PostgreSQL

Prepared by Sout Saret

Position: Senior Database Administrator

Introduction

- PostgreSQL is an Object-relational Database Management System (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department.
- PostgreSQL supports a large part of the SQL standard and offers many modern features:

  - Complex Queries

  - Foreign Keys

  - Triggers

  - Updatable Views

  - Transactional Integrity

  - Multi Version Concurrency Control (MVCC)
- PostgreSQL can be extended by the user in many ways, for example by adding new:

  - Data Types

  - Functions

  - Operations

  - Aggregate Functions

  - Index Method

  -Procedural Language

History

- With over two decades of development behind it, PostgreSQL is now the most advanced open-source database available anywhere.
- The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc. The implementation of POSTGRES began in 1986.
- In 1994, Andrew Yu and Jolly Chen added an SQL language interpreter to POSTGRES. Under a new name, Postgres95 was subsequently released to the web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.
- In 1995, POSTGRES was renamed as PostgreSQL.

Installation

- To make an installation PostgreSQL. We have two ways:

  1. Install PostgreSQL from source code:

  You can go to website to download from http://www.postgresql.org/ftp/source

  2. Install PostgreSQL from file:

  You can go to website to download binary file from http://www.postgresql.org/download/.

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

- A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server program is called postgres.

- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

Manipulated Database

- To create a database, we have two ways to create:

  1. First way: we can use by command to create:

  Ex: createdb dbname

  2. Second way: we can use by sql standard to create:

  Ex: create database dbname;

- To drop a database, we have two ways to drop:

  1. First way: we can use by command to drop:

  Ex: dropdb dbname

  2. Second way: we can use by sql standard to drop:

  Ex: drop database dbname;

Accessing Database

- To access to PostgreSQL database, we have several ways:

  1. Using PostgreSQL Terminal:

  PostgreSQL provides a terminal to access, is called psql.

  $ psql dbname

  2. Using GUI Tools:

  GUI Tools that can access to PostgreSQL such as Navicat, PgAdmin, PHPPgAdmin, …

  3. Using Customize Applications:

  Using one of the several available language bindings.

SQL Language

- Every Databases, they have to follow the SQL standard. The SQL provides as below:
  1. Data Definition Language (DDL)
  2. Data Manipulation Language (DML)
  3. Data Control Language (DCL)
  4. Transaction Control (TCL)

DDL

- Data Definition Language (DDL) statements are used to define the database structure or schema.

  Some examples:

  1. CREATE - to create objects in the database
  2. ALTER - alters the structure of the database
  3. DROP - delete objects from the database
  4. TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
  5. COMMENT - add comments to the data dictionary
  6. RENAME - rename an object

DML

- Data Manipulation Language (DML) statements are used for managing data within schema objects.

  Some examples:

  1. SELECT - retrieve data from the a database
  2. INSERT - insert data into a table
  3. UPDATE - updates existing data within a table
  4. DELETE - deletes all records from a table, the space for the records remain
  5. MERGE - UPSERT operation (insert or update)
  6. CALL - call a PL/SQL or Java subprogram
  7. EXPLAIN PLAN - explain access path to data
  8. LOCK TABLE - control concurrency

DCL

- Data Control Language (DCL) statements.

  Some examples:

  1. GRANT - gives user's access privileges to database

  2. REVOKE - withdraw access privileges given with the GRANT command

TCL

- Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

  Some example:

  1. COMMIT - save work done
  2. SAVEPOINT - identify a point in a transaction to which you can later roll back
  3. ROLLBACK - restore database to original since the last COMMIT
  4. SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

Creating a New Table

- You can create a table using CREATE keyword.

Syntax:

```
CREATE TABLE table_name (
        field_name data_type constrain_name,
        field_name1 data_type1 constrain_name1
);
```

Exampple:

```
CREARE TABLE employee (
        emp_id      integer PRIMARY KEY,
        emp_name    varchar(50) NOT NULL,
        emp_gender char(1) NOT NULL,
        emp_description varchar(255)
);
```

Inserting Data into Table

- To insert data into table, we can use INSERT keyword:

  We can insert data into table with different ways:

  Ex:

  - INSERT INTO table_name VALUES (value1,value2,value3); // Insert all fields in table
  - INSERT INTO table_name (field1, field2, field3) VALUES (value1,value2,value3); // Insert specific fields in table
  - INSERT INTO table_name VALUES (value1,value2,value3), (value4,value5,value6), (value7,value8,value9);
  - COPY weather FROM '/home/user/table_name.txt'; // Insert from file into table

Display Data (Querying Data)

- To retrieved data from table, we can use SELECT keyword.

  Example:

  - SELECT * FROM table_name;

  - SELECT field_name1, field_name2, field_name3 FROM table_name;

  - SELECT * FROM table_name WHERE condition;

  - SELECT * FROM table_name WHERE condition ORDER BY field_name ASC/DESC;

  - SELECT * FROM table_name WHERE condition GROUP BY field_name ORDER BY field_name ASC/DESC;

Updating Data (Modified Data)

- To update or modify data in table, we have to use UPDATE key word:

  Example:

  - UPDATE table_name SET field_name= value; // Update or Modify all data in a column

  - UPDATE table_name SET field_name= value WHERE condition; // Modify with condtion

Deleting Data

- To clear or delete data from table, we have to use DELETE keywork:

  Example:

  - DELETE FROM table_name; // Clear all Data from table

  - DELECT FROM table_name WHERE condition; // Delete data from table with condition

Joining Table

- Normally, we can query data from one table or multi tables at one time. So this is some examples that is used join:

  Example:

  - SELECT * FROM table_name; // No join table

  - SELECT * FROM table_name1, table_name2 WHERE table_name1.field_name=table_name2.field_name;

  - SELECT * FROM table_name1 JOIN table_name2 ON(table_name1.field_name=table_name2.field_name);

  - SELECT * FROM table_name1 JOIN table_name2 ON(table_name1.field_name=table_name2.field_name) WHERE condition;

Creating Views

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

  Example:

  ```
  - CREATE VIEW view_name AS
  SELECT * FROM table_name1
  JOIN table_name2 ON(table_name1.field_name=table_name2.field_name)
  JOIN table_name2 ON(table_name1.field_name=table_name3.field_name)
  WHERE codition;
  ```

- To show data from view:

  Example:

  ```
  - SELECT * FROM view_name;
  ```

Putting Primary Key & Foreign Key

- We need primary key to unique name data in field of tables:

  Example:

  - CREATE TABLE parents(

  p_id int primarykey,

  name varchar(50)

  );

- We need foreign key to make relationship between parents table and child table or another tables:

  Example:

  - CREATE TABLE children(

  c_id int primary key,

  p_id int references parents(p_id),

  name varchar(50)

  );

Transaction

- Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.
- For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account.

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

Window Function

- A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.
- A window function call always contains an OVER clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a regular function or aggregate function. The OVER clause determines exactly how the rows of the query are split up for processing by the window function.
- The PARTITION BY list within OVER specifies dividing the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

Inheritance

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

Example:

```
- CREATE TABLE parents(
p_id int,
name varchar(30)
)
- CREATE TABLE children(
c_id int,
name varchar(30)
) inherits (parents);
```

Note: Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness.

Value Expressions

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

A value expression is one of the following:

- A constant or literal value
- A column reference
- A positional parameter reference, in the body of a function definition or prepared statement
- A subscripted expression
- A field selection expression
- An operator invocation
- A function call
- An aggregate expression
- A window function call
- A type cast
- A collation expression
- A scalar subquery
- An array constructor
- A row constructor

Type Casts

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

  1. Using CAST keyword:

  Ex: SELECT CAST(field_name AS TYPE) FROM table_name;

  2. Using Symbol (::):

  Ex: SELECT field_name::TYPE FROM table_name;

Scalar Subqueries

- A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column.
- The SELECT query is executed and the single returned value is used in the surrounding value expression.
- It is an error to use a query that returns more than one row or more than one column as a scalar subquery.
- The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

Ex: SELECT field_name, (SELECT fiel_name FROM table_name2 WHERE table_name2.field_name=table_name1.field_name) FROM table_name1;

Calling Function

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

Ex: CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false) RETURNS text AS $$

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

- Using Positional Notation: SELECT concat_lower_or_upper('Hello', 'World', true); or SELECT concat_lower_or_upper('Hello', 'World');

- Using Named Notation: SELECT concat_lower_or_upper(a := 'Hello', b := 'World'); Or SELECT concat_lower_or_upper(a := 'Hello', b := 'World', uppercase := true); OR SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
- Using mix: SELECT concat_lower_or_upper('Hello', 'World', uppercase := true);

Default Values

- A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is.
- If no default value is declared explicitly, the default value is the null value. This usually makes sense because a null value can be considered to represent unknown data.

Example:

```
CREATE TABLE products (
pro_no integer primary key,
 name text,
price numeric DEFAULT 9.99
);
```

Constraints

- Constraints give you as much control over the data in your tables as you wish.
- A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression.

Ex: CREATE TABLE products (pro_no integer, name text, price numeric CHECK (price > 0));

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

Ex: CREATE TABLE products (pro_no integer NOT NULL, name text NOT NULL, price numeric);

- Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

Ex: CREATE TABLE products (pro_no integer UNIQUE, name text, price numeric);

- A primary key constraint is simply a combination of a unique constraint and a not-null constraint.

Ex: CREATE TABLE products (pro_no integer PRIMARY KEY, name text, price numeric);

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

Ex: CREATE TABLE orders (order_id integer PRIMARY KEY, pro_no integer REFERENCES products (pro_no), quantity integer);

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

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

System Columns

- Every table has several system columns that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns.
- **oid**: The object identifier (object ID) of a row. This column is only present if the table was created using WITH OIDS, or if the default_with_oids configuration variable was set at the time.
- **tableoid**: The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies.
- **xmin**: The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.).
- **cmin**: The command identifier (starting at zero) within the inserting transaction.
- **xmax**: The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.
- **cmax**: The command identifier within the deleting transaction, or zero.
- **ctid**: The ctid can be used to locate the row version very quickly, a row's ctid will change if it is updated or moved by VACUUM FULL. Therefore ctid is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

Modifying Tables

- PostgreSQL provides a family of commands to make modifications to existing tables.

We can do:

- Add columns: Ex: ALTER TABLE products ADD COLUMN description text;
- Removing a Column: Ex: ALTER TABLE products DROP COLUMN description;
- Adding a Constraint: Ex: ALTER TABLE products ADD CHECK (name <> '');

  ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (pro_no);

  ALTER TABLE products ADD FOREIGN KEY (pro_group_id) REFERENCES product_groups;
- Removing a Constraint: Ex: ALTER TABLE products DROP CONSTRAINT some_name;
- Changing a Column's Default Value: Ex: ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
- Changing a Column's Data Type: Ex: ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
- Renaming a Column: Ex: ALTER TABLE products RENAME COLUMN product_no TO product_number;
- Renaming a Table: Ex: ALTER TABLE products RENAME TO items;

Privileges

- There are different kinds of privileges: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, and USAGE.
- To assign privileges:

  Ex:

  ```
  GRANT UPDATE ON table_name TO user_name;

  GRANT ALL PRIVILEGES ON table_name TO user_name;
  ```

- To revoke a privilege:

  Ex:

  ```
  REVOKE ALL ON table_name FROM PUBLIC;
  ```

Schemas

- A database contains one or more named schemas, which in turn contain tables.
- There are several reasons why one might want to use schemas:

  - To allow many users to use one database without interfering with each other.

  - To organize database objects into logical groups to make them more manageable.

  - Third-party applications can be put into separate schemas so they do not collide with the names of other objects.
- To create a schema: Ex: CREATE SCHEMA schema_name;
- The Schema Search Path

  - To show the current search path: Ex: SHOW search_path;

  - To put our new schema in the path: Ex: SET search_path TO myschema,public;

Partitioning

- Partitioning refers to splitting what is logically one large table into smaller physical pieces.
- Partitioning can provide several benefits:

  - Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. The partitioning substitutes for leading columns of indexes, reducing index size and making it more likely that the heavily-used parts of the indexes fit in memory.

  - When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of using an index and random access reads scattered across the whole table.

  - Bulk loads and deletes can be accomplished by adding or removing partitions, if that requirement is planned into the partitioning design. ALTER TABLE NO INHERIT and DROP TABLE are both far faster than a bulk operation. These commands also entirely avoid the VACUUM overhead caused by a bulk DELETE.

  - Seldom-used data can be migrated to cheaper and slower storage media.

- The following forms of partitioning can be implemented in PostgreSQL:

  - **Range Partitioning**: The table is partitioned into "ranges" defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example one might partition by date ranges, or by ranges of identifiers for particular business objects.

  - **List Partitioning**: The table is partitioned by explicitly listing which key values appear in each partition.

Foreign Data

- PostgreSQL implements portions of the SQL/MED specification, allowing you to access data that resides outside PostgreSQL using regular SQL queries.
- Foreign data is accessed with help from a foreign data wrapper.
- A foreign data wrapper is a library that can communicate with an external data source, hiding the details of connecting to the data source and obtaining data from it.
- To access foreign data, you need to create a foreign server object, which defines how to connect to a particular external data source according to the set of options used by its supporting foreign data wrapper. Then you need to create one or more foreign tables, which define the structure of the remote data.
- A foreign table can be used in queries just like a normal table, but a foreign table has no storage in the PostgreSQL server.

Table Expressions

- The table expression contains a FROM clause that is optionally followed by WHERE, GROUP BY, and HAVING clauses.
- Join Types:

  - Cross join: Ex: T1 CROSS JOIN T2

  - Qualified joins:

  Ex:

  - T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression

  - T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING (join column list)

  - T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
- The possible types of qualified join are:

- INNER JOIN: For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

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

- 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.

- 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.

Table Functions

- Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows).
- They are used like a table, view, or subquery in the FROM clause of a query.
- Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as columns of a table, view, or subquery.

Example:

```
-CREATE TABLE foo (fooid int, foosubid int, fooname text);
- CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$     SELECT * FROM foo WHERE fooid = $1; $$ LANGUAGE SQL;
- SELECT * FROM getfoo(1) AS t1;
- SELECT * FROM foo WHERE foosubid IN (SELECT foosubid FROM getfoo(foo.fooid) z WHERE z.fooid = foo.fooid);
- CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
-SELECT * FROM vw_getfoo;
```

LATERAL Subqueries

- 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:

  Example: SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;

- LATERAL is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined. A common application is providing an argument value for a set-returning function.

Example:

- SELECT p1.id, p2.id, v1, v2 FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1, polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2 WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;

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:

  Syntax:

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

- OFFSET says to skip that many rows before beginning to return rows.
- OFFSET 0 is the same as omitting the OFFSET clause, and LIMIT NULL is the same as omitting the LIMIT clause.
- If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

WITH Queries (Common Table Expressions)

- WITH provides a way to write auxiliary statements for use in a larger 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.
- Recursive Query Evaluation:

1- Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.

2- So long as the working table is not empty, repeat these steps:

- Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.

- Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(*n*)]		fixed-length bit string
bit varying [(*n*)]	varbit	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data ("byte array")
character [(*n*)]	char [(*n*)]	fixed-length character string
character varying [(*n*)]	varchar [(*n*)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [*fields*] [(*p*)]		time span
json		textual JSON data
jsonb		binary JSON data, decomposed
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
money		currency amount

Name	Aliases	Description
numeric [(p, s)]	decimal [(p, s)]	exact numeric of selectable precision
path		geometric path on a plane
pg_lsn		PostgreSQL Log Sequence Number
point		geometric point on a plane
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(p)] [without time zone]		time of day (no time zone)
time [(p)] with time zone	timetz	time of day, including time zone
timestamp [(p)] [without time zone]		date and time (no time zone)
timestamp [(p)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot
uuid		universally unique identifier
xml		XML data

Numeric Types

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetary Types

- The money type stores a currency amount with a fixed fractional precision.

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

- Conversion from the real and double precision data types can be done by casting to numeric first:

Example:

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

- Conversion to other types could potentially lose precision, and must also be done in two stages:

Example:

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

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

Special Character Types

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

Binary Data Types

- The bytea data type allows storage of binary strings.

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

- A binary string is a sequence of octets (or bytes).
- bytea Literal Escaped Octets:

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

- bytea Output Escaped Octets:

Decimal Octet Value	Description	Escaped Output Representation	Example	Output Result
92	backslash	\\	SELECT E'\\134'::bytea;	\\
0 to 31 and 127 to 255	"non-printable" octets	\xxx (octal value)	SELECT E'\\001'::bytea;	\001
32 to 126	"printable" octets	client character set representation	SELECT E'\\176'::bytea;	~

Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
`timestamp [ (p) ] [ without time zone ]`	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
`timestamp [ (p) ] with time zone`	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits
`date`	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
`time [ (p) ] [ without time zone ]`	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
`time [ (p) ] with time zone`	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits
`interval [ fields ] [ (p) ]`	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits

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

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

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`

Special Values

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

Date/Time Output

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

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

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

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

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

Boolean Type

Name	Storage Size	Description
`boolean`	1 byte	state of true or false

Valid literal values for the "true" state are:

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

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

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

Enumerated Types

- Enumerated (enum) types are data types that comprise a static, ordered set of values.
- Declaration of Enumerated Types:

Example:

```
- CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
- CREATE TABLE person (
     name text,
     current_mood mood
  );
- INSERT INTO person VALUES ('Moe', 'happy');
- SELECT * FROM person WHERE current_mood = 'happy';
```

Geometric Types

- Geometric data types represent two-dimensional spatial objects.

Name	Storage Size	Description	Representation
`point`	16 bytes	Point on a plane	(x,y)
`line`	32 bytes	Infinite line	{A,B,C}
`lseg`	32 bytes	Finite line segment	((x1,y1),(x2,y2))
`box`	32 bytes	Rectangular box	((x1,y1),(x2,y2))
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`circle`	24 bytes	Circle	<(x,y),r> (center point and radius)

Network Address Types

Name	Storage Size	Description
`cidr`	7 or 19 bytes	IPv4 and IPv6 networks
`inet`	7 or 19 bytes	IPv4 and IPv6 hosts and networks
`macaddr`	6 bytes	MAC addresses

- The inet type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. The subnet is represented by the number of network address bits present in the host address (the "netmask").
- The cidr type holds an IPv4 or IPv6 network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying networks is address/y where address is the network represented as an IPv4 or IPv6 address, and y is the number of bits in the netmask.
- The macaddr type stores MAC addresses, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well).

cidr Type Input Examples

cidr Input	cidr Output	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

Bit String Types

- Bit strings are strings of 1's and 0's.
- There are two SQL bit types: bit(n) and bit varying(n), where n is a positive integer.
- bit type data must match the length n exactly; it is an error to attempt to store shorter or longer bit strings.
- bit varying data is of variable length up to the maximum length n; longer strings will be rejected.
- Using the Bit String Types:

Example:

```
- CREATE TABLE test (a BIT(3), b BIT VARYING(5));
- INSERT INTO test VALUES (B'101', B'00');
- INSERT INTO test VALUES (B'10', B'101');
  ERROR:  bit string length 2 does not match type bit(3)
- INSERT INTO test VALUES (B'10'::bit(3), B'101');
- SELECT * FROM test;
```

Text Search Types

- PostgreSQL provides two data types that are designed to support full text search are **tsvector** and **tsquery**.
- A tsvector value is a sorted list of distinct lexemes, which are words that have been normalized to merge different variants of the same word.

Example: SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;

- Raw document text should usually be passed through to_tsvector to normalize the words appropriately for searching:

Example: SELECT to_tsvector('english', 'The Fat Rats');

- A tsquery value stores lexemes that are to be searched for, and combines them honoring the Boolean operators & (AND), | (OR), and ! (NOT).

Example: SELECT 'fat & rat'::tsquery;

- The to_tsquery function is convenient for performing such normalization:

Example: SELECT to_tsquery('Fat:ab & Cats');

UUID Type

- The data type uuid stores Universally Unique Identifiers (UUID) as defined by RFC 4122, ISO/IEC 9834-8:2005, and related standards.
- This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm.
- A UUID is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of 8 digits followed by three groups of 4 digits followed by a group of 12 digits, for a total of 32 digits representing the 128 bits.
- An example of a UUID in this standard form is:

  a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

XML Type

- The xml data type can be used to store XML data.
- Its advantage over storing XML data in a text field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it.
- To produce a value of type xml from character data, use the function xmlparse:

Syntax: XMLPARSE ({ DOCUMENT | CONTENT } value)

Examples:

XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')

- The inverse operation, producing a character string value from xml, uses the function xmlserialize:

Syntax: XMLSERIALIZE ({ DOCUMENT | CONTENT } value AS type)

- When a character string value is cast to or from type xml without going through XMLPARSE or XMLSERIALIZE, respectively, the choice of DOCUMENT versus CONTENT is determined by the "XML option" session configuration parameter, which can be set using the standard command:

Syntax: SET XML OPTION { DOCUMENT | CONTENT };

Ex: SET xmloption TO { DOCUMENT | CONTENT };

JSON Types

- JSON data types are for storing JSON (JavaScript Object Notation) data.
- here are two JSON data types: json and jsonb.
- The json data type stores an exact copy of the input text, which processing functions must reparse on each execution.
- The jsonb data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed.
- JSON primitive types and corresponding PostgreSQL types:

JSON primitive type	PostgreSQL type	Notes
`string`	`text`	See notes above concerning encoding restrictions
`number`	`numeric`	`NaN` and `infinity` values are disallowed
`boolean`	`boolean`	Only lowercase `true` and `false` spellings are accepted
`null`	(none)	SQL `NULL` is a different concept

Arrays

- An array data type is named by appending square brackets ([]) to the data type name of the array elements.
- To illustrate the use of array types, we create this table:

Ex: CREATE TABLE table_arry(id integer, name text, pay_by_quarter [], schedule text [][]);

- To set an element of an array constant to NULL, write NULL for the element value. (Any upper- or lower-case variant of NULL will do.) If you want an actual string value "NULL", you must put double quotes around it.

Composite Types

- A composite type represents the structure of a row or record; it is essentially just a list of field names and their data types.

Syntax:

- CREATE TYPE composite_name AS (r double precision, I double precision);
- CREATE TYPE inventory_item AS (name text, supplier_id integer, price numeric);
- CREATE TABLE on_hand (item inventory_item, count integer);
- INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);

- To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name.

Example: SELECT item.name FROM on_hand WHERE item.price > 9.99;

Range Types

- Range types are data types representing a range of values of some element type (called the range's subtype).
- For instance, ranges of timestamp might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is tsrange (short for "timestamp range"), and timestamp is the subtype.
- The subtype must have a total order so that it is well-defined whether element values are within, before, or after a range of values.
- Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly.
- The following built-in range types:

  - int4range — Range of integer
  - int8range — Range of bigint
  - numrange — Range of numeric
  - tsrange — Range of timestamp without time zone
  - tstzrange — Range of timestamp with time zone
  - daterange — Range of date

Object Identifier Types

- Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables.
- Type oid represents an object identifier.
- There are also several alias types for oid: regproc, regprocedure, regoper, regoperator, regclass, regtype, regconfig, and regdictionary.

Name	References	Description	Value Example
`oid`	any	numeric object identifier	`564182`
`regproc`	`pg_proc`	function name	`sum`
`regprocedure`	`pg_proc`	function with argument types	`sum(int4)`
`regoper`	`pg_operator`	operator name	`+`
`regoperator`	`pg_operator`	operator with argument types	`*(integer,integer)` or `-(NONE,integer)`
`regclass`	`pg_class`	relation name	`pg_type`
`regtype`	`pg_type`	data type name	`integer`
`regconfig`	`pg_ts_config`	text search configuration	`english`
`regdictionary`	`pg_ts_dict`	text search dictionary	`simple`

pg_lsn Type

- The pg_lsn data type can be used to store LSN (Log Sequence Number) data which is a pointer to a location in the XLOG.
- This type is a representation of XLogRecPtr and an internal system type of PostgreSQL.
- An LSN is a 64-bit integer, representing a byte position in the write-ahead log stream.
- The pg_lsn type supports the standard comparison operators, like = and >.
- Two LSNs can be subtracted using the - operator; the result is the number of bytes separating those write-ahead log positions.

Pseudo-Types

- A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type.

Name	Description
`any`	Indicates that a function accepts any input data type.
`anyelement`	Indicates that a function accepts any data type (see Section 35.2.5).
`anyarray`	Indicates that a function accepts any array data type (see Section 35.2.5).
`anynonarray`	Indicates that a function accepts any non-array data type (see Section 35.2.5).
`anyenum`	Indicates that a function accepts any enum data type (see Section 35.2.5 and Section 8.7).
`anyrange`	Indicates that a function accepts any range data type (see Section 35.2.5 and Section 8.17).
`cstring`	Indicates that a function accepts or returns a null-terminated C string.
`internal`	Indicates that a function accepts or returns a server-internal data type.
`language_handler`	A procedural language call handler is declared to return `language_handler`.
`fdw_handler`	A foreign-data wrapper handler is declared to return `fdw_handler`.
`record`	Identifies a function returning an unspecified row type.
`trigger`	A trigger function is declared to return `trigger`.
`void`	Indicates that a function returns no value.
`opaque`	An obsolete type name that formerly served all the above purposes.

Mathematical Operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (integer division truncates the result)	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
|/	square root	|/ 25.0	5
||/	cube root	||/ 27.0	3
!	factorial	5 !	120
!!	factorial (prefix operator)	!! 5	120
@	absolute value	@ -5.0	5
&	bitwise AND	91 & 15	11
|	bitwise OR	32 | 3	35
#	bitwise XOR	17 # 5	20
~	bitwise NOT	~1	-2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

Mathematical Functions

Function	Return Type	Description	Example	Result
abs(x)	(same as input)	absolute value	abs(-17.4)	17.4
cbrt(dp)	dp	cube root	cbrt(27.0)	3
ceil(dp or numeric)	(same as input)	smallest integer not less than argument	ceil(-42.8)	-42
ceiling(dp or numeric)	(same as input)	smallest integer not less than argument (alias for ceil)	ceiling(-95.3)	-95
degrees(dp)	dp	radians to degrees	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	integer quotient of y/x	div(9,4)	2
exp(dp or numeric)	(same as input)	exponential	exp(1.0)	2.71828182845905
floor(dp or numeric)	(same as input)	largest integer not greater than argument	floor(-42.8)	-43
ln(dp or numeric)	(same as input)	natural logarithm	ln(2.0)	0.693147180559945
log(dp or numeric)	(same as input)	base 10 logarithm	log(100.0)	2
log(b numeric, x numeric)	numeric	logarithm to base b	log(2.0, 64.0)	6.0000000000
mod(y, x)	(same as argument types)	remainder of y/x	mod(9,4)	1
pi()	dp	"π" constant	pi()	3.14159265358979
power(a dp, b dp)	dp	a raised to the power of b	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	a raised to the power of b	power(9.0, 3.0)	729
radians(dp)	dp	degrees to radians	radians(45.0)	0.785398163397448
round(dp or numeric)	(same as input)	round to nearest integer	round(42.4)	42
round(v numeric, s int)	numeric	round to s decimal places	round(42.4382, 2)	42.44
sign(dp or numeric)	(same as input)	sign of the argument (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp or numeric)	(same as input)	square root	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	(same as input)	truncate toward zero	trunc(42.8)	42
trunc(v numeric, s int)	numeric	truncate to s decimal places	trunc(42.4382, 2)	42.43
width_bucket(op numeric, b1 numeric, b2 numeric, count int)	int	return the bucket to which operand would be assigned in an equidepth histogram with count buckets, in the range b1 to b2	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(op dp, b1 dp, b2 dp, count int)	int	return the bucket to which operand would be assigned in an equidepth histogram with count buckets, in the range b1 to b2	width_bucket(5.35, 0.024, 10.06, 5)	3

Random Functions

Function	Return Type	Description
`random()`	dp	random value in the range 0.0 <= x < 1.0
`setseed(dp)`	void	set seed for subsequent `random()` calls (value between -1.0 and 1.0, inclusive)

Table 9-5. Trigonometric Functions

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

String Functions and Operators

<table>
<tr><th>Function</th><th>Return Type</th><th>Description</th><th>Example</th><th>Result</th></tr>
<tr><td>string || string</td><td>text</td><td>String concatenation</td><td>'Post' || 'greSQL'</td><td>PostgreSQL</td></tr>
<tr><td>string || non-string or non-string || string</td><td>text</td><td>String concatenation with one non-string input</td><td>'Value: ' || 42</td><td>Value: 42</td></tr>
<tr><td>bit_length(string)</td><td>int</td><td>Number of bits in string</td><td>bit_length('jose')</td><td>32</td></tr>
<tr><td>char_length(string) or character_length(string)</td><td>int</td><td>Number of characters in string</td><td>char_length('jose')</td><td>4</td></tr>
<tr><td>lower(string)</td><td>text</td><td>Convert string to lower case</td><td>lower('TOM')</td><td>tom</td></tr>
<tr><td>octet_length(string)</td><td>int</td><td>Number of bytes in string</td><td>octet_length('jose')</td><td>4</td></tr>
<tr><td>overlay(string placing string from int [for int])</td><td>text</td><td>Replace substring</td><td>overlay('Txxxxas' placing 'hom' from 2 for 4)</td><td>Thomas</td></tr>
<tr><td>position(substring in string)</td><td>int</td><td>Location of specified substring</td><td>position('om' in 'Thomas')</td><td>3</td></tr>
<tr><td>substring(string [from int] [for int])</td><td>text</td><td>Extract substring</td><td>substring('Thomas' from 2 for 3)</td><td>hom</td></tr>
<tr><td>substring(string from pattern)</td><td>text</td><td>Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.</td><td>substring('Thomas' from '...$')</td><td>mas</td></tr>
<tr><td>substring(string from pattern for escape)</td><td>text</td><td>Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.</td><td>substring('Thomas' from '%#"o_a#"_' for '#')</td><td>oma</td></tr>
<tr><td>trim([leading | trailing | both] [characters] from string)</td><td>text</td><td>Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string</td><td>trim(both 'x' from 'xTomxx')</td><td>Tom</td></tr>
<tr><td>trim([leading | trailing | both] [from] string [, characters])</td><td>text</td><td>Non-standard version of trim()</td><td>trim(both from 'xTomxx', 'x')</td><td>Tom</td></tr>
<tr><td>upper(string)</td><td>text</td><td>Convert string to upper case</td><td>upper('tom')</td><td>TOM</td></tr>
</table>

Bit String Functions and Operators

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th><th>Result</th></tr>
<tr><td>||</td><td>concatenation</td><td>B'10001' || B'011'</td><td>10001011</td></tr>
<tr><td>&</td><td>bitwise AND</td><td>B'10001' & B'01101'</td><td>00001</td></tr>
<tr><td>|</td><td>bitwise OR</td><td>B'10001' | B'01101'</td><td>11101</td></tr>
<tr><td>#</td><td>bitwise XOR</td><td>B'10001' # B'01101'</td><td>11100</td></tr>
<tr><td>~</td><td>bitwise NOT</td><td>~ B'10001'</td><td>01110</td></tr>
<tr><td><<</td><td>bitwise shift left</td><td>B'10001' << 3</td><td>01000</td></tr>
<tr><td>>></td><td>bitwise shift right</td><td>B'10001' >> 2</td><td>00100</td></tr>
</table>

Data Type Formatting Functions

Function	Return Type	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(current_timestamp, 'HH12:MI:SS')`
`to_char(interval, text)`	`text`	convert interval to string	`to_char(interval '15h 2m 12s', 'HH24:MI:SS')`
`to_char(int, text)`	`text`	convert integer to string	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	convert real/double precision to string	`to_char(125.8::real, '999D9')`
`to_char(numeric, text)`	`text`	convert numeric to string	`to_char(-125.8, '999D99S')`
`to_date(text, text)`	`date`	convert string to date	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_number(text, text)`	`numeric`	convert string to numeric	`to_number('12,454.8-', '99G999D9S')`
`to_timestamp(text, text)`	`timestamp with time zone`	convert string to time stamp	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`
`to_timestamp(double precision)`	`timestamp with time zone`	convert Unix epoch to time stamp	`to_timestamp(1284352323)`

Date/Time Operators

Operator	Example	Result
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Enum Support Functions

- There are several functions that allow cleaner programming without hard-coding particular values of an enum type.

Example:

CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');

Function	Description	Example	Example Result
`enum_first(anyenum)`	Returns the first value of the input enum type	`enum_first(null::rainbow)`	`red`
`enum_last(anyenum)`	Returns the last value of the input enum type	`enum_last(null::rainbow)`	`purple`
`enum_range(anyenum)`	Returns all values of the input enum type in an ordered array	`enum_range(null::rainbow)`	`{red,orange,yellow,green,blue,purple}`
`enum_range(anyenum, anyenum)`	Returns the range between the two given enum values, as an ordered array. The values must be from the same enum type. If the first parameter is null, the result will start with the first value of the enum type. If the second parameter is null, the result will end with the last value of the enum type.	`enum_range('orange'::rainbow, 'green'::rainbow)`	`{orange,yellow,green}`
		`enum_range(NULL, 'green'::rainbow)`	`{red,orange,yellow,green}`
		`enum_range('orange'::rainbow, NULL)`	`{orange,yellow,green,blue,purple}`

Geometric Functions

Function	Return Type	Description	Example
area(*object*)	double precision	area	area(box '((0,0),(1,1))')
center(*object*)	point	center	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(*object*)	double precision	length	length(path '((-1,0),(1,0))')
npoints(path)	int	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	number of points	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	convert path to closed	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')
width(box)	double precision	horizontal size of box	width(box '((0,0),(1,1))')

Geometric Type Conversion Functions

Function	Return Type	Description	Example
`box(circle)`	`box`	circle to box	`box(circle '((0,0),2.0)')`
`box(point, point)`	`box`	points to box	`box(point '(0,0)', point '(1,1)')`
`box(polygon)`	`box`	polygon to box	`box(polygon '((0,0),(1,1),(2,0))')`
`circle(box)`	`circle`	box to circle	`circle(box '((0,0),(1,1))')`
`circle(point, double precision)`	`circle`	center and radius to circle	`circle(point '(0,0)', 2.0)`
`circle(polygon)`	`circle`	polygon to circle	`circle(polygon '((0,0),(1,1),(2,0))')`
`line(point, point)`	`line`	points to line	`line(point '(-1,0)', point '(1,0)')`
`lseg(box)`	`lseg`	box diagonal to line segment	`lseg(box '((-1,0),(1,0))')`
`lseg(point, point)`	`lseg`	points to line segment	`lseg(point '(-1,0)', point '(1,0)')`
`path(polygon)`	`path`	polygon to path	`path(polygon '((0,0),(1,1),(2,0))')`
`point(double precision, double precision)`	`point`	construct point	`point(23.4, -44.5)`
`point(box)`	`point`	center of box	`point(box '((-1,0),(1,0))')`
`point(circle)`	`point`	center of circle	`point(circle '((0,0),2.0)')`
`point(lseg)`	`point`	center of line segment	`point(lseg '((-1,0),(1,0))')`
`point(polygon)`	`point`	center of polygon	`point(polygon '((0,0),(1,1),(2,0))')`
`polygon(box)`	`polygon`	box to 4-point polygon	`polygon(box '((0,0),(1,1))')`
`polygon(circle)`	`polygon`	circle to 12-point polygon	`polygon(circle '((0,0),2.0)')`
`polygon(npts, circle)`	`polygon`	circle to *npts*-point polygon	`polygon(12, circle '((0,0),2.0)')`
`polygon(path)`	`polygon`	path to polygon	`polygon(path '((0,0),(1,1),(2,0))')`

cidr and inet Operators

Operator	Description	Example
<	is less than	`inet '192.168.1.5' < inet '192.168.1.6'`
<=	is less than or equal	`inet '192.168.1.5' <= inet '192.168.1.5'`
=	equals	`inet '192.168.1.5' = inet '192.168.1.5'`
>=	is greater or equal	`inet '192.168.1.5' >= inet '192.168.1.5'`
>	is greater than	`inet '192.168.1.5' > inet '192.168.1.4'`
<>	is not equal	`inet '192.168.1.5' <> inet '192.168.1.4'`
<<	is contained by	`inet '192.168.1.5' << inet '192.168.1/24'`
<<=	is contained by or equals	`inet '192.168.1/24' <<= inet '192.168.1/24'`
>>	contains	`inet '192.168.1/24' >> inet '192.168.1.5'`
>>=	contains or equals	`inet '192.168.1/24' >>= inet '192.168.1/24'`
&&	contains or is contained by	`inet '192.168.1/24' && inet '192.168.1.80/28'`
~	bitwise NOT	`~ inet '192.168.1.6'`
&	bitwise AND	`inet '192.168.1.6' & inet '0.0.0.255'`
|	bitwise OR	`inet '192.168.1.6' | inet '0.0.0.255'`
+	addition	`inet '192.168.1.6' + 25`
-	subtraction	`inet '192.168.1.43' - 36`
-	subtraction	`inet '192.168.1.43' - inet '192.168.1.19'`

cidr and inet Functions

Function	Return Type	Description	Example	Result
abbrev(inet)	text	abbreviated display format as text	abbrev(inet '10.1.0.0/16')	10.1.0.0/16
abbrev(cidr)	text	abbreviated display format as text	abbrev(cidr '10.1.0.0/16')	10.1/16
broadcast(inet)	inet	broadcast address for network	broadcast('192.168.1.5/24')	192.168.1.255/24
family(inet)	int	extract family of address; 4 for IPv4, 6 for IPv6	family('::1')	6
host(inet)	text	extract IP address as text	host('192.168.1.5/24')	192.168.1.5
hostmask(inet)	inet	construct host mask for network	hostmask('192.168.23.20/30')	0.0.0.3
masklen(inet)	int	extract netmask length	masklen('192.168.1.5/24')	24
netmask(inet)	inet	construct netmask for network	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	extract network part of address	network('192.168.1.5/24')	192.168.1.0/24
set_masklen(inet, int)	inet	set netmask length for inet value	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	set netmask length for cidr value	set_masklen('192.168.1.0/24'::cidr, 16)	192.168.0.0/16
text(inet)	text	extract IP address and netmask length as text	text(inet '192.168.1.5')	192.168.1.5/32

Table 9-36. macaddr Functions

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

Text Search Operators

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th><th>Result</th></tr>
<tr><td>@@</td><td>tsvector matches tsquery ?</td><td>to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')</td><td>t</td></tr>
<tr><td>@@@</td><td>deprecated synonym for @@</td><td>to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')</td><td>t</td></tr>
<tr><td>||</td><td>concatenate tsvectors</td><td>'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector</td><td>'a':1 'b':2,5 'c':3 'd':4</td></tr>
<tr><td>&&</td><td>AND tsquerys together</td><td>'fat | rat'::tsquery && 'cat'::tsquery</td><td>('fat' | 'rat') & 'cat'</td></tr>
<tr><td>||</td><td>OR tsquerys together</td><td>'fat | rat'::tsquery || 'cat'::tsquery</td><td>('fat' | 'rat') | 'cat'</td></tr>
<tr><td>!!</td><td>negate a tsquery</td><td>!! 'cat'::tsquery</td><td>!'cat'</td></tr>
<tr><td>@></td><td>tsquery contains another ?</td><td>'cat'::tsquery @> 'cat & rat'::tsquery</td><td>f</td></tr>
<tr><td><@</td><td>tsquery is contained in ?</td><td>'cat'::tsquery <@ 'cat & rat'::tsquery</td><td>t</td></tr>
</table>

Text Search Functions

Function	Return Type	Description	Example	Result
get_current_ts_config()	regconfig	get default text search configuration	get_current_ts_config()	english
length(tsvector)	integer	number of lexemes in tsvector	length('fat:2,4 cat:3 rat:5A'::tsvector)	3
numnode(tsquery)	integer	number of lexemes plus operators in tsquery	numnode('(fat & rat) | cat'::tsquery)	5
plainto_tsquery([*config* regconfig ,] *query* text)	tsquery	produce tsquery ignoring punctuation	plainto_tsquery('english', 'The Fat Rats')	'fat' & 'rat'
querytree(*query* tsquery)	text	get indexable part of a tsquery	querytree('foo & ! bar'::tsquery)	'foo'
setweight(tsvector, "char")	tsvector	assign weight to each element of tsvector	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
strip(tsvector)	tsvector	remove positions and weights from tsvector	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([*config* regconfig ,] *query* text)	tsquery	normalize words and convert to tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
to_tsvector([*config* regconfig ,] *document* text)	tsvector	reduce document text to tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text])	text	display a query match	ts_headline('x y z', 'z'::tsquery)	x y z
ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])	float4	rank document for query	ts_rank(textsearch, query)	0.818
ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])	float4	rank document for query using cover density	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_rewrite(*query* tsquery, *target* tsquery, *substitute* tsquery)	tsquery	replace target with substitute within query	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')
ts_rewrite(*query* tsquery, *select* text)	tsquery	replace using targets and substitutes from a SELECT command	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
tsvector_update_trigger()	trigger	trigger function for automatic tsvector column update	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)	
tsvector_update_trigger_column()	trigger	trigger function for automatic tsvector column update	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)	

Text Search Debugging Functions

Function	Return Type	Description	Example	Result
ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *token* text, OUT *dictionaries* regdictionary[], OUT *dictionary* regdictionary, OUT *lexemes* text[])	setof record	test a configuration	ts_debug('english', 'The Brightest supernovaes')	(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...
ts_lexize(*dict* regdictionary, *token* text)	text[]	test a dictionary	ts_lexize('english_stem', 'stars')	{star}
ts_parse(*parser_name* text, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	test a parser	ts_parse('default', 'foo - bar')	(1,foo) ...
ts_parse(*parser_oid* oid, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	test a parser	ts_parse(3722, 'foo - bar')	(1,foo) ...
ts_token_type(*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT *description* text)	setof record	get token types defined by parser	ts_token_type('default')	(1,asciiword,"Word, all ASCII") ...
ts_token_type(*parser_oid* oid, OUT *tokid* integer, OUT *alias* text, OUT *description* text)	setof record	get token types defined by parser	ts_token_type(3722)	(1,asciiword,"Word, all ASCII") ...
ts_stat(*sqlquery* text, [*weights* text,] OUT *word* text, OUT *ndoc* integer, OUT *nentry* integer)	setof record	get statistics of a tsvector column	ts_stat('SELECT vector from apod')	(foo,10,15) ...

json and jsonb Operators

Operator	Right Operand Type	Description	Example	Example Result
->	int	Get JSON array element (indexed from zero)	'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2	{"c":"baz"}
->	text	Get JSON object field by key	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	Get JSON array element as text	'[1,2,3]'::json->>2	3
->>	text	Get JSON object field as text	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	Get JSON object at specified path	'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	Get JSON object at specified path as text	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'	3

Table 9-41. Additional jsonb Operators

Operator	Right Operand Type	Description	Example
=	jsonb	Are the two JSON values equal?	'[1,2,3]'::jsonb = '[1,2,3]'::jsonb
@>	jsonb	Does the left JSON value contain within it the right value?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	Is the left JSON value contained within the right value?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	Does the key/element **string** exist within the JSON value?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	Do any of these key/element **strings** exist?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	Do all of these key/element **strings** exist?	'["a", "b"]'::jsonb ?& array['a', 'b']

JSON Creation Functions

Function	Description	Example	Example Result
`to_json(anyelement)`	Returns the value as JSON. Arrays and composites are converted (recursively) to arrays and objects; otherwise, if there is a cast from the type to `json`, the cast function will be used to perform the conversion; otherwise, a JSON scalar value is produced. For any scalar type other than a number, a boolean, or a null value, the text representation will be used, properly quoted and escaped so that it is a valid JSON string.	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`
`array_to_json(anyarray [, pretty_bool])`	Returns the array as a JSON array. A PostgreSQL multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if `pretty_bool` is true.	`array_to_json('{{1,5},{99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	Returns the row as a JSON object. Line feeds will be added between level-1 elements if `pretty_bool` is true.	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`
`json_build_array(VARIADIC "any")`	Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list.	`json_build_array(1,2,'3',4,5)`	`[1, 2, "3", 4, 5]`
`json_build_object(VARIADIC "any")`	Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating names and values.	`json_build_object('foo',1,'bar',2)`	`{"foo": 1, "bar": 2}`

`json_object(text[])`	Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating name/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a name/value pair.	`json_object('{a, 1, b, "def", c, 3.5}')`
		
		`json_object('{{a, 1},{b, "def"},{c, 3.5}}')`
			`{"a": "1", "b": "def", "c": "3.5"}`

`json_object(keys text[], values text[])`	This form of `json_object` takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form.	`json_object('{a, b}', '{1,2}')`	`{"a": "1", "b": "2"}`

Sequence Manipulation Functions

Function	Return Type	Description
`currval(regclass)`	`bigint`	Return value most recently obtained with `nextval` for specified sequence
`lastval()`	`bigint`	Return value most recently obtained with `nextval` for any sequence
`nextval(regclass)`	`bigint`	Advance sequence and return new value
`setval(regclass, bigint)`	`bigint`	Set sequence's current value
`setval(regclass, bigint, boolean)`	`bigint`	Set sequence's current value and `is_called` flag

- nextval: Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute nextval concurrently, each will safely receive a distinct sequence value.
- currval: Return the value most recently obtained by nextval for this sequence in the current session. (An error is reported if nextval has never been called for this sequence in this session.) Because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed nextval since the current session did.
- lastval: Return the value most recently returned by nextval in the current session. This function is identical to currval, except that instead of taking the sequence name as an argument it fetches the value of the last sequence used by nextval in the current session. It is an error to call lastval if nextval has not yet been called in the current session.
- setval:Reset the sequence object's counter value. The two-parameter form sets the sequence's last_value field to the specified value and sets its is_called field to true, meaning that the next nextval will advance the sequence before returning a value. The value reported by currval is also set to the specified value. In the three-parameter form, is_called can be set to either true or false. true has the same effect as the two-parameter form. If it is set to false, the next nextval will return exactly the specified value, and sequence advancement commences with the following nextval.

COALESCE & NULLIF

- The COALESCE function returns the first of its arguments that is not null.
- Null is returned only if all arguments are null.

Example:

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

* This returns description if it is not null, otherwise short_description if it is not null, otherwise (none).

- The NULLIF function returns a null value if value1 equals value2; otherwise it returns value1.

Example:

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

GREATEST and LEAST

- The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions.
- The expressions must all be convertible to a common data type, which will be the type of the result

Syntax:

- GREATEST(value [, ...])
- LEAST(value [, ...])

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension. Some other databases make them return NULL if any argument is NULL, rather than only when all are NULL.

Array Functions

Function	Return Type	Description	Example	Result
`array_append(anyarray, anyelement)`	`anyarray`	append an element to the end of an array	`array_append(ARRAY[1,2], 3)`	`{1,2,3}`
`array_cat(anyarray, anyarray)`	`anyarray`	concatenate two arrays	`array_cat(ARRAY[1,2,3], ARRAY[4,5])`	`{1,2,3,4,5}`
`array_ndims(anyarray)`	`int`	returns the number of dimensions of the array	`array_ndims(ARRAY[[1,2,3], [4,5,6]])`	`2`
`array_dims(anyarray)`	`text`	returns a text representation of array's dimensions	`array_dims(ARRAY[[1,2,3], [4,5,6]])`	`[1:2][1:3]`
`array_fill(anyelement, int[], [, int[]])`	`anyarray`	returns an array initialized with supplied value and dimensions, optionally with lower bounds other than 1	`array_fill(7, ARRAY[3], ARRAY[2])`	`[2:4]={7,7,7}`
`array_length(anyarray, int)`	`int`	returns the length of the requested array dimension	`array_length(array[1,2,3], 1)`	`3`
`array_lower(anyarray, int)`	`int`	returns lower bound of the requested array dimension	`array_lower('[0:2]={1,2,3}'::int[], 1)`	`0`
`array_prepend(anyelement, anyarray)`	`anyarray`	append an element to the beginning of an array	`array_prepend(1, ARRAY[2,3])`	`{1,2,3}`
`array_remove(anyarray, anyelement)`	`anyarray`	remove all elements equal to the given value from the array (array must be one-dimensional)	`array_remove(ARRAY[1,2,3,2], 2)`	`{1,3}`
`array_replace(anyarray, anyelement, anyelement)`	`anyarray`	replace each array element equal to the given value with a new value	`array_replace(ARRAY[1,2,5,4], 5, 3)`	`{1,2,3,4}`
`array_to_string(anyarray, text [, text])`	`text`	concatenates array elements using supplied delimiter and optional null string	`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')`	`1,2,3,*,5`
`array_upper(anyarray, int)`	`int`	returns upper bound of the requested array dimension	`array_upper(ARRAY[1,8,3,7], 1)`	`4`
`cardinality(anyarray)`	`int`	returns the total number of elements in the array, or 0 if the array is empty	`cardinality(ARRAY[[1,2],[3,4]])`	`4`
`string_to_array(text, text [, text])`	`text[]`	splits string into array elements using supplied delimiter and optional null string	`string_to_array('xx~^~yy~^~zz', '~^~', 'yy')`	`{xx,NULL,zz}`
`unnest(anyarray)`	`setof anyelement`	expand an array to a set of rows	`unnest(ARRAY[1,2])`	1 2 (2 rows)
`unnest(anyarray, anyarray [, ...])`	`setof anyelement, anyelement [, ...]`	expand multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause; see Section 7.2.1.4	`unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])`	1 foo 2 bar NULL baz (3 rows)

Range Functions

Function	Return Type	Description	Example	Result
`lower(anyrange)`	range's element type	lower bound of range	`lower(numrange(1.1,2.2))`	`1.1`
`upper(anyrange)`	range's element type	upper bound of range	`upper(numrange(1.1,2.2))`	`2.2`
`isempty(anyrange)`	`boolean`	is the range empty?	`isempty(numrange(1.1,2.2))`	`false`
`lower_inc(anyrange)`	`boolean`	is the lower bound inclusive?	`lower_inc(numrange(1.1,2.2))`	`true`
`upper_inc(anyrange)`	`boolean`	is the upper bound inclusive?	`upper_inc(numrange(1.1,2.2))`	`false`
`lower_inf(anyrange)`	`boolean`	is the lower bound infinite?	`lower_inf('(,)'::daterange)`	`true`
`upper_inf(anyrange)`	`boolean`	is the upper bound infinite?	`upper_inf('(,)'::daterange)`	`true`

Aggregate Functions

Function	Argument Type(s)	Return Type	Description
array_agg(*expression*)	any	array of the argument type	input values, including nulls, concatenated into an array
avg(*expression*)	smallint, int, bigint, real, double precision, numeric, or interval	numeric for any integer-type argument, double precision for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all input values
bit_and(*expression*)	smallint, int, bigint, or bit	same as argument data type	the bitwise AND of all non-null input values, or null if none
bit_or(*expression*)	smallint, int, bigint, or bit	same as argument data type	the bitwise OR of all non-null input values, or null if none
bool_and(*expression*)	bool	bool	true if all input values are true, otherwise false
bool_or(*expression*)	bool	bool	true if at least one input value is true, otherwise false
count(*)		bigint	number of input rows
count(*expression*)	any	bigint	number of input rows for which the value of *expression* is not null
every(*expression*)	bool	bool	equivalent to bool_and
json_agg(*record*)	record	json	aggregates records as a JSON array of objects
json_object_agg(*name*, *value*)	("any", "any")	json	aggregates name/value pairs as a JSON object
max(*expression*)	any array, numeric, string, or date/time type	same as argument type	maximum value of *expression* across all input values
min(*expression*)	any array, numeric, string, or date/time type	same as argument type	minimum value of *expression* across all input values
string_agg(*expression*, *delimiter*)	(text, text) or (bytea, bytea)	same as argument types	input values concatenated into a string, separated by delimiter
sum(*expression*)	smallint, int, bigint, real, double precision, numeric, or interval	bigint for smallint or int arguments, numeric for bigint arguments, double precision for floating-point arguments, otherwise the same as the argument data type	sum of *expression* across all input values
xmlagg(*expression*)	xml	xml	concatenation of XML values (see also Section 9.14.1.7)

Aggregate Functions for Statistics

Function	Argument Type	Return Type	Description
`corr(Y, X)`	`double precision`	`double precision`	correlation coefficient
`covar_pop(Y, X)`	`double precision`	`double precision`	population covariance
`covar_samp(Y, X)`	`double precision`	`double precision`	sample covariance
`regr_avgx(Y, X)`	`double precision`	`double precision`	average of the independent variable (`sum(X)/N`)
`regr_avgy(Y, X)`	`double precision`	`double precision`	average of the dependent variable (`sum(Y)/N`)
`regr_count(Y, X)`	`double precision`	`bigint`	number of input rows in which both expressions are nonnull
`regr_intercept(Y, X)`	`double precision`	`double precision`	y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs
`regr_r2(Y, X)`	`double precision`	`double precision`	square of the correlation coefficient
`regr_slope(Y, X)`	`double precision`	`double precision`	slope of the least-squares-fit linear equation determined by the (X, Y) pairs
`regr_sxx(Y, X)`	`double precision`	`double precision`	`sum(X^2) - sum(X)^2/N` ("sum of squares" of the independent variable)
`regr_sxy(Y, X)`	`double precision`	`double precision`	`sum(X*Y) - sum(X) * sum(Y)/N` ("sum of products" of independent times dependent variable)
`regr_syy(Y, X)`	`double precision`	`double precision`	`sum(Y^2) - sum(Y)^2/N` ("sum of squares" of the dependent variable)
`stddev(expression)`	`smallint`, `int`, `bigint`, `real`, `double precision`, or `numeric`	`double precision` for floating-point arguments, otherwise `numeric`	historical alias for `stddev_samp`
`stddev_pop(expression)`	`smallint`, `int`, `bigint`, `real`, `double precision`, or `numeric`	`double precision` for floating-point arguments, otherwise `numeric`	population standard deviation of the input values
`stddev_samp(expression)`	`smallint`, `int`, `bigint`, `real`, `double precision`, or `numeric`	`double precision` for floating-point arguments, otherwise `numeric`	sample standard deviation of the input values
`variance(expression)`	`smallint`, `int`, `bigint`, `real`, `double precision`, or `numeric`	`double precision` for floating-point arguments, otherwise `numeric`	historical alias for `var_samp`
`var_pop(expression)`	`smallint`, `int`, `bigint`, `real`, `double precision`, or `numeric`	`double precision` for floating-point arguments, otherwise `numeric`	population variance of the input values (square of the population standard deviation)
`var_samp(expression)`	`smallint`, `int`, `bigint`, `real`, `double precision`, or `numeric`	`double precision` for floating-point arguments, otherwise `numeric`	sample variance of the input values (square of the sample standard deviation)

Ordered-Set Aggregate Functions

Function	Direct Argument Type(s)	Aggregated Argument Type(s)	Return Type	Description
`mode() WITHIN GROUP (ORDER BY sort_expression)`		any sortable type	same as sort expression	returns the most frequent input value (arbitrarily choosing the first one if there are multiple equally-frequent results)
`percentile_cont(fraction) WITHIN GROUP (ORDER BY sort_expression)`	`double precision`	`double precision` or `interval`	same as sort expression	continuous percentile: returns a value corresponding to the specified fraction in the ordering, interpolating between adjacent input items if needed
`percentile_cont(fractions) WITHIN GROUP (ORDER BY sort_expression)`	`double precision[]`	`double precision` or `interval`	array of sort expression's type	multiple continuous percentile: returns an array of results matching the shape of the *fractions* parameter, with each non-null element replaced by the value corresponding to that percentile
`percentile_disc(fraction) WITHIN GROUP (ORDER BY sort_expression)`	`double precision`	any sortable type	same as sort expression	discrete percentile: returns the first input value whose position in the ordering equals or exceeds the specified fraction
`percentile_disc(fractions) WITHIN GROUP (ORDER BY sort_expression)`	`double precision[]`	any sortable type	array of sort expression's type	multiple discrete percentile: returns an array of results matching the shape of the *fractions* parameter, with each non-null element replaced by the input value corresponding to that percentile

Hypothetical-Set Aggregate Functions

Function	Direct Argument Type(s)	Aggregated Argument Type(s)	Return Type	Description
`rank(`*args*`) WITHIN GROUP (ORDER BY `*sorted_args*`)`	`VARIADIC "any"`	`VARIADIC "any"`	`bigint`	rank of the hypothetical row, with gaps for duplicate rows
`dense_rank(`*args*`) WITHIN GROUP (ORDER BY `*sorted_args*`)`	`VARIADIC "any"`	`VARIADIC "any"`	`bigint`	rank of the hypothetical row, without gaps
`percent_rank(`*args*`) WITHIN GROUP (ORDER BY `*sorted_args*`)`	`VARIADIC "any"`	`VARIADIC "any"`	`double precision`	relative rank of the hypothetical row, ranging from 0 to 1
`cume_dist(`*args*`) WITHIN GROUP (ORDER BY `*sorted_args*`)`	`VARIADIC "any"`	`VARIADIC "any"`	`double precision`	relative rank of the hypothetical row, ranging from 1/*N* to 1

Window Functions

Function	Return Type	Description
`row_number()`	`bigint`	number of the current row within its partition, counting from 1
`rank()`	`bigint`	rank of the current row with gaps; same as `row_number` of its first peer
`dense_rank()`	`bigint`	rank of the current row without gaps; this function counts peer groups
`percent_rank()`	`double precision`	relative rank of the current row: (`rank` - 1) / (total rows - 1)
`cume_dist()`	`double precision`	relative rank of the current row: (number of rows preceding or peer with current row) / (total rows)
`ntile(`*`num_buckets`*` integer)`	`integer`	integer ranging from 1 to the argument value, dividing the partition as equally as possible
`lag(`*`value`*` any [, `*`offset`*` integer [, `*`default`*` any ]])`	`same type as value`	returns *`value`* evaluated at the row that is *`offset`* rows before the current row within the partition; if there is no such row, instead return *`default`*. Both *`offset`* and *`default`* are evaluated with respect to the current row. If omitted, *`offset`* defaults to 1 and *`default`* to null
`lead(`*`value`*` any [, `*`offset`*` integer [, `*`default`*` any ]])`	`same type as value`	returns *`value`* evaluated at the row that is *`offset`* rows after the current row within the partition; if there is no such row, instead return *`default`*. Both *`offset`* and *`default`* are evaluated with respect to the current row. If omitted, *`offset`* defaults to 1 and *`default`* to null
`first_value(`*`value`*` any)`	`same type as value`	returns *`value`* evaluated at the row that is the first row of the window frame
`last_value(`*`value`*` any)`	`same type as value`	returns *`value`* evaluated at the row that is the last row of the window frame
`nth_value(`*`value`*` any, `*`nth`*` integer)`	`same type as value`	returns *`value`* evaluated at the row that is the *`nth`* row of the window frame (counting from 1); null if no such row

Subquery Expressions

- The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is "true"; if the subquery returns no rows, the result of EXISTS is "false".

Example: SELECT col1 FROM tab1 WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);

Set Returning Functions

Table 9-54. Series Generating Functions

Function	Argument Type	Return Type	Description
`generate_series(start, stop)`	`int` or `bigint`	`setof int` or `setof bigint` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of one
`generate_series(start, stop, step)`	`int` or `bigint`	`setof int` or `setof bigint` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of `step`
`generate_series(start, stop, step interval)`	`timestamp` or `timestamp with time zone`	`setof timestamp` or `setof timestamp with time zone` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of `step`

Table 9-55. Subscript Generating Functions

Function	Return Type	Description
`generate_subscripts(array anyarray, dim int)`	`setof int`	Generate a series comprising the given array's subscripts.
`generate_subscripts(array anyarray, dim int, reverse boolean)`	`setof int`	Generate a series comprising the given array's subscripts. When `reverse` is true, the series is returned in reverse order.

System Administration Functions

- Configuration Settings Functions

Name	Return Type	Description
current_setting(setting_name)	text	get current value of setting
set_config(setting_name, new_value, is_local)	text	set parameter and return new value

- Ex:

  -SELECT current_setting('datestyle');

  - SELECT set_config('log_statement_stats', 'off', false);

Server Signaling Functions

Name	Return Type	Description
pg_cancel_backend(pid int)	boolean	Cancel a backend's current query. You can execute this against another backend that has exactly the same role as the user calling the function. In all other cases, you must be a superuser.
pg_reload_conf()	boolean	Cause server processes to reload their configuration files
pg_rotate_logfile()	boolean	Rotate server's log file
pg_terminate_backend(pid int)	boolean	Terminate a backend. You can execute this against another backend that has exactly the same role as the user calling the function. In all other cases, you must be a superuser.

Backup Control Functions

Name	Return Type	Description
pg_create_restore_point(name text)	pg_lsn	Create a named point for performing restore (restricted to superusers)
pg_current_xlog_insert_location()	pg_lsn	Get current transaction log insert location
pg_current_xlog_location()	pg_lsn	Get current transaction log write location
pg_start_backup(label text [, fast boolean])	pg_lsn	Prepare for performing on-line backup (restricted to superusers or replication roles)
pg_stop_backup()	pg_lsn	Finish performing on-line backup (restricted to superusers or replication roles)
pg_is_in_backup()	bool	True if an on-line exclusive backup is still in progress.
pg_backup_start_time()	timestamp with time zone	Get start time of an on-line exclusive backup in progress.
pg_switch_xlog()	pg_lsn	Force switch to a new transaction log file (restricted to superusers)
pg_xlogfile_name(location pg_lsn)	text	Convert transaction log location string to file name
pg_xlogfile_name_offset(location pg_lsn)	text, integer	Convert transaction log location string to file name and decimal byte offset within file
pg_xlog_location_diff(location pg_lsn, location pg_lsn)	numeric	Calculate the difference between two transaction log locations

Backup Control Functions

Name	Return Type	Description
pg_create_restore_point(name text)	pg_lsn	Create a named point for performing restore (restricted to superusers)
pg_current_xlog_insert_location()	pg_lsn	Get current transaction log insert location
pg_current_xlog_location()	pg_lsn	Get current transaction log write location
pg_start_backup(label text [, fast boolean])	pg_lsn	Prepare for performing on-line backup (restricted to superusers or replication roles)
pg_stop_backup()	pg_lsn	Finish performing on-line backup (restricted to superusers or replication roles)
pg_is_in_backup()	bool	True if an on-line exclusive backup is still in progress.
pg_backup_start_time()	timestamp with time zone	Get start time of an on-line exclusive backup in progress.
pg_switch_xlog()	pg_lsn	Force switch to a new transaction log file (restricted to superusers)
pg_xlogfile_name(location pg_lsn)	text	Convert transaction log location string to file name
pg_xlogfile_name_offset(location pg_lsn)	text, integer	Convert transaction log location string to file name and decimal byte offset within file
pg_xlog_location_diff(location pg_lsn, location pg_lsn)	numeric	Calculate the difference between two transaction log locations

Recovery Control Functions

Name	Return Type	Description
pg_is_in_recovery()	bool	True if recovery is still in progress.
pg_last_xlog_receive_location()	pg_lsn	Get last transaction log location received and synced to disk by streaming replication. While streaming replication is in progress this will increase monotonically. If recovery has completed this will remain static at the value of the last WAL record received and synced to disk during recovery. If streaming replication is disabled, or if it has not yet started, the function returns NULL.
pg_last_xlog_replay_location()	pg_lsn	Get last transaction log location replayed during recovery. If recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last WAL record applied during that recovery. When the server has been started normally without recovery the function returns NULL.
pg_last_xact_replay_timestamp()	timestamp with time zone	Get time stamp of last transaction replayed during recovery. This is the time at which the commit or abort WAL record for that transaction was generated on the primary. If no transactions have been replayed during recovery, this function returns NULL. Otherwise, if recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last transaction applied during that recovery. When the server has been started normally without recovery the function returns NULL.

Name	Return Type	Description
pg_is_xlog_replay_paused()	bool	True if recovery is paused.
pg_xlog_replay_pause()	void	Pauses recovery immediately.
pg_xlog_replay_resume()	void	Restarts recovery if it was paused.

Snapshot Synchronization Functions

- A snapshot determines which data is visible to the transaction that is using the snapshot.
- Synchronized snapshots are necessary when two or more sessions need to see identical content in the database.
- The function pg_export_snapshot saves the current snapshot and returns a text string identifying the snapshot.

Name	Return Type	Description
`pg_export_snapshot()`	`text`	Save the current snapshot and return its identifier

Advisory Lock Functions

Name	Return Type	Description
pg_advisory_lock(key bigint)	void	Obtain exclusive session level advisory lock
pg_advisory_lock(key1 int, key2 int)	void	Obtain exclusive session level advisory lock
pg_advisory_lock_shared(key bigint)	void	Obtain shared session level advisory lock
pg_advisory_lock_shared(key1 int, key2 int)	void	Obtain shared session level advisory lock
pg_advisory_unlock(key bigint)	boolean	Release an exclusive session level advisory lock
pg_advisory_unlock(key1 int, key2 int)	boolean	Release an exclusive session level advisory lock
pg_advisory_unlock_all()	void	Release all session level advisory locks held by the current session
pg_advisory_unlock_shared(key bigint)	boolean	Release a shared session level advisory lock
pg_advisory_unlock_shared(key1 int, key2 int)	boolean	Release a shared session level advisory lock
pg_advisory_xact_lock(key bigint)	void	Obtain exclusive transaction level advisory lock
pg_advisory_xact_lock(key1 int, key2 int)	void	Obtain exclusive transaction level advisory lock
pg_advisory_xact_lock_shared(key bigint)	void	Obtain shared transaction level advisory lock
pg_advisory_xact_lock_shared(key1 int, key2 int)	void	Obtain shared transaction level advisory lock
pg_try_advisory_lock(key bigint)	boolean	Obtain exclusive session level advisory lock if available
pg_try_advisory_lock(key1 int, key2 int)	boolean	Obtain exclusive session level advisory lock if available
pg_try_advisory_lock_shared(key bigint)	boolean	Obtain shared session level advisory lock if available
pg_try_advisory_lock_shared(key1 int, key2 int)	boolean	Obtain shared session level advisory lock if available
pg_try_advisory_xact_lock(key bigint)	boolean	Obtain exclusive transaction level advisory lock if available
pg_try_advisory_xact_lock(key1 int, key2 int)	boolean	Obtain exclusive transaction level advisory lock if available
pg_try_advisory_xact_lock_shared(key bigint)	boolean	Obtain shared transaction level advisory lock if available
pg_try_advisory_xact_lock_shared(key1 int, key2 int)	boolean	Obtain shared transaction level advisory lock if available

Index

- Indexes are a common way to enhance database performance.
- An index allows the database server to find and retrieve specific rows much faster than it could do without an index.
- To create index we use as below:

Syntax:

```
CREATE INDEX index_name ON table_name (filename);
```

Ex:

```
CREATE INDEX table_id_index ON table(id);
```

- But you might have to run the ANALYZE command regularly to update statistics to allow the query planner to make educated decisions.

Index Types

- Index types: B-tree, Hash, GiST, SP-GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries.
- By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations.
- B-trees can handle equality and range queries on data that can be sorted into some ordering.
- Hash indexes can only handle simple equality comparisons. Ex: **CREATE INDEX name ON table USING hash (column);**
- GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented.
- GIN indexes are inverted indexes which can handle values that contain more than one key, arrays for example. Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies and the particular operators with which a GIN index can be used vary depending on the indexing strategy.

Multicolumn Indexes

- An index can be defined on more than one column of a table. Ex: **CREATE INDEX test2_mm_idx ON test2 (major, minor);**
- A multicolumn B-tree index can be used with query conditions that involve any subset of the index's columns, but the index is most efficient when there are constraints on the leading (leftmost) columns.
- A multicolumn GiST index can be used with query conditions that involve any subset of the index's columns.
- A multicolumn GIN index can be used with query conditions that involve any subset of the index's columns.

Full Text Search

- To provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query.
- Full text indexing allows documents to be preprocessed and an index saved for later rapid searching. Preprocessing includes:

- Parsing documents into tokens. It is useful to identify various classes of tokens, e.g., numbers, words, complex words, email addresses, so that they can be processed differently.

- Converting tokens into lexemes. A lexeme is a string, just like a token, but it has been normalized so that different forms of the same word are made alike.

- Storing preprocessed documents optimized for searching. For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes it is often desirable to store positional information to use for proximity ranking, so that a document that contains a more "dense" region of query words is assigned a higher rank than one with scattered query words.

Concurrency Control

- Data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC).
- The main advantage of using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading.

Transaction Isolation

- The phenomena which are prohibited at various levels are:

  - dirty read

    A transaction reads data written by a concurrent uncommitted transaction.

  - nonrepeatable read

    A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

  - phantom read

    A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

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

Explicit Locking

- ACCESS EXCLUSIVE lock cannot be held by more than one transaction at a time (self-conflicting).
- ACCESS SHARE lock can be held by multiple transactions (not self-conflicting).

Table-level Locks:

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Row-level Locks

- Can be exclusive or shared locks.
- They block only writers to the same row.
- Locking a row might cause a disk write, e.g., SELECT FOR UPDATE modifies selected rows to mark them locked, and so will result in disk writes.

Deadlocks

- Deadlock occurs when transaction 2 has already exclusive-locked table B and now wants an exclusive lock on table A.
- Deadlocks can also occur as the result of row-level locks (and thus, they can occur even if explicit locking is not used).
- The best defense against deadlocks is generally to avoid them by being certain that all applications using a database acquire locks on multiple objects in a consistent order.
- So long as no deadlock situation is detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).

Advisory Locks

- Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model.
- There are two ways to acquire an advisory lock in PostgreSQL:

1. At session level or at transaction level. Once acquired at session level, an advisory lock is held until explicitly released or the session ends.

2. Transaction-level lock requests, on the other hand, behave more like regular lock requests: they are automatically released at the end of the transaction, and there is no explicit unlock operation.

Data Consistency Checks at the Application Level

- Enforcing Consistency With Serializable Transactions:

  When using this technique, it will avoid creating an unnecessary burden for application programmers if the application software goes through a framework which automatically retries transactions which are rolled back with a serialization failure. It may be a good idea to set default_transaction_isolation to serializable. It would also be wise to take some action to ensure that no other transaction isolation level is used, either inadvertently or to subvert integrity checks, through checks of the transaction isolation level in triggers.

- Enforcing Consistency With Explicit Blocking Locks:

  Non-serializable writes are possible, to ensure the current validity of a row and protect it against concurrent updates one must use SELECT FOR UPDATE, SELECT FOR SHARE, or an appropriate LOCK TABLE statement. (SELECT FOR UPDATE and SELECT FOR SHARE lock just the returned rows against concurrent updates, while LOCK TABLE locks the whole table.).

Locking and Indexes

- B-tree, GiST and SP-GiST indexes:

  Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. These index types provide the highest concurrency without deadlock conditions.

- Hash indexes:

  Share/exclusive hash-bucket-level locks are used for read/write access. Locks are released after the whole bucket is processed. Bucket-level locks provide better concurrency than index-level ones, but deadlock is possible since the locks are held longer than one index operation.

- GIN indexes:

  Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. But note that insertion of a GIN-indexed value usually produces several index key insertions per row, so GIN might do substantial work for a single value's insertion.

Performance Tips

- EXPLAIN
- EXPLAIN ANALYZE

Populating a Database

- Disable Autocommit
- Use COPY
- Remove Indexes
- Remove Foreign Key Constraints
- Increase maintenance_work_mem
- Increase checkpoint_segments
- Disable WAL Archival and Streaming Replication:

    To prevent incremental WAL logging while loading, disable archiving and streaming replication, by setting wal_level to minimal, archive_mode to off, and max_wal_senders to zero. But note that changing these settings requires a server restart.

- Run ANALYZE Afterwards
- Some Notes About pg_dump

Non-Durable Settings

- Durability is a database feature that guarantees the recording of committed transactions even if the server crashes or loses power.
- Place the database cluster's data directory in a memory-backed file system (i.e. RAM disk). This eliminates all database disk I/O, but limits data storage to the amount of available memory (and perhaps swap).
- Turn off fsync; there is no need to flush data to disk.
- Turn off synchronous_commit; there might be no need to force WAL writes to disk on every commit. This setting does risk transaction loss (though not data corruption) in case of a crash of the database.
- Turn off full_page_writes; there is no need to guard against partial page writes.
- Increase checkpoint_segments and checkpoint_timeout ; this reduces the frequency of checkpoints, but increases the storage requirements of /pg_xlog.
- Create unlogged tables to avoid WAL writes, though it makes the tables non-crash-safe

III. Server Administration

Installation from Source Code

- Short Version:

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```