

Calling the function :

SELECT \* FROM getfoo(1) AS t1;

[effectively this function is used instead of a subquery, making the SQL more readable but also can be used in other SQL queries.]

SELECT \* FROM foo

WHERE foosubid IN (

SELECT foosubid

FROM getfoo(foo.fooid) z

WHERE z.fooid = foo.fooid

);

[note: remember that AS for the table alias is optional, and left out here for z]

## 7.2.1.5. LATERAL Subqueries (advanced)

Subqueries appearing in FROM can be preceded by the key word LATERAL. This allows them to reference columns provided by preceding FROM items. (Without LATERAL, each subquery is evaluated independently and so cannot cross-reference any other FROM item.)

Table functions appearing in FROM can also be preceded by the key word LATERAL, but for functions the key word is optional; the function's arguments can contain references to columns provided by preceding FROM items in any case.

A LATERAL item can appear at top level in the FROM list, or within a JOIN tree. In the latter case it can also refer to any items that are on the left-hand side of a JOIN that it is on the right-hand side of.

A trivial example of LATERAL is

SELECT \* FROM foo, LATERAL (SELECT \* FROM bar WHERE bar.id = foo.bar\_id) ss;

This is not especially useful since it has exactly the same result as the more conventional

SELECT \* FROM foo, bar WHERE bar.id = foo.bar\_id;

LATERAL is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined.

It is often particularly handy to LEFT JOIN to a LATERAL subquery, so that source rows will appear in the result even if the LATERAL subquery produces no rows for them. For example, if get\_product\_names() returns the names of products made by a manufacturer, but some manufacturers in our table currently produce no products, we could find out which ones those are like this:

SELECT m.name

FROM manufacturers m LEFT JOIN LATERAL get\_product\_names(m.id) pname ON true

WHERE pname IS NULL;

A common application is providing an argument value for a set-returning function.

## 7.2.2. The WHERE Clause

The syntax of the [WHERE **Clause**](https://www.postgresql.org/docs/12/sql-select.html#SQL-WHERE) is

WHERE ***search\_condition***

where ***search\_condition*** is any value expression (see [**Section 4.2**](https://www.postgresql.org/docs/12/sql-expressions.html)) that returns a value of type boolean.

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

**Note**

The join condition of an inner join can be written either in the WHERE clause or in the JOIN clause. For example, these table expressions are equivalent:

FROM a, b WHERE a.id = b.id AND b.val > 5

and:

FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5

or perhaps even:

FROM a NATURAL JOIN b WHERE b.val > 5

Which one of these you use is mainly a matter of style. The JOIN syntax in the FROM clause is probably not as portable to other SQL database management systems, even though it is in the SQL standard. For outer joins there is no choice: they must be done in the FROM clause. The ON or USING clause of an outer join is *not* equivalent to a WHERE condition, because it results in the addition of rows (for unmatched input rows) as well as the removal of rows in the final result.

Here are some examples of WHERE clauses:

SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)

fdt is the table derived in the FROM clause. Rows that do not meet the search condition of the WHERE clause are eliminated from fdt. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how fdt is referenced in the subqueries. Qualifying c1 as fdt.c1 is only necessary if c1 is also the name of a column in the derived input table of the subquery. But qualifying the column name adds clarity even when it is not needed. This example shows how the column naming scope of an outer query extends into its inner queries [then why LATERAL?].

## 7.2.3. The GROUP BY and HAVING Clauses

After passing the WHERE filter, the derived input table might be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

SELECT ***select\_list***

FROM ...

[WHERE ...]

GROUP BY ***grouping\_column\_reference*** [, ***grouping\_column\_reference***]...

The [GROUP BY **Clause**](https://www.postgresql.org/docs/12/sql-select.html#SQL-GROUPBY) is used to group together those rows in a table that have the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to combine each set of rows having common values into one group row that represents all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups. For instance:

=> **SELECT \* FROM test1;**

x | y

---+---

a | 3

c | 2

b | 5

a | 1

(4 rows)

=> **SELECT x FROM test1 GROUP BY x;**

x

---

a

b

c

(3 rows)

In the second query, we could not have written SELECT \* FROM test1 GROUP BY x, because there is no single value for the column y that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a single value in each group.

In general, if a table is grouped, columns that are not listed in GROUP BY cannot be referenced except in aggregate expressions. An example with aggregate expressions is:

=> **SELECT x, sum(y) FROM test1 GROUP BY x;**

x | sum

---+-----

a | 4

b | 5

c | 2

(3 rows)

Here sum is an aggregate function that computes a single value over the entire group. More information about the available aggregate functions can be found in [**Section 9.20**](https://www.postgresql.org/docs/12/functions-aggregate.html).

**Tip**

Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the DISTINCT clause (see [**Section 7.3.3**](https://www.postgresql.org/docs/12/queries-select-lists.html#QUERIES-DISTINCT)).

Here is another example: it calculates the total sales for each product (rather than the total sales of all products):

SELECT product\_id, p.name, (sum(s.units) \* p.price) AS sales

FROM products p LEFT JOIN sales s USING (product\_id)

GROUP BY product\_id, p.name, p.price;

In this example, the columns product\_id, p.name, and p.price must be in the GROUP BY clause since they are referenced in the query select list (but see below). The column s.units does not have to be in the GROUP BY list since it is only used in an aggregate expression (sum(...)), which represents the sales of a product. For each product, the query returns a summary row about all sales of the product.

If the products table is set up so that, say, product\_id is the primary key, then it would be enough to group by product\_id in the above example, since name and price would be *functionally dependent* on the product ID, and so there would be no ambiguity about which name and price value to return for each product ID group.

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

If a table has been grouped using GROUP BY, but only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from the result. The syntax is:

SELECT ***select\_list*** FROM ... [WHERE ...] GROUP BY ... HAVING ***boolean\_expression***

Expressions in the HAVING clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Example:

=> **SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;**

x | sum

---+-----

a | 4

b | 5

(2 rows)

=> **SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';**

x | sum

---+-----

a | 4

b | 5

(2 rows)

Again, a more realistic example:

SELECT product\_id, p.name, (sum(s.units) \* (p.price - p.cost)) AS profit

FROM products p LEFT JOIN sales s USING (product\_id)

WHERE s.date > CURRENT\_DATE - INTERVAL '4 weeks'

GROUP BY product\_id, p.name, p.price, p.cost

HAVING sum(p.price \* s.units) > 5000;

In the example above, the WHERE clause is selecting rows by a column that is not grouped (the expression is only true for sales during the last four weeks), while the HAVING clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.

If a query contains aggregate function calls, but no GROUP BY clause, grouping still occurs: the result is a single group row (or perhaps no rows at all, if the single row is then eliminated by HAVING). The same is true if it contains a HAVING clause, even without any aggregate function calls or GROUP BY clause.

## 7.2.4. GROUPING SETS, CUBE, and ROLLUP

More complex grouping operations than those described above are possible using the concept of *grouping sets*. The data selected by the FROM and WHERE clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple GROUP BY clauses, and then the results returned. For example:

=> **SELECT \* FROM items\_sold;**

brand | size | sales

-------+------+-------

Foo | L | 10

Foo | M | 20

Bar | M | 15

Bar | L | 5

(4 rows)

=> **SELECT brand, size, sum(sales) FROM items\_sold GROUP BY GROUPING SETS ((brand), (size), ());**

brand | size | sum

-------+------+-----

Foo | | 30

Bar | | 20

| L | 15

| M | 35

| | 50

(5 rows)

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause. An empty grouping set means that all rows are aggregated down to a single group (which is output even if no input rows were present), as described above for the case of aggregate functions with no GROUP BY clause.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear. To distinguish which grouping a particular output row resulted from, see [**Table 9.59**](https://www.postgresql.org/docs/12/functions-aggregate.html#FUNCTIONS-GROUPING-TABLE).

A shorthand notation is provided for specifying two common types of grouping set. A clause of the form

ROLLUP ( ***e1***, ***e2***, ***e3***, ... )

represents the given list of expressions and all prefixes of the list including the empty list; thus it is equivalent to

GROUPING SETS (

( ***e1***, ***e2***, ***e3***, ... ),

...

( ***e1***, ***e2*** ),

( ***e1*** ),

( )

)

This is commonly used for analysis over hierarchical data; e.g. total salary by department, division, and company-wide total.

A clause of the form

CUBE ( ***e1***, ***e2***, ... )

represents the given list and all of its possible subsets (i.e. the power set). Thus

CUBE ( a, b, c )

is equivalent to

GROUPING SETS (

( a, b, c ),

( a, b ),

( a, c ),

( a ),

( b, c ),

( b ),

( c ),

( )

)

The individual elements of a CUBE or ROLLUP clause may be either individual expressions, or sublists of elements in parentheses. In the latter case, the sublists are treated as single units for the purposes of generating the individual grouping sets. For example:

CUBE ( (a, b), (c, d) )

is equivalent to

GROUPING SETS (

( a, b, c, d ),

( a, b ),

( c, d ),

( )

)

and

ROLLUP ( a, (b, c), d )

is equivalent to

GROUPING SETS (

( a, b, c, d ),

( a, b, c ),

( a ),

( )

)

The CUBE and ROLLUP constructs can be used either directly in the GROUP BY clause, or nested inside a GROUPING SETS clause. If one GROUPING SETS clause is nested inside another, the effect is the same as if all the elements of the inner clause had been written directly in the outer clause.

If multiple grouping items are specified in a single GROUP BY clause, then the final list of grouping sets is the cross product of the individual items. For example:

GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))

is equivalent to

GROUP BY GROUPING SETS (

(a, b, c, d), (a, b, c, e),

(a, b, d), (a, b, e),

(a, c, d), (a, c, e),

(a, d), (a, e)

)

**Note**

The construct (a, b) is normally recognized in expressions as a [**row constructor**](https://www.postgresql.org/docs/12/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). Within the GROUP BY clause, this does not apply at the top levels of expressions, and (a, b) is parsed as a list of expressions as described above. If for some reason you *need* a row constructor in a grouping expression, use ROW(a, b).

## 7.2.5. Window Function Processing (advanced)

If the query contains any window functions (see [**Section 3.5**](https://www.postgresql.org/docs/12/tutorial-window.html), [**Section 9.21**](https://www.postgresql.org/docs/12/functions-window.html) and [**Section 4.2.8**](https://www.postgresql.org/docs/12/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS)), these functions are evaluated after any grouping, aggregation, and HAVING filtering is performed. That is, if the query uses any aggregates, GROUP BY, or HAVING, then the rows seen by the window functions are the group rows instead of the original table rows from FROM/WHERE.

## Select Lists

As shown in the previous section, the table expression in the SELECT command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which *columns* of the intermediate table are actually output.

## 7.3.1. Select-List Items

The simplest kind of select list is \* which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in [**Section 4.2**](https://www.postgresql.org/docs/12/sql-expressions.html)). For instance, it could be a list of column names:

SELECT a, b, c FROM ...

The columns names a, b, and c are either the actual names of the columns of tables referenced in the FROM clause, or the aliases given to them as explained in [**Section 7.2.1.2**](https://www.postgresql.org/docs/12/queries-table-expressions.html#QUERIES-TABLE-ALIASES). The name space available in the select list is the same as in the WHERE clause, unless grouping is used, in which case it is the same as in the HAVING clause.

If more than one table has a column of the same name, the table name must also be given, as in:

SELECT tbl1.a, tbl2.a, tbl1.b FROM ...

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

SELECT tbl1.\*, tbl2.a FROM ...

See [**Section 8.16.5**](https://www.postgresql.org/docs/12/rowtypes.html#ROWTYPES-USAGE) for more about the ***table\_name***.\* notation.

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they can be constant arithmetic expressions, for instance.

## 7.3.2. Column Labels

The entries in the select list can be assigned names for subsequent processing, such as for use in an ORDER BY clause or for display by the client application. For example:

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

If no output column name is specified using AS, the system assigns a default column name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

The AS keyword is optional, but only if the new column name does not match any PostgreSQL keyword (see [**Appendix C**](https://www.postgresql.org/docs/12/sql-keywords-appendix.html)). To avoid an accidental match to a keyword, you can double-quote the column name. For example, VALUE is a keyword, so this does not work:

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

but this does:

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

For protection against possible future keyword additions, it is recommended that you always either write AS or double-quote the output column name.

**Note**

The naming of output columns here is different from that done in the FROM clause (see [**Section 7.2.1.2**](https://www.postgresql.org/docs/12/queries-table-expressions.html#QUERIES-TABLE-ALIASES)). It is possible to rename the same column twice, but the name assigned in the select list is the one that will be passed on.

## 7.3.3. DISTINCT

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

SELECT DISTINCT ***select\_list*** ...

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

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

SELECT DISTINCT ON (***expression*** [, ***expression*** ...]) ***select\_list*** ...

Here ***expression*** is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the “first row” of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ON processing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subqueries in FROM, this construct can be avoided, but it is often the most convenient alternative.

## Combining Queries (UNION, INTERSECT, EXCEPT)

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

***query1*** UNION [ALL] ***query2***

***query1*** INTERSECT [ALL] ***query2***

***query1*** EXCEPT [ALL] ***query2***

***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 is executed as:

(***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 duplicate rows from its result, in the same way as DISTINCT, unless UNION ALL is used.

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

EXCEPT returns all rows that are in the result of ***query1*** but not in the result of ***query2***. (This is sometimes called the *difference* between two queries.) Again, duplicates are eliminated unless EXCEPT ALL is used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be “union compatible”, which means that they return the same number of columns and the corresponding columns have compatible data types, as described in [**Section 10.5**](https://www.postgresql.org/docs/12/typeconv-union-case.html).

## 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 an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The ORDER BY clause specifies the sort order:

SELECT ***select\_list***

FROM ***table\_expression***

ORDER BY ***sort\_expression1*** [ASC | DESC] [NULLS { FIRST | LAST }]

[, ***sort\_expression2*** [ASC | DESC] [NULLS { FIRST | LAST }] ...]

The sort expression(s) can be any expression that would be valid in the query's select list. An example is:

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

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression can be followed by an optional ASC or DESC keyword to set the sort direction to ascending or descending. ASC order is the default. Ascending order puts smaller values first, where “smaller” is defined in terms of the < operator. Similarly, descending order is determined with the > operator. [**[5]**](https://www.postgresql.org/docs/12/queries-order.html#ftn.id-1.5.6.9.5.10)

The NULLS FIRST and NULLS LAST options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if larger than any non-null value; that is, NULLS FIRST is the default for DESC order, and NULLS LAST otherwise.

Note that the ordering options are considered independently for each sort column. For example ORDER BY x, y DESC means ORDER BY x ASC, y DESC, which is not the same as ORDER BY x DESC, y DESC.

A ***sort\_expression*** can also be the column label or number of an output column, as in:

SELECT a + b AS sum, c FROM table1 ORDER BY sum;

SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;

both of which sort by the first output column. Note that an output column name has to stand alone, that is, it cannot be used in an expression — for example, this is *not* correct:

SELECT a + b AS sum, c FROM table1 ORDER BY sum + c; -- wrong

This restriction is made to reduce ambiguity. There is still ambiguity if an ORDER BY item is a simple name that could match either an output column name or a column from the table expression. The output column is used in such cases. This would only cause confusion if you use AS to rename an output column to match some other table column's name.

ORDER BY can be applied to the result of a UNION, INTERSECT, or EXCEPT combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

[**[5]**](https://www.postgresql.org/docs/12/queries-order.html#id-1.5.6.9.5.10) Actually, PostgreSQL uses the *default B-tree operator class* for the expression's data type to determine the sort ordering for ASC and DESC. Conventionally, data types will be set up so that the < and > operators correspond to this sort ordering, but a user-defined data type's designer could choose to do something different.

## LIMIT and OFFSET

LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query:

SELECT ***select\_list***

FROM ***table\_expression***

[ ORDER BY ... ]

[ LIMIT { ***number*** | ALL } ] [ OFFSET ***number*** ]

If a limit count is given, no more than that many rows will be returned (but possibly fewer, if the query itself yields fewer rows). LIMIT ALL is the same as omitting the LIMIT clause, as is LIMIT with a NULL argument.

OFFSET says to skip that many rows before beginning to return rows. OFFSET 0 is the same as omitting the OFFSET clause, as is OFFSET with a NULL argument.

If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is important to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows. You might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating query plans, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

The rows skipped by an OFFSET clause still have to be computed inside the server; therefore a large OFFSET might be inefficient.

## VALUES Lists

VALUES provides a way to generate a “constant table” that can be used in a query without having to actually create and populate a table on-disk. The syntax is

VALUES ( ***expression*** [, ...] ) [, ...]

Each parenthesized list of expressions generates a row in the table. The lists must all have the same number of elements (i.e., the number of columns in the table), and corresponding entries in each list must have compatible data types. The actual data type assigned to each column of the result is determined using the same rules as for UNION (see [**Section 10.5**](https://www.postgresql.org/docs/12/typeconv-union-case.html)).

As an example:

VALUES (1, 'one'), (2, 'two'), (3, 'three');

will return a table of two columns and three rows. It's effectively equivalent to:

SELECT 1 AS column1, 'one' AS column2

UNION ALL

SELECT 2, 'two'

UNION ALL

SELECT 3, 'three';

By default, PostgreSQL assigns the names column1, column2, etc. to the columns of a VALUES table. The column names are not specified by the SQL standard and different database systems do it differently, so it's usually better to override the default names with a table alias list, like this:

=> SELECT \* FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);

num | letter

-----+--------

1 | one

2 | two

3 | three

(3 rows)

Syntactically, VALUES followed by expression lists is treated as equivalent to:

SELECT ***select\_list*** FROM ***table\_expression***

and can appear anywhere a SELECT can. For example, you can use it as part of a UNION, or attach a ***sort\_specification*** (ORDER BY, LIMIT, and/or OFFSET) to it. VALUES is most commonly used as the data source in an INSERT command, and next most commonly as a subquery.

For more information see [**VALUES**](https://www.postgresql.org/docs/12/sql-values.html).

## WITH Queries (Common Table Expressions)

[**7.8.1.** SELECT **in** WITH](https://www.postgresql.org/docs/12/queries-with.html#QUERIES-WITH-SELECT)

[**7.8.2. Data-Modifying Statements in** WITH](https://www.postgresql.org/docs/12/queries-with.html#QUERIES-WITH-MODIFYING)

WITH provides a way to write auxiliary statements for use in a larger query. These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE.

## 7.8.1. SELECT in WITH

The basic value of SELECT in WITH is to break down complicated queries into simpler parts. An example is:

WITH regional\_sales AS (

SELECT region, SUM(amount) AS total\_sales

FROM orders

GROUP BY region

), top\_regions AS (

SELECT region

FROM regional\_sales

WHERE total\_sales > (SELECT SUM(total\_sales)/10 FROM regional\_sales)

)

SELECT region,

product,

SUM(quantity) AS product\_units,

SUM(amount) AS product\_sales

FROM orders

WHERE region IN (SELECT region FROM top\_regions)

GROUP BY region, product;

which displays per-product sales totals in only the top sales regions. The WITH clause defines two auxiliary statements named regional\_sales and top\_regions, where the output of regional\_sales is used in top\_regions and the output of top\_regions is used in the primary SELECT query. This example could have been written without WITH, but we'd have needed two levels of nested sub-SELECTs. It's a bit easier to follow this way.

## Recursion (advanced)

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output. A very simple example is this query to sum the integers from 1 through 100:

WITH RECURSIVE t(n) AS (

VALUES (1)

UNION ALL

SELECT n+1 FROM t WHERE n < 100

)

SELECT sum(n) FROM t;

# 4.1. Lexical Structure

SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (“;”). The end of the input stream also terminates a command.

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.

Example of 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).

## 4.1.1. Identifiers and Key Words

Tokens such as SELECT, UPDATE, or VALUES in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens MY\_TABLE and A are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in.

SQL identifiers and key words must begin with a letter (a-z, but also letters with diacritical marks and non-Latin letters) or an underscore (\_). Subsequent characters in an identifier or key word can be letters, underscores, digits (0-9), or dollar signs ($). Note that dollar signs are not allowed in identifiers according to the letter of the SQL standard.

Key words and identifiers are case insensitive. By default they are maximum of 63 bytes long.

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;

Delimited identifiers using double quotes: "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.

Example of quoting:

UPDATE "my\_table" SET "a" = 5;

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

[A variant of quoted identifiers allows including escaped Unicode characters identified by their code points.

Russian word “slon” (elephant) in Cyrillic letters:

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

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by PostgreSQL, but "Foo" and "FOO" are different from these three and each other.

## 4.1.2. Constants

There are three kinds of *implicitly-typed constants* in PostgreSQL: strings, bit strings, and numbers,

## 4.1.2.1. String Constants

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

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

SELECT 'foo'

'bar';

is equivalent to:

SELECT 'foobar';

but:

SELECT 'foo' 'bar';

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

## 4.1.2.2. String Constants With C-Style Escapes

PostgreSQL also accepts “escape” string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter E (upper or lower case) just before the opening single quote, e.g., E'foo'. (When continuing an escape string constant across lines, write E only before the first opening quote.) Within an escape string, a backslash character (\) begins a C-like *backslash escape* sequence, in which the combination of backslash and following character(s) represent a special byte value, as shown in [**Table 4.1**](https://www.postgresql.org/docs/12/sql-syntax-lexical.html#SQL-BACKSLASH-TABLE).

### Table 4.1. Backslash Escape Sequences

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

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

## 4.1.2.4. Dollar-Quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes or backslashes, since each of those must be doubled. To allow more readable queries in such situations, PostgreSQL provides another way, called “dollar quoting”, to write string constants. Example:

$$Dianne's horse$$

$SomeTag$Dianne's horse$SomeTag$

Dollar quoting is not part of the SQL standard.

## 4.1.2.5. Bit-String Constants

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

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

Both forms of bit-string constant can be continued across lines in the same way as regular string constants.

## 4.1.2.6. Numeric Constants

(‘Numeric’ here is in terms of general numbers. This could be confused with the distinct, SQL stn *numeric* type, aka *decimal*, which is a non floating-point, exact decimal precision number type, useful to monetary calculations. See 8.1 in the official docs for details on different number types.)

Numeric constants are accepted in these general forms:

***digits***

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

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

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

Representing decimal places and exponents, where ***digits*** is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used.

These are some examples of valid numeric constants:

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

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type integer if its value fits in type integer (32 bits); otherwise it is presumed to be type bigint if its value fits in type bigint (64 bits); otherwise it is taken to be the floating-point type ‘numeric’.

Constants that contain decimal points and/or exponents are always initially presumed to be type numeric.

## 4.1.2.7. Constants Of Other Types

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

***type*** '***string***'

'***string***'::***type***

CAST ( '***string***' AS ***type*** )

Explicit typing, as above, can be omitted if there is no ambiguity.

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

***typename*** ( '***string***' )

but not all type names can be used in this way; see [**Section 4.2.9**](https://www.postgresql.org/docs/12/sql-expressions.html#SQL-SYNTAX-TYPE-CASTS) for details.

To avoid ambiguity, the ***type*** '***string***' syntax can only be used to specify the type of a literal constant.

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

## 4.1.3. Operators

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

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

## 4.1.5. Comments

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

-- This is a standard SQL comment

Alternatively, C-style block comments can be used:

/\* multiline comment

\* with nesting: /\* nested block comment \*/

\*/

## 4.1.6. Operator Precedence

### Table 4.2. Operator Precedence (highest to lowest)

|  |  |  |
| --- | --- | --- |
| **Operator / Element** | **Associativity** | **Description** |
| . | left | table/column name separator |
| :: | left | PostgreSQL-style typecast |
| [ ] | left | array element selection |
| + - | right | unary plus, unary minus |
| ^ | left | exponentiation |
| \* / % | left | multiplication, division, modulo |
| + - | left | addition, subtraction |
| (any other operator) | left | all other native and user-defined operators |
| BETWEEN IN LIKE ILIKE SIMILAR |  | range containment, set membership, string matching |
| < > = <= >= <> |  | comparison operators |
| IS ISNULL NOTNULL |  | IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc |
| NOT | right | logical negation |
| AND | left | logical conjunction |
| OR | left | logical disjunction |

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

# 4.2. Value Expressions

Value expressions are used 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.

## 4.2.1. Column References

A column can be referenced in the form:

***correlation***.***columnname***

***correlation*** is the name of a table (possibly qualified with a schema name), or an alias for a table. The correlation name and separating dot can be omitted if the column name is unique across all the tables being used in the current query.

## 4.2.2. Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. The form of a parameter reference is:

$***number***

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

CREATE FUNCTION dept(text) RETURNS dept

AS $$ SELECT \* FROM dept WHERE name = $1 $$

LANGUAGE SQL;

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

## 4.2.3. Subscripts

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

***expression***[***subscript***]

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

***expression***[***lower\_subscript***:***upper\_subscript***]

(Here, the brackets [ ] are meant to appear literally.) Each ***subscript*** is itself an expression, which must yield an integer value.

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

mytable.arraycolumn[4]

mytable.two\_d\_column[17][34]

$1[10:42]

(arrayfunction(a,b))[42]

The parentheses in the last example are required. See [**Section 8.15**](https://www.postgresql.org/docs/12/arrays.html) for more about arrays.

## 4.2.4. Field Selection

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

***expression***.***fieldname***

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

mytable.mycolumn

$1.somecolumn

(rowfunction(a,b)).col3

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

(compositecol).somefield

(mytable.compositecol).somefield

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

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

(compositecol).\*

## 4.2.5. Operator Invocations

There are three possible syntaxes for an operator invocation:

|  |
| --- |
| ***expression*** ***operator*** ***expression*** (binary infix operator) |
| ***operator*** ***expression*** (unary prefix operator) |
| ***expression*** ***operator*** (unary postfix operator) |

where the ***operator*** token follows the syntax rules of [**Section 4.1.3**](https://www.postgresql.org/docs/12/sql-syntax-lexical.html#SQL-SYNTAX-OPERATORS), or is one of the key words AND, OR, and NOT, or is a qualified operator name in the form:

OPERATOR(***schema***.***operatorname***)

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. [**Chapter 9**](https://www.postgresql.org/docs/12/functions.html) describes the built-in operators.

## 4.2.6. Function Calls

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

***function\_name*** ([***expression*** [, ***expression*** ... ]] )

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

sqrt(2)

The list of built-in functions is in [**Chapter 9**](https://www.postgresql.org/docs/12/functions.html). Other functions can be added by the user.

## 4.2.7. Aggregate Expressions

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

***aggregate\_name*** (***expression*** [ , ... ] [ ***order\_by\_clause*** ] ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** (ALL ***expression*** [ , ... ] [ ***order\_by\_clause*** ] ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** (DISTINCT ***expression*** [ , ... ] [ ***order\_by\_clause*** ] ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** ( \* ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** ( [ ***expression*** [ , ... ] ] ) WITHIN GROUP ( ***order\_by\_clause*** ) [ FILTER ( WHERE ***filter\_clause*** ) ]

Example:

Select max(age) from Employees

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

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

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

Ordinarily, the input rows are fed to the aggregate function in an unspecified order. In many cases this does not matter; for example, min produces the same result no matter what order it receives the inputs in. However, some aggregate functions (such as array\_agg and string\_agg) produce results that depend on the ordering of the input rows. When using such an aggregate, the optional ***order\_by\_clause*** can be used to specify the desired ordering. The ***order\_by\_clause*** has the same syntax as for a query-level ORDER BY clause, as described in [**Section 7.5**](https://www.postgresql.org/docs/12/queries-order.html), except that its expressions are always just expressions and cannot be output-column names or numbers. For example:

SELECT array\_agg(a ORDER BY b DESC) FROM table;

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

SELECT string\_agg(a, ',' ORDER BY a) FROM table;

not this:

SELECT string\_agg(a ORDER BY a, ',') FROM table; -- incorrect

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

If DISTINCT is specified in addition to an ***order\_by\_clause***, then all the ORDER BY expressions must match regular arguments of the aggregate; that is, you cannot sort on an expression that is not included in the DISTINCT list.

**Note**

The ability to specify both DISTINCT and ORDER BY in an aggregate function is a PostgreSQL extension.

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

When an aggregate expression appears in a subquery (see [**Section 4.2.11**](https://www.postgresql.org/docs/12/sql-expressions.html#SQL-SYNTAX-SCALAR-SUBQUERIES) and [**Section 9.22**](https://www.postgresql.org/docs/12/functions-subquery.html)), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments (and ***filter\_clause*** if any) contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or HAVING clause applies with respect to the query level that the aggregate belongs to.

### Ordered Set Aggregates

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

An example of an ordered-set aggregate call is:

SELECT percentile\_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;

percentile\_cont

-----------------

50489

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

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

SELECT

count(\*) AS unfiltered,

count(\*) FILTER (WHERE i < 5) AS filtered

FROM generate\_series(1,10) AS s(i);

unfiltered | filtered

------------+----------

10 | 4

(1 row)

## 4.2.8. Window Function Calls (advanced SQL)

See <https://en.wikipedia.org/wiki/SQL_window_function> for an introduction.

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

***function\_name*** ([***expression*** [, ***expression*** ... ]]) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ***window\_name***

***function\_name*** ([***expression*** [, ***expression*** ... ]]) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ( ***window\_definition*** )

***function\_name*** ( \* ) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ***window\_name***

***function\_name*** ( \* ) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ( ***window\_definition*** )

where ***window\_definition*** has the syntax

[ ***existing\_window\_name*** ]

[ PARTITION BY ***expression*** [, ...] ]

[ ORDER BY ***expression*** [ ASC | DESC | USING ***operator*** ] [ NULLS { FIRST | LAST } ] [, ...] ]

[ ***frame\_clause*** ]

The optional ***frame\_clause*** can be one of

{ RANGE | ROWS | GROUPS } ***frame\_start*** [ ***frame\_exclusion*** ]

{ RANGE | ROWS | GROUPS } BETWEEN ***frame\_start*** AND ***frame\_end*** [ ***frame\_exclusion*** ]

where ***frame\_start*** and ***frame\_end*** can be one of

UNBOUNDED PRECEDING

***offset*** PRECEDING

CURRENT ROW

***offset*** FOLLOWING

UNBOUNDED FOLLOWING

and ***frame\_exclusion*** can be one of

EXCLUDE CURRENT ROW

EXCLUDE GROUP

EXCLUDE TIES

EXCLUDE NO OTHERS

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

***window\_name*** is a reference to a named window specification defined in the query's WINDOW clause. Alternatively, a full ***window\_definition*** can be given within parentheses, using the same syntax as for defining a named window in the WINDOW clause; see the [**SELECT**](https://www.postgresql.org/docs/12/sql-select.html) reference page for details. It's worth pointing out that OVER wname is not exactly equivalent to OVER (wname ...); the latter implies copying and modifying the window definition, and will be rejected if the referenced window specification includes a frame clause.

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

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

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

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

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

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

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

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

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

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

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

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

The built-in window functions are described in [**Table 9.60**](https://www.postgresql.org/docs/12/functions-window.html#FUNCTIONS-WINDOW-TABLE). Other window functions can be added by the user. Also, any built-in or user-defined general-purpose or statistical aggregate can be used as a window function. (Ordered-set and hypothetical-set aggregates cannot presently be used as window functions.)

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

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

More information about window functions can be found in [**Section 3.5**](https://www.postgresql.org/docs/12/tutorial-window.html), [**Section 9.21**](https://www.postgresql.org/docs/12/functions-window.html), and [**Section 7.2.5**](https://www.postgresql.org/docs/12/queries-table-expressions.html#QUERIES-WINDOW).

## 4.2.9. Type Casts (Advanced SQL)

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

CAST ( ***expression*** AS ***type*** )

***expression***::***type***

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

When a cast is applied to a value expression of a known type, it represents a run-time type conversion.

Automatic casting is only done for casts that are marked “OK to apply implicitly” in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

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

***typename*** ( ***expression*** )

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

## 4.2.10. Collation Expressions (advanced SQL, usually for administrators)

Collation concerns ordering/sorting/comparing of strings for differing character sets, Latin1, etc.

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

***expr*** COLLATE ***collation***

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

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

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

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

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

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

Note that in the latter case the COLLATE clause is attached to an input argument of the operator we wish to affect. It doesn't matter which argument of the operator or function call the COLLATE clause is attached to, because the collation that is applied by the operator or function is derived by considering all arguments, and an explicit COLLATE clause will override the collations of all other arguments. (Attaching non-matching COLLATE clauses to more than one argument, however, is an error. For more details see [**Section 23.2**](https://www.postgresql.org/docs/12/collation.html).) Thus, this gives the same result as the previous example:

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

But this is an error:

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

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

**4.2.11. Scalar Subqueries**

A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column. (See [**Chapter 7**](https://www.postgresql.org/docs/12/queries.html) for information about writing queries.) The SELECT query is executed and the single returned value is used in the surrounding value expression.

For example, the following gives a table of the largest city populations in each state:

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

FROM states;

## 4.2.12. Array Constructors (Advanced SQL)

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

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

Result:

array

---------

{1,2,7}

(1 row)

By default, the array element type is the common type of the member expressions, determined using the same rules as for UNION or CASE constructs (see [**Section 10.5**](https://www.postgresql.org/docs/12/typeconv-union-case.html)). You can override this by explicitly casting the array constructor to the desired type, for example:

SELECT ARRAY[1,2,22.7]::integer[];

array

----------

{1,2,23}

(1 row)

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

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

array

---------------

{{1,2},{3,4}}

(1 row)

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

array

---------------

{{1,2},{3,4}}

(1 row)

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

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

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

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

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;

array

------------------------------------------------

{{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}

(1 row)

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

SELECT ARRAY[]::integer[];

array

-------

{}

(1 row)

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

SELECT ARRAY(SELECT oid FROM pg\_proc WHERE proname LIKE 'bytea%');

array

-----------------------------------------------------------------------

{2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}

(1 row)

SELECT ARRAY(SELECT ARRAY[i, i\*2] FROM generate\_series(1,5) AS a(i));

array

----------------------------------

{{1,2},{2,4},{3,6},{4,8},{5,10}}

(1 row)

The subquery must return a single column

The subscripts (value to reference an element) of an array value built with ARRAY always begin with one. For more information about arrays, see [**Section 8.15**](https://www.postgresql.org/docs/12/arrays.html).

## 4.2.13. Row Constructors (Advanced SQL)

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

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

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

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter.

For more detail see [**Section 9.23**](https://www.postgresql.org/docs/12/functions-comparisons.html). Row constructors can also be used in connection with subqueries, as discussed in [**Section 9.22**](https://www.postgresql.org/docs/12/functions-subquery.html).

## 4.2.14. Expression Evaluation Rules

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

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

SELECT true OR somefunc();

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

SELECT somefunc() OR true;

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

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

When it is essential to force evaluation order, a CASE construct (see [**Section 9.17**](https://www.postgresql.org/docs/12/functions-conditional.html)) can be used. For example, this is an untrustworthy way of trying to avoid division by zero in a WHERE clause:

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

But this is safe:

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

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

CASE is not a cure-all for such issues, however. One limitation of the technique illustrated above is that it does not prevent early evaluation of constant subexpressions. As described in [**Section 37.7**](https://www.postgresql.org/docs/12/xfunc-volatility.html), functions and operators marked IMMUTABLE can be evaluated when the query is planned rather than when it is executed. Thus for example

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

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

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

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

SELECT CASE WHEN min(employees) > 0

THEN avg(expenses / employees)

END

FROM departments;

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

# Selected Types

As with the rest of this learner, types covered here have very advanced details removed or briefly summarised.

Additionally, the following advanced types, infrequently used by normal business/analytic queries, are given only a short synopsis with examples to aid recognition:

[8.4. Binary Data Types](https://www.postgresql.org/docs/12/datatype-binary.html)

[8.7. Enumerated Types](https://www.postgresql.org/docs/12/datatype-enum.html)

[8.8. Geometric Types](https://www.postgresql.org/docs/12/datatype-geometric.html)

[8.9. Network Address Types](https://www.postgresql.org/docs/12/datatype-net-types.html)

[8.10. Bit String Types](https://www.postgresql.org/docs/12/datatype-bit.html)

[8.12. UUID Type](https://www.postgresql.org/docs/12/datatype-uuid.html)

[8.13. XML Type](https://www.postgresql.org/docs/12/datatype-xml.html)

[8.14. JSON Types](https://www.postgresql.org/docs/12/datatype-json.html)

[8.15. Arrays](https://www.postgresql.org/docs/12/arrays.html)

[8.16. Composite Types](https://www.postgresql.org/docs/12/rowtypes.html)

[8.17. Range Types](https://www.postgresql.org/docs/12/rangetypes.html)

[8.18. Domain Types](https://www.postgresql.org/docs/12/domains.html)

[8.19. Object Identifier Types](https://www.postgresql.org/docs/12/datatype-oid.html)

[8.20. pg\_lsn Type](https://www.postgresql.org/docs/12/datatype-pg-lsn.html)

[8.21. Pseudo-Types](https://www.postgresql.org/docs/12/datatype-pseudo.html)

## Table of numeric types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals. [**Table 8.2**](https://www.postgresql.org/docs/12/datatype-numeric.html#DATATYPE-NUMERIC-TABLE) lists the available types.

### Table 8.2. Numeric Types

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **bytes** | **Description** | **Range** |
| smallint | 2 | Small range integer | -32768 to +32767 |
| integer | 4 | typical choice for integer | -2147483648 to +2147483647 |
| bigint | 8 | large-range integer | -9223372036854775808 to +9223372036854775807 |
| decimal | varies | User specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| numeric | varies | User specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| real | 4 | Variable precision, inexact | 6 decimal digits precision |
| double precision | 8 | Variable precision, inexact | 15 decimal digits precision |
| smallserial | 2 | small autoincrementing integer | 1 to 32767 |
| serial | 4 | Auto incrementing integer | 1 to 2147483647 |
| bigserial | 8 | large auto incrementing integer | 1 to 9223372036854775807 |

The syntax of constants for the numeric types is described in [**Section 4.1.2**](https://www.postgresql.org/docs/12/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS). The numeric types have a full set of corresponding arithmetic operators and functions. Refer to [**Chapter 9**](https://www.postgresql.org/docs/12/functions.html) for more information. The following sections describe the types in detail.

## 8.1.1. Integer Types

The types smallint, integer, and bigint store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type *integer* is the common choice, as it offers the best balance between range, storage size, and performance. The *smallint* type is generally only used if disk space is at a premium. The *bigint* type is designed to be used when the range of the integer type is insufficient.

SQL only specifies the integer types integer (or int), smallint, and bigint. The type names int2, int4, and int8 are extensions, which are also used by some other SQL database systems.

## 8.1.2. Arbitrary Precision Numbers (*numeric* type)

The type *numeric* has a fractional part (as with a floating-point type) and can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required since floating point types do not have exact results. Calculations with numeric values yield exact results where possible, e.g. addition, subtraction, multiplication. However, calculations on numeric values are very slow compared to the integer types, or to the floating-point types described in the next section.

We use the following terms below: The *precision* of a numeric is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The *scale* of a numeric is the count of decimal digits in the fractional part, to the right of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the maximum precision and the maximum scale of a numeric column can be configured. To declare a column (eg, when creating a table) of type numeric use the syntax:

NUMERIC(***precision***, ***scale***)

The precision must be positive, the scale zero or positive. Alternatively:

NUMERIC(***precision***)

selects a scale of 0. Specifying:

NUMERIC

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas numeric columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

**Note**

The maximum allowed precision when explicitly specified in the type declaration is 1000; NUMERIC without a specified precision is subject to the limits described in [**Table 8.2**](https://www.postgresql.org/docs/12/datatype-numeric.html#DATATYPE-NUMERIC-TABLE).

If the scale of a value to be stored is greater than the declared scale of the column, the system will round (as below).

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the numeric type is more akin to varchar(***n***) than to char(***n***).) The actual storage requirement is two bytes for each group of four decimal digits, plus three to eight bytes overhead.

In addition to ordinary numeric values, the numeric type allows the special value NaN, meaning “not-a-number”. Any operation on NaN yields another NaN. When writing this value as a constant in an SQL command, you must put quotes around it, for example UPDATE table SET x = 'NaN'. On input, the string NaN is recognized in a case-insensitive manner.

**Note**

In most implementations of the “not-a-number” concept, NaN is not considered equal to any other numeric value (including NaN). In order to allow numeric values to be sorted and used in tree-based indexes, PostgreSQL treats NaN values as equal, and greater than all non-NaN values.

The types decimal and numeric are equivalent. Both types are part of the SQL standard.

When rounding values, the numeric type rounds ties away from zero, while (on most machines) the real and double precision types round ties to the nearest even number. For example:

SELECT x,

round(x::numeric) AS num\_round,

round(x::double precision) AS dbl\_round

FROM generate\_series(-3.5, 3.5, 1) as x;

x | num\_round | dbl\_round

------+-----------+-----------

-3.5 | -4 | -4

-2.5 | -3 | -2

-1.5 | -2 | -2

-0.5 | -1 | -0

0.5 | 1 | 0

1.5 | 2 | 2

2.5 | 3 | 2

3.5 | 4 | 4

(8 rows)

## 8.1.3. Floating-Point Types

The data types real and double precision are inexact, variable-precision numeric types. On all currently supported platforms, these types are implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

* If you require exact storage and calculations (such as for monetary amounts), use the numeric type instead.
* If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
* Comparing two floating-point values for equality might not always work as expected.

On all currently supported platforms, the real type has a range of around 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The double precision type has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow err**Note**

In addition to ordinary numeric values, the floating-point types have several special values:

Infinity  
-Infinity  
NaN

These represent the IEEE 754 special values “infinity”, “negative infinity”, and “not-a-number”, respectively. When writing these values as constants in an SQL command, you must put quotes around them, for example UPDATE table SET x = '-Infinity'. On input, these strings are recognized in a case-insensitive manner.

**Note**

IEEE754 specifies that NaN should not compare equal to any other floating-point value (including NaN). In order to allow floating-point values to be sorted and used in tree-based indexes, PostgreSQL treats NaN values as equal, and greater than all non-NaN values.

## 8.1.4. Serial Types

A PostgreSQL-specific way to create an autoincrementing column. Another way is to use the SQL-standard identity column feature, described at [**CREATE TABLE**](https://www.postgresql.org/docs/12/sql-createtable.html).

The data types smallserial, serial and bigserial are not true types, but merely a notational convenience for creating unique identifier columns (similar to the AUTO\_INCREMENT property supported by some other databases). In the current implementation, specifying:

**Note**

Because smallserial, serial and bigserial are implemented using sequences, there may be "holes" or gaps in the sequence of values which appears in the column, even if no rows are ever deleted. A value allocated from the sequence is still "used up" even if a row containing that value is never successfully inserted into the table column. This may happen, for example, if the inserting transaction rolls back. See nextval() in [**Section 9.16**](https://www.postgresql.org/docs/12/functions-sequence.html) for details.

## 8.2. Monetary Types

The money type stores a currency amount with a fixed fractional precision; see [**Table 8.3**](https://www.postgresql.org/docs/12/datatype-money.html#DATATYPE-MONEY-TABLE). The fractional precision is determined by the database's [**lc\_monetary**](https://www.postgresql.org/docs/12/runtime-config-client.html#GUC-LC-MONETARY) setting. The range shown in the table assumes there are two fractional digits. Input is accepted in a variety of formats, including integer and floating-point literals, as well as typical currency formatting, such as '$1,000.00'. Output is generally in the latter form but depends on the locale.

### Table 8.3. Monetary Types

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **bytes** | **Description** | **Range** |
| money | 8 bytes | currency amount | -92233720368547758.08 to +92233720368547758.07 |

Since the output of this data type is locale-sensitive, it might not work to load money data into a database that has a different setting of lc\_monetary. To avoid problems, before restoring a dump into a new database make sure lc\_monetary has the same or equivalent value as in the database that was dumped.

Values of the numeric, int, and bigint data types can be cast to money. Conversion from the real and double precision data types can be done by casting to numeric first, for example:

SELECT '12.34'::float8::numeric::money;

However, this is not recommended. Floating point numbers should not be used to handle money due to the potential for rounding errors.

A money value can be cast to numeric without loss of precision. Conversion to other types could potentially lose precision, and must also be done in two stages:

SELECT '52093.89'::money::numeric::float8;

Division of a money value by an integer value is performed with truncation of the fractional part towards zero. To get a rounded result, divide by a floating-point value, or cast the money value to numeric before dividing and back to money afterwards. (The latter is preferable to avoid risking precision loss.) When a money value is divided by another money value, the result is double precision (i.e., a pure number, not money); the currency units cancel each other out in the division.

## 8.3. Character Types

### Table 8.4. Character Types

|  |  |
| --- | --- |
| **Name** | **Description** |
| character varying(***n***), varchar(***n***) | variable-length with limit |
| character(***n***), char(***n***) | fixed-length, blank padded |
| text | variable unlimited length |

[**Table 8.4**](https://www.postgresql.org/docs/12/datatype-character.html#DATATYPE-CHARACTER-TABLE) shows the general-purpose character types available in PostgreSQL.

SQL defines two primary character types: character varying(***n***) and character(***n***), where ***n*** is a positive integer. Both of these types can store strings up to ***n*** characters (not bytes) in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type character will be space-padded; values of type character varying will simply store the shorter string.

If one explicitly casts a value to character varying(***n***) or character(***n***), then an over-length value will be truncated to ***n*** characters without raising an error. (This too is required by the SQL standard.)

The notations varchar(***n***) and char(***n***) are aliases for character varying(***n***) and character(***n***), respectively. character without length specifier is equivalent to character(1). If character varying is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL provides the text type, which stores strings of any length. Although the type text is not in the SQL standard, several other SQL database management systems have it as well.

Values of type character are physically padded with spaces to the specified width ***n***, and are stored and displayed that way. However, trailing spaces are treated as semantically insignificant and disregarded when comparing two values of type character. In collations where whitespace is significant, this behavior can produce unexpected results; for example SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2) returns true, even though C locale would consider a space to be greater than a newline. Trailing spaces are removed when converting a character value to one of the other string types.

Note that trailing spaces *are* semantically significant in character varying and text values, and when using pattern matching, that is LIKE and regular expressions.

**Tip**

There is no performance difference among these three types, apart from increased storage space when using the blank-padded type, and a few extra CPU cycles to check the length when storing into a length-constrained column. While character(***n***) has performance advantages in some other database systems, there is no such advantage in PostgreSQL; in fact character(***n***) is usually the slowest of the three because of its additional storage costs. In most situations text or character varying should be used instead.

**Example 8.1. Using the Character Types**

CREATE TABLE test1 (a character(4));

INSERT INTO test1 VALUES ('ok');

SELECT a, char\_length(a) FROM test1; -- (1)

a | char\_length

------+-------------

ok | 2

CREATE TABLE test2 (b varchar(5));

INSERT INTO test2 VALUES ('ok');

INSERT INTO test2 VALUES ('good ');

INSERT INTO test2 VALUES ('too long');

ERROR: value too long for type character varying(5)

INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation

SELECT b, char\_length(b) FROM test2;

b | char\_length

-------+-------------

ok | 2

good | 5

too l | 5

|  |  |
| --- | --- |
| [**(1)**](https://www.postgresql.org/docs/12/datatype-character.html#co.datatype-char) | The char\_length function is discussed in [**Section 9.4**](https://www.postgresql.org/docs/12/functions-string.html). |

There are two other fixed-length character types in PostgreSQL, shown in [**Table 8.5**](https://www.postgresql.org/docs/12/datatype-character.html#DATATYPE-CHARACTER-SPECIAL-TABLE). The *name* type exists *only* for the storage of identifiers in the internal system catalogs and is not intended for use by the general user. 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.

### Table 8.5. Special Character Types

|  |  |  |
| --- | --- | --- |
| **Name** | **Storage Size** | **Description** |
| "char" | 1 byte | single-byte internal type |
| name | 64 bytes | internal type for object names |

## 8.4. Binary Data Types (advanced SQL)

The bytea data type allows storage of binary strings; see [**Table 8-6**](https://www.postgresql.org/docs/9.3/datatype-binary.html#DATATYPE-BINARY-TABLE). String representations of binary (raw) data, perhaps jpegs, doc files etc.

### Table 8-6. Binary Data Types

|  |  |  |
| --- | --- | --- |
| **Name** | **Storage Size** | **Description** |
| bytea | 1 or 4 bytes plus the actual binary string | variable-length binary string |

A binary string is a sequence of octets (aka bytes). Binary strings are distinguished from character strings in two ways. First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (usually, octets outside the decimal range 32 to 126). Character strings disallow zero octets, and also disallow any other octet values and sequences of octet values that are invalid according to the database's selected character set encoding. Second, operations on binary strings process the actual bytes, whereas the processing of character strings depends on locale settings. In short, binary strings are appropriate for storing data that the programmer thinks of as "raw bytes", whereas character strings are appropriate for storing text.

The SQL standard defines a different binary string type, called BLOB or BINARY LARGE OBJECT. The input format is different from bytea, but the provided functions and operators are mostly the same.

### Table 8-7. bytea Literal Escaped Octets (to aid recognition in 3rd party query SQL)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Decimal Octet Value** | **Description** | **Escaped Input** | **Example** | **Output (Hex)** |
| 0 | zero octet | '\000' | SELECT '\000'::bytea; | \x00 |
| 39 | single quote | '''' or '\047' | SELECT ''''::bytea; | \x27 |
| 92 | backslash | '\' or '\\134' | SELECT '\\'::bytea; | \x5c |
| 0 to 31 and 127 to 255 | "non-printable" octets | '\***xxx'*** (octal value) | SELECT '\001'::bytea; | \x01 |

Bytea octets are output in hex format by default.

## 8.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in [**Table 8-9**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-TABLE). The operations available on these data types are described in [**Section 9.9**](https://www.postgresql.org/docs/9.3/functions-datetime.html). Dates are counted according to the Gregorian calendar, even in years before that calendar was introduced (see [**Section B.4**](https://www.postgresql.org/docs/9.3/datetime-units-history.html) for more information).

### Table 8-9. Date/Time Types

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **bytes** | **Description** | **Lowest** | **Highest** | **Resolution** |
| timestamp [ (***p***) ] [without time zone] | 8 | both date and time (no time zone) | 4713 BC | 294276 AD | 1 μs / 14 digits |
| timestamp [ (***p***) ] with time zone | 8 | both date and time, with time zone | 4713 BC | 294276 AD | 1 μs / 14 digits |
| date | 4 | date (no time of day) | 4713 BC | 5874897 AD | 1 day |
| time [ (***p***) ] [without time zone] | 8 | time of day (no date) | 00:00:00 | 24:00:00 | 1 μs / 14 digits |
| time [ (***p***) ] with time zone | 12 | times of day only, with time zone | 00:00:00 + 1459 | 24:00:00 -1459 | 1 μs / 14 digits |
| interval [ ***fields*** ] [(***p***)] | 16 | time interval | negative 178000000 years | 178000000 years | 1 μs / 14 digits |

**Note:** The SQL standard requires that writing just timestamp be equivalent to timestamp without time zone, and PostgreSQL honors that behavior. (Releases prior to 7.3 treated it as timestamp with time zone.) timestamptz is accepted as an abbreviation for timestamp with time zone; this is a PostgreSQL extension.

time, timestamp, and interval accept an optional precision value ***p*** which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of ***p*** is from 0 to 6 for the timestamp and interval types.

**Note:** When timestamp values are stored as eight-byte integers (currently the default), microsecond precision is available over the full range of values. When timestamp values are stored as double precision floating-point numbers instead (a deprecated compile-time option), the effective limit of precision might be less than 6. timestamp values are stored as seconds before or after midnight 2000-01-01. When timestamp values are implemented using floating-point numbers, microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. Note that using floating-point datetimes allows a larger range of timestamp values to be represented than shown above: from 4713 BC up to 5874897 AD.

The same compile-time option also determines whether time and interval values are stored as floating-point numbers or eight-byte integers. In the floating-point case, large interval values degrade in precision as the size of the interval increases.

For the time types, the allowed range of ***p*** is from 0 to 6 when eight-byte integer storage is used, or from 0 to 10 when floating-point storage is used.

The interval type has an additional option, which is to restrict the set of stored fields by writing one of these phrases:

YEAR

MONTH

DAY

HOUR

MINUTE

SECOND

YEAR TO MONTH

DAY TO HOUR

DAY TO MINUTE

DAY TO SECOND

HOUR TO MINUTE

HOUR TO SECOND

MINUTE TO SECOND

Note that if both ***fields*** and ***p*** are specified, the ***fields*** must include SECOND, since the precision applies only to the seconds.

The type time with time zone is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of date, time, timestamp without time zone, and timestamp with time zone should provide a complete range of date/time functionality required by any application.

The types abstime and reltime are lower precision types which are used internally. You are discouraged from using these types in applications; these internal types might disappear in a future release.

**8.5.1. Date/Time Input**

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of day, month, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the [**DateStyle**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-DATESTYLE) parameter to MDY to select month-day-year interpretation, DMY to select day-month-year interpretation, or YMD to select year-month-day interpretation.

PostgreSQL is more flexible in handling date/time input than the SQL standard requires. See [**Appendix B**](https://www.postgresql.org/docs/9.3/datetime-appendix.html) for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to [**Section 4.1.2.7**](https://www.postgresql.org/docs/9.3/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC) for more information. SQL requires the following syntax

***type*** [ (***p***) ] '***value***'

where ***p*** is an optional precision specification giving the number of fractional digits in the seconds field. Precision can be specified for time, timestamp, and interval types. The allowed values are mentioned above. If no precision is specified in a constant specification, it defaults to the precision of the literal value.

**8.5.1.1. Dates**

[**Table 8-10**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-DATE-TABLE) shows some possible inputs for the date type.

### Table 8-10. Date Input

|  |  |
| --- | --- |
| **Example** | **Description** |
| 1999-01-08 | ISO 8601; January 8 in any mode (recommended format) |
| January 8, 1999 | unambiguous in any datestyle input mode |
| 1/8/1999 | January 8 in MDY mode; August 1 in DMY mode |
| 1/18/1999 | January 18 in MDY mode; rejected in other modes |
| 01/02/03 | January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode |
| 1999-Jan-08 | January 8 in any mode |
| Jan-08-1999 | January 8 in any mode |
| 08-Jan-1999 | January 8 in any mode |
| 99-Jan-08 | January 8 in YMD mode, else error |
| 08-Jan-99 | January 8, except error in YMD mode |
| Jan-08-99 | January 8, except error in YMD mode |
| 19990108 | ISO 8601; January 8, 1999 in any mode |
| 990108 | ISO 8601; January 8, 1999 in any mode |
| 1999.008 | year and day of year |
| J2451187 | Julian date |
| January 8, 99 BC | year 99 BC |

**8.5.1.2. Times**

The time-of-day types are time [ (***p***) ] without time zone and time [ (***p***) ] with time zone. time alone is equivalent to time without time zone.

Valid input for these types consists of a time of day followed by an optional time zone. (See [**Table 8-11**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-TIME-TABLE) and [**Table 8-12**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-TIMEZONE-TABLE).) If a time zone is specified in the input for time without time zone, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as America/New\_York. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the time with time zone value.

### Table 8-11. Time Input

|  |  |
| --- | --- |
| **Example** | **Description** |
| 04:05:06.789 | ISO 8601 |
| 04:05:06 | ISO 8601 |
| 04:05 | ISO 8601 |
| 040506 | ISO 8601 |
| 04:05 AM | same as 04:05; AM does not affect value |
| 04:05 PM | same as 16:05; input hour must be <= 12 |
| 04:05:06.789-8 | ISO 8601 |
| 04:05:06-08:00 | ISO 8601 |
| 04:05-08:00 | ISO 8601 |
| 040506-08 | ISO 8601 |
| 04:05:06 PST | time zone specified by abbreviation |
| 2003-04-12 04:05:06 America/New\_York | time zone specified by full name |

### Table 8-12. Time Zone Input

|  |  |
| --- | --- |
| **Example** | **Description** |
| PST | Abbreviation (for Pacific Standard Time) |
| America/New\_York | Full time zone name |
| PST8PDT | POSIX-style time zone specification |
| -8:00 | ISO-8601 offset for PST |
| -800 | ISO-8601 offset for PST |
| -8 | ISO-8601 offset for PST |
| zulu | Military abbreviation for UTC |
| z | Short form of zulu |

Refer to [**Section 8.5.3**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-TIMEZONES) for more information on how to specify time zones.

**8.5.1.3. Time Stamps**

Valid input for the time stamp types consists of the concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC. (Alternatively, AD/BC can appear before the time zone, but this is not the preferred ordering.) Thus:

1999-01-08 04:05:06

and:

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

are valid values, which follow the ISO 8601 standard. In addition, the common format:

January 8 04:05:06 1999 PST

is supported.

The SQL standard differentiates timestamp without time zone and timestamp with time zone literals by the presence of a "+" or "-" symbol and time zone offset after the time. Hence, according to the standard,

TIMESTAMP '2004-10-19 10:23:54'

is a timestamp without time zone, while

TIMESTAMP '2004-10-19 10:23:54+02'

is a timestamp with time zone. PostgreSQL never examines the content of a literal string before determining its type, and therefore will treat both of the above as timestamp without time zone. To ensure that a literal is treated as timestamp with time zone, give it the correct explicit type:

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

In a literal that has been determined to be timestamp without time zone, PostgreSQL will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input value, and is not adjusted for time zone.

For timestamp with time zone, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's [**TimeZone**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-TIMEZONE) parameter, and is converted to UTC using the offset for the timezone zone.

When a timestamp with time zone value is output, it is always converted from UTC to the current timezone zone, and displayed as local time in that zone. To see the time in another time zone, either change timezone or use the AT TIME ZONE construct (see [**Section 9.9.3**](https://www.postgresql.org/docs/9.3/functions-datetime.html#FUNCTIONS-DATETIME-ZONECONVERT)).

Conversions between timestamp without time zone and timestamp with time zone normally assume that the timestamp without time zone value should be taken or given as timezone local time. A different time zone can be specified for the conversion using AT TIME ZONE.

**8.5.1.4. Special Values**

PostgreSQL supports several special date/time input values for convenience, as shown in [**Table 8-13**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-SPECIAL-TABLE). The values infinity and -infinity are specially represented inside the system and will be displayed unchanged; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, now and related strings are converted to a specific time value as soon as they are read.) All of these values need to be enclosed in single quotes when used as constants in SQL commands.

**Table 8-13. Special Date/Time Inputs**

|  |  |  |
| --- | --- | --- |
| **Input String** | **Valid Types** | **Description** |
| epoch | date, timestamp | 1970-01-01 00:00:00+00 (Unix system time zero) |
| infinity | date, timestamp | later than all other time stamps |
| -infinity | date, timestamp | earlier than all other time stamps |
| now | date, time, timestamp | current transaction's start time |
| today | date, timestamp | midnight today |
| tomorrow | date, timestamp | midnight tomorrow |
| yesterday | date, timestamp | midnight yesterday |
| allballs | time | 00:00:00.00 UTC |

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. The latter four accept an optional subsecond precision specification. (See [**Section 9.9.4**](https://www.postgresql.org/docs/9.3/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT).) Note that these are SQL functions and are not recognized in data input strings.

**8.5.2. Date/Time Output**

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES (Unix date format), or German. The default is the ISO format. (The SQL standard requires the use of the ISO 8601 format. The name of the "SQL" output format is a historical accident.) [**Table 8-14**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-OUTPUT-TABLE) shows examples of each output style. The output of the date and time types is generally only the date or time part in accordance with the given examples. However, the POSTGRES style outputs date-only values in ISO format.

**Table 8-14. Date/Time Output Styles**

|  |  |  |
| --- | --- | --- |
| **Style Spec** | **Description** | **Example** |
| ISO | ISO 8601, SQL standard | 1997-12-17 07:37:16-08 |
| SQL | traditional style | 12/17/1997 07:37:16.00 PST |
| Postgres | original style | Wed Dec 17 07:37:16 1997 PST |
| German | regional style | 17.12.1997 07:37:16.00 PST |

**Note:** ISO 8601 specifies the use of uppercase letter T to separate the date and time. PostgreSQL accepts that format on input, but on output it uses a space rather than T, as shown above. This is for readability and for consistency with RFC 3339 as well as some other database systems.

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See [**Section 8.5.1**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-INPUT) for how this setting also affects interpretation of input values.) [**Table 8-15**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-DATETIME-OUTPUT2-TABLE) shows examples.

**Table 8-15. Date Order Conventions**

|  |  |  |
| --- | --- | --- |
| **datestyle Setting** | **Input Ordering** | **Example Output** |
| SQL, DMY | ***day***/***month***/***year*** | 17/12/1997 15:37:16.00 CET |
| SQL, MDY | ***month***/***day***/***year*** | 12/17/1997 07:37:16.00 PST |
| Postgres, DMY | ***day***/***month***/***year*** | Wed 17 Dec 07:37:16 1997 PST |

The date/time style can be selected by the user using the SET datestyle command, the [**DateStyle**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-DATESTYLE) parameter in the postgresql.conf configuration file, or the PGDATESTYLE environment variable on the server or client.

The formatting function to\_char (see [**Section 9.8**](https://www.postgresql.org/docs/9.3/functions-formatting.html)) is also available as a more flexible way to format date/time output.

**8.5.3. Time Zones**

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900s, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. PostgreSQL uses the widely-used IANA (Olson) time zone database for information about historical time zone rules. For times in the future, the assumption is that the latest known rules for a given time zone will continue to be observed indefinitely far into the future.

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

* Although the date type cannot have an associated time zone, the time type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
* The default time zone is specified as a constant numeric offset from UTC. It is therefore impossible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We do not recommend using the type time with time zone (though it is supported by PostgreSQL for legacy applications and for compliance with the SQL standard). PostgreSQL assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the [**TimeZone**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-TIMEZONE) configuration parameter before being displayed to the client.

PostgreSQL allows you to specify time zones in three different forms:

* A full time zone name, for example America/New\_York. The recognized time zone names are listed in the pg\_timezone\_names view (see [**Section 47.71**](https://www.postgresql.org/docs/9.3/view-pg-timezone-names.html)). PostgreSQL uses the widely-used IANA time zone data for this purpose, so the same time zone names are also recognized by much other software.
* A time zone abbreviation, for example PST. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which can imply a set of daylight savings transition-date rules as well. The recognized abbreviations are listed in the pg\_timezone\_abbrevs view (see [**Section 47.70**](https://www.postgresql.org/docs/9.3/view-pg-timezone-abbrevs.html)). You cannot set the configuration parameters [**TimeZone**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-TIMEZONE) or [**log\_timezone**](https://www.postgresql.org/docs/9.3/runtime-config-logging.html#GUC-LOG-TIMEZONE) to a time zone abbreviation, but you can use abbreviations in date/time input values and with the AT TIME ZONE operator.
* In addition to the timezone names and abbreviations, PostgreSQL will accept POSIX-style time zone specifications of the form ***STDoffset*** or ***STDoffsetDST***, where ***STD*** is a zone abbreviation, ***offset*** is a numeric offset in hours west from UTC, and ***DST*** is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if EST5EDT were not already a recognized zone name, it would be accepted and would be functionally equivalent to United States East Coast time. When a daylight-savings zone name is present, it is assumed to be used according to the same daylight-savings transition rules used in the IANA time zone database's posixrules entry. In a standard PostgreSQL installation, posixrules is the same as US/Eastern, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the posixrules file.

In short, this is the difference between abbreviations and full names: abbreviations represent a specific offset from UTC, whereas many of the full names imply a local daylight-savings time rule, and so have two possible UTC offsets. As an example, 2014-06-04 12:00 America/New\_York represents noon local time in New York, which for this particular date was Eastern Daylight Time (UTC-4). So 2014-06-04 12:00 EDT specifies that same time instant. But 2014-06-04 12:00 EST specifies noon Eastern Standard Time (UTC-5), regardless of whether daylight savings was nominally in effect on that date.

To complicate matters, some jurisdictions have used the same timezone abbreviation to mean different UTC offsets at different times; for example, in Moscow MSK has meant UTC+3 in some years and UTC+4 in others. PostgreSQL interprets such abbreviations according to whatever they meant (or had most recently meant) on the specified date; but, as with the EST example above, this is not necessarily the same as local civil time on that date.

One should be wary that the POSIX-style time zone feature can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, SET TIMEZONE TO FOOBAR0 will work, leaving the system effectively using a rather peculiar abbreviation for UTC. Another issue to keep in mind is that in POSIX time zone names, positive offsets are used for locations west of Greenwich. Everywhere else, PostgreSQL follows the ISO-8601 convention that positive timezone offsets are east of Greenwich.

In all cases, timezone names and abbreviations are recognized case-insensitively. (This is a change from PostgreSQL versions prior to 8.2, which were case-sensitive in some contexts but not others.)

Neither timezone names nor abbreviations are hard-wired into the server; they are obtained from configuration files stored under .../share/timezone/ and .../share/timezonesets/ of the installation directory (see [**Section B.3**](https://www.postgresql.org/docs/9.3/datetime-config-files.html)).

The [**TimeZone**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-TIMEZONE) configuration parameter can be set in the file postgresql.conf, or in any of the other standard ways described in [**Chapter 18**](https://www.postgresql.org/docs/9.3/runtime-config.html). There are also some special ways to set it:

* The SQL command SET TIME ZONE sets the time zone for the session. This is an alternative spelling of SET TIMEZONE TO with a more SQL-spec-compatible syntax.
* The PGTZ environment variable is used by libpq clients to send a SET TIME ZONE command to the server upon connection.

**8.5.4. Interval Input**

interval values can be written using the following verbose syntax:

[@] ***quantity*** ***unit*** [***quantity*** ***unit***...] [***direction***]

where ***quantity*** is a number (possibly signed); ***unit*** is microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, or abbreviations or plurals of these units; ***direction*** can be ago or empty. The at sign (@) is optional noise. The amounts of the different units are implicitly added with appropriate sign accounting. ago negates all the fields. This syntax is also used for interval output, if [**IntervalStyle**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-INTERVALSTYLE) is set to postgres\_verbose.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, '1 12:59:10' is read the same as '1 day 12 hours 59 min 10 sec'. Also, a combination of years and months can be specified with a dash; for example '200-10' is read the same as '200 years 10 months'. (These shorter forms are in fact the only ones allowed by the SQL standard, and are used for output when IntervalStyle is set to sql\_standard.)

Interval values can also be written as ISO 8601 time intervals, using either the "format with designators" of the standard's section 4.4.3.2 or the "alternative format" of section 4.4.3.3. The format with designators looks like this:

P ***quantity*** ***unit*** [ ***quantity*** ***unit*** ...] [ T [ ***quantity*** ***unit*** ...]]

The string must start with a P, and may include a T that introduces the time-of-day units. The available unit abbreviations are given in [**Table 8-16**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-INTERVAL-ISO8601-UNITS). Units may be omitted, and may be specified in any order, but units smaller than a day must appear after T. In particular, the meaning of M depends on whether it is before or after T.

### Table 8-16. ISO 8601 Interval Unit Abbreviations

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| Y | Years |
| M | Months (in the date part) |
| W | Weeks |
| D | Days |
| H | Hours |
| M | Minutes (in the time part) |
| S | Seconds |

In the alternative format:

P [ ***years***-***months***-***days*** ] [ T ***hours***:***minutes***:***seconds*** ]

the string must begin with P, and a T separates the date and time parts of the interval. The values are given as numbers similar to ISO 8601 dates.

When writing an interval constant with a ***fields*** specification, or when assigning a string to an interval column that was defined with a ***fields*** specification, the interpretation of unmarked quantities depends on the ***fields***. For example INTERVAL '1' YEAR is read as 1 year, whereas INTERVAL '1' means 1 second. Also, field values "to the right" of the least significant field allowed by the ***fields*** specification are silently discarded. For example, writing INTERVAL '1 day 2:03:04' HOUR TO MINUTE results in dropping the seconds field, but not the day field.

According to the SQL standard all fields of an interval value must have the same sign, so a leading negative sign applies to all fields; for example the negative sign in the interval literal '-1 2:03:04' applies to both the days and hour/minute/second parts. PostgreSQL allows the fields to have different signs, and traditionally treats each field in the textual representation as independently signed, so that the hour/minute/second part is considered positive in this example. If IntervalStyle is set to sql\_standard then a leading sign is considered to apply to all fields (but only if no additional signs appear). Otherwise the traditional PostgreSQL interpretation is used. To avoid ambiguity, it's recommended to attach an explicit sign to each field if any field is negative.

In the verbose input format, and in some fields of the more compact input formats, field values can have fractional parts; for example '1.5 week' or '01:02:03.45'. Such input is converted to the appropriate number of months, days, and seconds for storage. When this would result in a fractional number of months or days, the fraction is added to the lower-order fields using the conversion factors 1 month = 30 days and 1 day = 24 hours. For example, '1.5 month' becomes 1 month and 15 days. Only seconds will ever be shown as fractional on output.

[**Table 8-17**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#DATATYPE-INTERVAL-INPUT-EXAMPLES) shows some examples of valid interval input.

### Table 8-17. Interval Input

|  |  |
| --- | --- |
| **Example** | **Description** |
| 1-2 | SQL standard format: 1 year 2 months |
| 3 4:05:06 | SQL standard format: 3 days 4 hours 5 minutes 6 seconds |
| 1 year 2 months 3 days 4 hours 5 minutes 6 seconds | Traditional Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds |
| P1Y2M3DT4H5M6S | ISO 8601 "format with designators": same meaning as above |
| P0001-02-03T04:05:06 | ISO 8601 "alternative format": same meaning as above |

Internally interval values are stored as months, days, and seconds. This is done because the number of days in a month varies, and a day can have 23 or 25 hours if a daylight savings time adjustment is involved. The months and days fields are integers while the seconds field can store fractions. Because intervals are usually created from constant strings or timestamp subtraction, this storage method works well in most cases, but can cause unexpected results:

SELECT EXTRACT(hours from '80 minutes'::interval);

date\_part

-----------

1

SELECT EXTRACT(days from '80 hours'::interval);

date\_part

-----------

0

Functions justify\_days and justify\_hours are available for adjusting days and hours that overflow their normal ranges.

**8.5.5. Interval Output**

The output format of the interval type can be set to one of the four styles sql\_standard, postgres, postgres\_verbose, or iso\_8601, using the command SET intervalstyle. The default is the postgres format. [**Table 8-18**](https://www.postgresql.org/docs/9.3/datatype-datetime.html#INTERVAL-STYLE-OUTPUT-TABLE) shows examples of each output style.

The sql\_standard style produces output that conforms to the SQL standard's specification for interval literal strings, if the interval value meets the standard's restrictions (either year-month only or day-time only, with no mixing of positive and negative components). Otherwise the output looks like a standard year-month literal string followed by a day-time literal string, with explicit signs added to disambiguate mixed-sign intervals.

The output of the postgres style matches the output of PostgreSQL releases prior to 8.4 when the [**DateStyle**](https://www.postgresql.org/docs/9.3/runtime-config-client.html#GUC-DATESTYLE) parameter was set to ISO.

The output of the postgres\_verbose style matches the output of PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to non-ISO output.

The output of the iso\_8601 style matches the "format with designators" described in section 4.4.3.2 of the ISO 8601 standard.

### Table 8-18. Interval Output Style Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Style Specification** | **Year-Month Interval** | **Day-Time Interval** | **Mixed Interval** |
| sql\_standard | 1-2 | 3 4:05:06 | -1-2 +3 -4:05:06 |
| postgres | 1 year 2 mons | 3 days 04:05:06 | -1 year -2 mons +3 days -04:05:06 |
| postgres\_verbose | @ 1 year 2 mons | @ 3 days 4 hours 5 mins 6 secs | @ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago |
| iso\_8601 | P1Y2M | P3DT4H5M6S | P-1Y-2M3DT-4H-5M-6S |