

Postgresql – Install, Configure, SQL

GnuGroup – ILGLabs

www.gnugroup.org

info@gnugroup.org



PostgreSQL Overview

PostgreSQL is a powerful, open source object-relational database system ORDBMS.

It has around 20+ years of active development and a proven architecture that has earned it a strong reputation for **reliability**, **data integrity**, and **correctness**.



What is PostgreSQL?

PostgreSQL (pronounced as post-gress-Q-L) is an open source relational database management system (DBMS) developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

PostgreSQL, often simply Postgres, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and on standards-compliance.

As a database server, its primary function is to store data securely, supporting best practices, and to allow for retrieval at the request of other software applications.

It can handle workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users.

Technically

- PostgreSQL implements the majority of the SQL:2016 standard.
- ACID-compliant and transactional (including most DDL statements) avoiding locking issues using multiversion concurrency control (MVCC), provides immunity to dirty reads and full serializability
- Handles complex SQL queries using many indexing methods that are not available in other databases;
- Has updateable views and materialized views, triggers, foreign keys; supports functions and stored procedures, and other expandability.
- Has a large number of extensions written by third parties.

Source- Wikipedia

Key features of PostgreSQL

PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It supports text, images, sounds, and video, and includes programming interfaces for C / C++ , Java , Perl , Python , Ruby, Tcl and Open Database Connectivity (ODBC).

PostgreSQL supports a large part of the SQL standard and offers many modern features including the following:

- Complex SQL queries**

- SQL Sub-selects**

- Foreign keys**

- Trigger**

- Views**

- Transactions**

- Multiversion concurrency control (MVCC)**

- Streaming Replication (as of 9.0)**

- Hot Standby (as of 9.0)**

- PostgreSQL can be extended by the user in many ways, for example by adding new:
 - Data types
 - Functions
 - Operators
 - Aggregate functions
 - Index methods

Procedural Languages Support

PostgreSQL supports four standard procedural languages which allows the users to write their own code in any of the languages and it can be executed by PostgreSQL database server.

These procedural languages are - PL/pgSQL, PL/Tcl, PL/Perl and PL/Python.

Besides, other non-standard procedural languages like PL/PHP, PL/V8, PL/Ruby, PL/Java, etc., are also supported.

PostgreSQL Environment Setup

Installing PostgreSQL on Linux/Unix

Follow the following steps to install PostgreSQL on your Linux machine.

Make sure you are logged in as root before you proceed for the installation.

Pick the version number of PostgreSQL you want and, as exactly as possible, the platform you want from a EnterpriseDB downloaded postgresql-9.x.x-x-linux-x64.run for my 64 bit CentOS-6 machine.

Now, let's execute it as follows:

```
[root@host]# chmod +x postgresql-9.2.4-1-linux-x64.run
```

```
[root@host]# ./postgresql-9.2.4-1-linux-x64.run
```

Welcome to the PostgreSQL Setup Wizard.

Please specify the directory where PostgreSQL will be installed.

Installation Directory [/opt/PostgreSQL/9.2]:

Once you launch the installer, it asks you few basic questions like location of the installation, password of the user, who will use database, port number, etc. So keep all of them at their default values except password, which you can provide password as per your choice. It will install PostgreSQL at your Linux machine and will display the following message:

Please wait while Setup installs PostgreSQL on your computer.

Installing

0% _____ 50% _____ 100%

#####

--

Setup has finished installing PostgreSQL on your computer.

Follow the following post-installation steps to create your database:

```
[root@host]# su - postgres
Password:
bash-4.1$ createdb testdb
bash-4.1$ psql testdb
psql (8.4.13, server 9.2.4)
test=#
```

You can start/restart postgres server in case it is not running using the following command:

```
[root@host]# service postgresql restart
Stopping postgresql service:    [ OK ]
Starting postgresql service:    [ OK ]
```

PostgreSQL - Syntax

Using psql, you can generate the complete list of commands by using the \help command. For the syntax of a specific command, use the following command:

```
postgres-# \help <command_name>
```

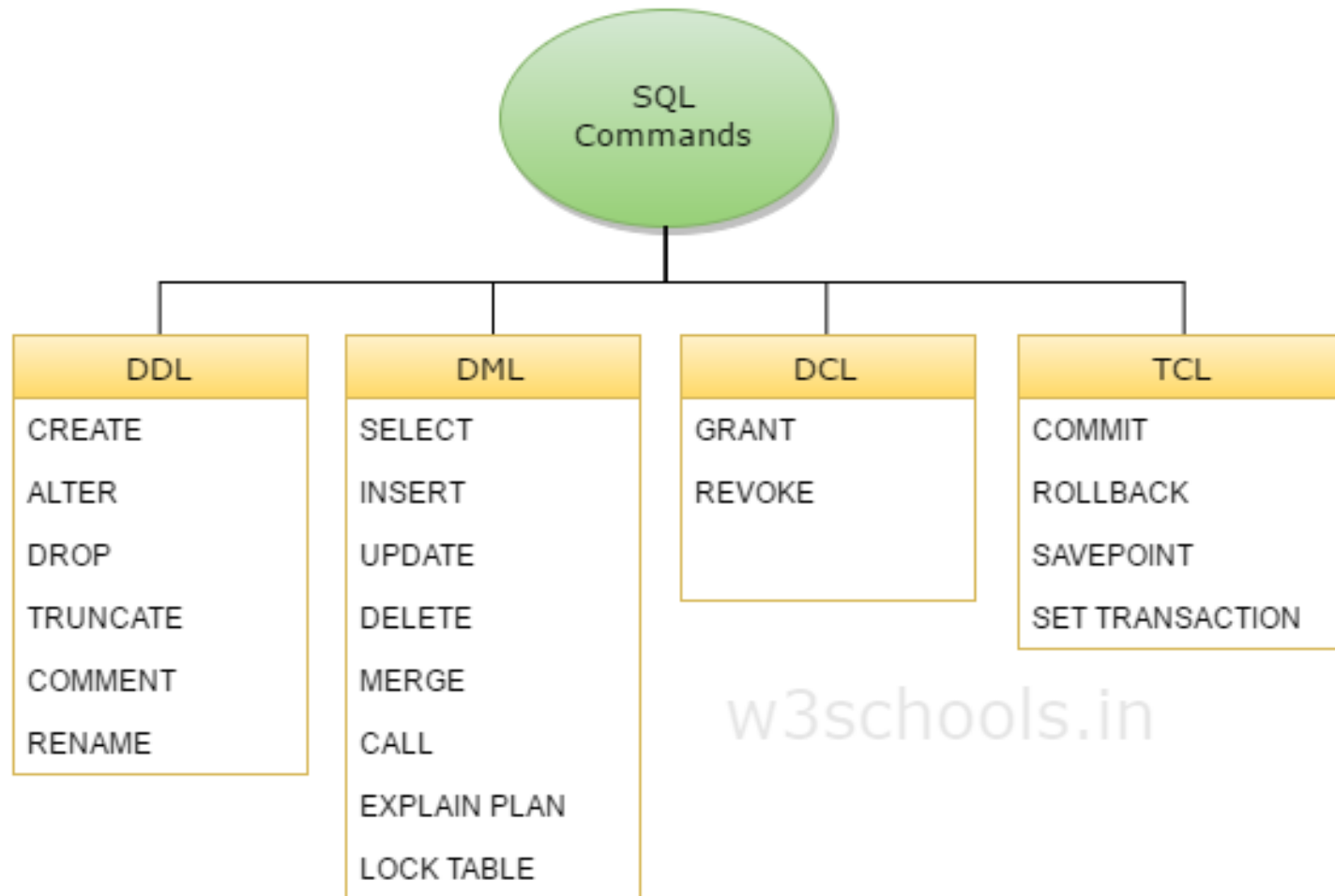
SQL Statement

An SQL statement is comprised of tokens where each token can represent either a **keyword**, **identifier**, **quoted identifier**, **constant**, or special character symbol.

The table below uses a simple SELECT statement to illustrate a basic, but complete, SQL statement and its components.

	SELECT	id, name	FROM	states
Token Type	Keyword	Identifiers	Keyword	Identifier
Description	Command	Id and name columns	Clause	Table name

DML, DDL, DCL, TCL



w3schools.in

DDL

DDL is short name of Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.

- CREATE – to create database and its objects like (table, index, views, store procedure, function and triggers)

ALTER – alters the structure of the existing database

- DROP – delete objects from the database
- TRUNCATE – remove all records from a table, including all spaces allocated for the records are removed
- COMMENT – add comments to the data dictionary
- RENAME – rename an object

DML

DML is short name of Data Manipulation Language which deals with data manipulation, and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE etc, and it is used to store, modify, retrieve, delete and update data in database.

- SELECT retrieve data from the a database
- INSERT– insert data into a table
- UPDATE – updates existing data within a table
- DELETE – Delete all records from a database table
- MERGE – UPSERT operation (insert or update)
- CALL – call a PL/SQL or Java subprogram
- EXPLAIN PLAN – interpretation of the data access path
- LOCK TABLE – concurrency Control

DCL

DCL is short name of Data Control Language which includes commands such as GRANT, and mostly concerned with rights, permissions and other controls of the database system.

- GRANT – allow users access privileges to database
- REVOKE – withdraw users access privileges given by using the GRANT command

TCL is short name of Transaction Control Language which deals with transaction within a database.

- COMMIT – commits a Transaction
- ROLLBACK – rollback a transaction in case of any error occurs
- SAVEPOINT – to rollback the transaction making points within groups
- SET TRANSACTION – specify characteristics for the transaction

PostgreSQL - Data Type

While creating table, for each column, you specify a data type, i.e., what kind of data you want to store in the table fields.

- This enables several benefits:
- **Consistency**: Operations against columns of same data type give consistent results, and are usually the fastest.
- **Validation**: Proper use of data types implies format validation of data and rejection of data outside the scope of data type.
- **Compactness**: As a column can store a single type of value, it is stored in a compact way.
- **Performance**: Proper use of data types gives the most efficient storage of data. The values stored can be processed quickly, which enhances the performance.
- PostgreSQL supports a wide set of Data Types. Besides, users can create their own custom data type using CREATE TYPE SQL command.

There are different categories of data types in PostgreSQL.

Numeric Types

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

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

Name	Storage Size	Description	Range
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. Values of the numeric, int, and bigint data types can be cast to money. Using Floating point numbers is not recommended to handle money due to the potential for rounding errors.

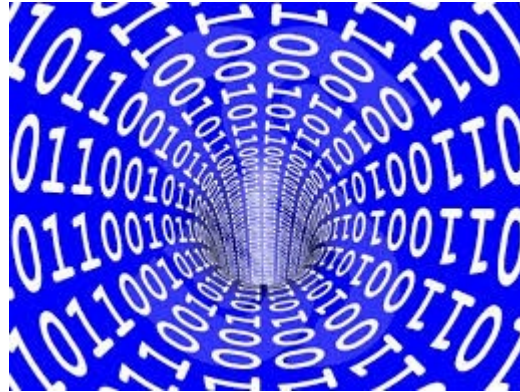
Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

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

Binary Data Types



The bytea data type allows storage of binary strings as in the table below.

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

Date/Time Types

PostgreSQL supports the full set of SQL date and time types, as shown in table below. Dates are counted according to the Gregorian calendar. Here, all the types have resolution of 1 microsecond / 14 digits except date type, whose resolution is day.



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

Boolean Type

PostgreSQL provides the standard SQL type boolean. The boolean type can have several states: true, false, and a third state, unknown, which is represented by the SQL null value.

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



Enumerated Type

ENUM ***Data Type***

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the enum types supported in a number of programming languages.

Unlike other types, Enumerated Types need to be created using CREATE TYPE command.

This type is used to store a static, ordered set of values; for example, compass directions, i.e., NORTH, SOUTH, EAST, and WEST or days of the week as below:

```
CREATE TYPE week AS ENUM ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun');
```

Enumerated once created, they can be used like any other types.

Geometric Type

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



Name	Storage Size	Representation	Description
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line (not fully implemented)	$((x1,y1),(x2,y2))$
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	Polygon (similar to closed path)	$((x1,y1),...)$
circle	24 bytes	Circle	$\langle(x,y),r\rangle$ (center point and radius)

Network Address Type

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses. It is better to use these types instead of plain text types to store network addresses, because these types offer input error checking and specialized operators and functions.

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

Bit String Type

Bit String Types are used to store bit masks.

They are either 0 or 1.

There are two SQL bit types: `bit(n)` and `bit varying(n)`, where `n` is a positive integer.

Text Search Type

This type supports **full text search**, which is the activity of searching through a collection of natural-language documents to locate those that best match a query.

There are two Data Types for this :

Name	Description
tsvector	This is a sorted list of distinct words that have been normalized to merge different variants of the same word, called as "lexemes".
tsquery	This stores lexemes that are to be searched for, and combines them honoring the Boolean operators & (AND), (OR), and ! (NOT). Parentheses can be used to enforce grouping of the operators.

UUID Type

A UUID (Universally Unique Identifiers) 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 is: 550e8400-e29b-41d4-a716-446655440000

XML Type

The xml data type can be used to store XML data. For storing XML data, first you create XML values using function xmlparse as follows:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?>
<tutorial>
<title>PostgreSQL Tutorial </title>
  <topics>...</topics>
</tutorial>')

XMLPARSE (CONTENT 'xyz<foo>bar</foo><bar>foo</bar>')
```


JSON Type

The json data type can be used to store JSON (JavaScript Object Notation) data. Such data can also be stored as text, but the json data type has the advantage of checking that each stored value is a valid JSON value.

There are also related support functions available which can be used directly to handle JSON data type as follows:

Example	Example Result
<code>array_to_json('{{1,5},{99,100}}':int[])</code>	<code>[[1,5],[99,100]]</code>
<code>row_to_json(row(1,'foo'))</code>	<code>{"f1":1,"f2":"foo"}</code>

Json E.g

```
CREATE TABLE orders ( ID serial NOT NULL PRIMARY KEY, info json NOT NULL);
```

```
insert into orders(info)
```

```
Values (
```

```
    '{"customers":"PISPL","item":{"product":"OpenStack","qty":1}}'
```

```
);
```

PostgreSQL provides two native operators -> and ->> to help you query JSON data.

The operator -> returns JSON object field by key.

The operator ->> returns JSON object field by text.

```
mydb=# select info->>'customers' as Customer from orders;
```

```
customer
```

```
-----
```

```
WockHardt
```

```
PISPL
```

```
Infosys
```

```
ITaakash
```

```
(4 rows)
```

PostgreSQL JSON functions

json_each() function allows us to expand the outermost JSON object into a set of key-value pairs.

```
mydb=# select json_each(info) from orders;
          json_each
-----
(customers, ""WockHardt"")
(item, "{"product":"","Medicine","", "qty":"","6}")
(customers, ""PISPL"")
(item, "{"product":"","OpenStack","", "qty":"","1}")
(customers, ""Infosys"")
(item, "{"product":"","PG","", "qty":"","1}")
(customers, ""ITaakash"")
(item, "{"product":"","PG","", "qty":"","2}")
(8 rows)
```

Guess what will this function do **json_each_text()**

To get a set of keys in the outermost JSON object, you use the `json_object_keys()` function.

```
SELECT
    json_object_keys (info->'items')
FROM
    orders;
```

json_typeof() function returns type of the outermost JSON value as a string.

It can be number, boolean, null, object, array, and string.

```
SELECT
    json_typeof (info->'items')
FROM
    orders;
```

Array Type

Arrays of any built-in or user-defined base type, enum type, or composite type can be created.

Declaration of Arrays

Array type can be declared as :

```
CREATE TABLE monthly_savings (  
    name text,  
    saving_per_quarter integer[],  
    scheme text[][]  
);
```

or by using keyword "ARRAY" as:

```
CREATE TABLE monthly_savings (  
    name text,  
    saving_per_quarter integer ARRAY[4],  
    scheme text[][]  
);
```

Inserting values

Array values can be inserted as a literal constant, enclosing the element values within curly braces and separating them by commas. An example is as below:

```
INSERT INTO monthly_savings
VALUES ('Angelina',
'{20000, 14600, 23500, 13250}',
'{"FD", "MF"}, {"FD", "Property"}');
```

Accessing Arrays

An example for accessing Arrays is shown below. The command below will select persons whose savings are more in second quarter than fourth quarter.

```
SELECT name FROM monthly_savings WHERE saving_per_quarter[2] >
saving_per_quarter[4];
```

Modifying Arrays

An example of modifying arrays is as shown below.

```
UPDATE monthly_savings SET saving_per_quarter = '{25000,25000,27000,27000}'  
WHERE name = 'Angelina';
```

or

using the ARRAY expression syntax:

```
UPDATE monthly_savings SET saving_per_quarter = ARRAY[25000,25000,27000,27000]  
WHERE name = 'Angelina';
```

Searching Arrays

An example of searching arrays is as shown below.

```
SELECT * FROM monthly_savings WHERE saving_per_quarter[1] = 10000
```

OR

```
saving_per_quarter[2] = 10000
```

OR

```
saving_per_quarter[3] = 10000
```

OR

```
saving_per_quarter[4] = 10000;
```

If the size of array is known, above search method can be used.

Else, the following example shows how to search when size is not known.

```
SELECT * FROM monthly_savings WHERE 10000 = ANY (saving_per_quarter);
```


Composite Types

Declaration of Composite Types

The following example shows how to declare a composite type:

```
CREATE TYPE inventory_item AS (  
    name text,  
    supplier_id integer,  
    price numeric  
);
```

This data type can be used in the create tables as below:

```
CREATE TABLE on_hand (  
    item inventory_item,  
    count integer  
);
```

```
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```



Accessing Composite Types

To access a field of a composite column, use a dot followed by the field name, much like selecting a field from a table name.

E.g

To select some subfields from our on_hand example table, the query would be as shown below:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

you can even use the table name as well (for instance in a multitable query), like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Range Types

- Range types *represent data type that uses a range of data*. Range type can be discrete ranges (e.g., all integer values 1 to 10) or continuous ranges (e.g., any point in time between 10:00am and 11:00am).

The built-in range types available include ranges:

- **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

Custom range types can be created to make new types of ranges available, such as IP address ranges using the inet type as a base, or float ranges using the float data type as a base.

- Range types support inclusive and exclusive range boundaries using the [] and () characters, respectively, e.g., '[4,9]' represents all integers starting from and including 4 up to but not including 9.

E.g Ranges

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
```

Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables.

If **WITH OIDS** is specified or default_with_oids configuration variable is enabled, only in such cases OIDs are added to user-created tables.

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

Pseudo Types

PostgreSQL type system contains a number of special-purpose entries that are collectively called pseudo-types.

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

Name	Description
any	Indicates that a function accepts any input data type.
anyelement	Indicates that a function accepts any data type.
anyarray	Indicates that a function accepts any array data type.
anynonarray	Indicates that a function accepts any non-array data type.
anyenum	Indicates that a function accepts any enum data type.
anyrange	Indicates that a function accepts any range data type.
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.

PostgreSQL CREATE Database

- Using CREATE DATABASE, an SQL command.
- Using createdb a command-line executable.

Using CREATE DATABASE

This command will create a database from PostgreSQL shell prompt, but you should have appropriate privilege to create database. By default, the new database will be created by cloning the standard system database template1.

Syntax

- The basic syntax of CREATE DATABASE statement is as follows:

CREATE DATABASE dbname;

where dbname is the name of a database to create.

Example

Following is a simple example, which will create testdb in your PostgreSQL schema:

postgres=# CREATE DATABASE testdb;

postgres-#

Using createdb Command

PostgreSQL command line executable createdb is a wrapper around the SQL command CREATE DATABASE. The only difference between this command and SQL command CREATE DATABASE is that the former can be directly run from the command line and it allows a comment to be added into the database, all in one command.

Open the command prompt and go to the directory where PostgreSQL is installed. Go to the bin directory and execute the following command to create a database.

```
createdb -h localhost -p 5432 -U postgres testdb
```

```
password *****
```

Above command will prompt you for password of the PostgreSQL admin user which is postgres by default so provide password and proceed to create your new database.


```
postgres-# \l
```

List of databases

Name	Owner	Encoding	Collate	Ctype	Access privileges
postgres	postgres	UTF8	C	C	
template0	postgres	UTF8	C	C	=c/postgres + postgres=CTc/postgres
template1	postgres	UTF8	C	C	=c/postgres + postgres=CTc/postgres
testdb	postgres	UTF8	C	C	

(4 rows)

```
postgres-#
```

PostgreSQL - SELECT Database

You can select database using either of the following methods:

- 1) Database SQL Prompt
- 2) OS Command Prompt

Database SQL Prompt

Assume you already have launched your PostgreSQL client and you have landed at the following SQL prompt:

postgres=#

You can check available database list using \l, i.e., backslash l command as follows:

postgres=# \l

```
postgres=# \l
```

List of databases					
Name	Owner	Encoding	Collate	Ctype	Access privileges
postgres	postgres	UTF8	C	C	
template0	postgres	UTF8	C	C	=c/postgres +
					postgres=CTc/postgres
template1	postgres	UTF8	C	C	=c/postgres +
					postgres=CTc/postgres
testdb	postgres	UTF8	C	C	

(4 rows)

```
postgres=#
```

Now, type the below command to connect/select a desired database, here we will connect to the testdb database:

```
postgres=# \c testdb;
psql (9.2.4)
Type "help" for help.
You are now connected to database "testdb" as user "postgres".
testdb=#
```

OS Command Prompt

You can select your database from command prompt itself at the time when you login to your database. Following is the simple example:

```
psql -h localhost -p 5432 -U postgres testdb
```

```
Password for user postgres: ****
```

```
psql (9.2.4)
```

```
Type "help" for help.
```

```
You are now connected to database "testdb" as user "postgres".
```

```
testdb=#
```

You are now logged into PostgreSQL testdb and ready to execute your commands inside testdb. To exit from the database, you can use the command \q.

PostgreSQL - DROP Database

In this slides we will discuss how to delete the database in PostgreSQL. They are two options to delete a database:

Using **DROP DATABASE**, an SQL command.

Using **dropdb** a command-line executable.

Be careful before using this operation because by deleting an existing database would result in loss of complete information stored in the database.

Example

Following is a simple example, which will delete testdb from your PostgreSQL schema:

```
postgres=# DROP DATABASE testdb;  
postgres-#
```

Using dropdb Command

- PostgreSQL command line executable dropdb is command-line wrapper around the SQL command DROP DATABASE.
- There is no effective difference between dropping databases via this utility and via other methods for accessing the server. dropdb destroys an existing PostgreSQL database.
- The user, who executes this command must be a database superuser or the owner of the database.

Example

Following example demonstrates deleting a database from OS command prompt:

```
dropdb -h localhost -p 5432 -U postgres testdb  
Password for user postgres: ****
```

The above command drops database testdb. Here, I've used the postgres (found under the pg_roles of template1) user name to drop the database.

PostgreSQL - CREATE Table

Examples

Following is an example, which creates a COMPANY table with ID as primary key and NOT NULL are the constraints showing that these fields can not be NULL while creating records in this table:

```
CREATE TABLE COMPANY(  
    ID INT PRIMARY KEY     NOT NULL,  
    NAME           TEXT     NOT NULL,  
    AGE            INT       NOT NULL,  
    ADDRESS        CHAR(50),  
    SALARY         REAL  
);
```

You can verify if your table has been created successfully using \d command, which will be used to list down all the tables in an attached database.

```
testdb-# \d
```

```
CREATE TABLE DEPARTMENT(  
    ID INT PRIMARY KEY     NOT NULL,  
    DEPT          CHAR(50) NOT NULL,  
    EMP_ID        INT       NOT NULL  
);
```

```

List of relations
Schema | Name      | Type  | Owner
-----+-----+-----+-----
public | company   | table | postgres
public | department | table | postgres
(2 rows)

```

Use `\d tablename` to describe each table as shown below:

testdb-# `\d company`

Above PostgreSQL statement will produce the following result:

```

Table "public.company"
Column | Type          | Modifiers
-----+-----+-----
id      | integer       | not null
name    | text          | not null
age     | integer       | not null
address | character(50) |
salary  | real          |
join_date | date          |
Indexes:
    "company_pkey" PRIMARY KEY, btree (id)

```


PostgreSQL - DROP Table

The PostgreSQL DROP TABLE statement is used to remove a table definition and all associated data, indexes, rules, triggers, and constraints for that table.

You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Example

We had created the tables DEPARTMENT and COMPANY in the previous chapter. First verify these tables (use \d to list the tables):

```
testdb-# \d
```

This would produce the following result:

List of relations			
Schema	Name	Type	Owner
public	company	table	postgres
public	department	table	postgres
(2 rows)			

This means DEPARTMENT and COMPANY tables are present. So let us drop them as follows:

```
testdb=# drop table department, company;
```

This would produce the following result:
DROP TABLE

```
testdb=# \d  
relations found.
```

```
Testdb=#
```

The message returned DROP TABLE indicates that drop command had been executed successfully.

PostgreSQL Schema

A **schema is a named collection of tables**. A schema can also contain views, indexes, sequences, data types, operators, and functions.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested. PostgreSQL statement CREATE SCHEMA creates a schema.

Example

Let us see an example for creating a schema. Connect to the database testdb and create a schema myschema as follows:

```
testdb=# create schema myschema;  
CREATE SCHEMA
```

The message "CREATE SCHEMA" signifies that the schema is created successfully.

Now, let us create a table in the above schema as follows:

```
testdb=# create table myschema.company(  
  ID    INT                NOT NULL,  
  NAME  VARCHAR (20)       NOT NULL,  
  AGE   INT                NOT NULL,  
  ADDRESS CHAR (25) ,  
  SALARY DECIMAL (18, 2),  
  PRIMARY KEY (ID)  
);
```

This will create an empty table. You can verify the table created with the command below:

```
testdb=# select * from myschema.company;
```

This would produce the following result:

```
id | name | age | address | salary
```

```
----+-----+-----+-----+-----
```

```
(0 rows)
```

Drop schema

To drop a schema if it's empty (all objects in it have been dropped), then use:

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

Advantages of using a Schema

- 1) It allows many users to use one database without interfering with each other.
- 2) It organizes database objects into logical groups to make them more manageable.
- 3) Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

PostgreSQL - INSERT Query

PostgreSQL INSERT INTO statement allows one to insert new rows into a table. One can insert a single row at a time or several rows as a result of a query.

Basic syntax of INSERT INTO statement is as follows.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
```

```
VALUES (value1, value2, value3,...valueN);
```

Examples

Let us create COMPANY table in testdb as follows:

```
CREATE TABLE COMPANY(  
  ID INT PRIMARY KEY      NOT NULL,  
  NAME          TEXT      NOT NULL,  
  AGE           INT       NOT NULL,  
  ADDRESS       CHAR(50),  
  SALARY        REAL,  
  JOIN_DATE     DATE  
);
```

Following example inserts a row into the COMPANY table:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,JOIN_DATE) VALUES (1, 'Paul', 32, 'California', 20000.00 , '2001-07-13');
```

Following example is to insert a row; here salary column is omitted and therefore it will have the default value:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,JOIN_DATE) VALUES (2, 'Allen', 25, 'Texas', '2007-12-13');
```

Following example uses the DEFAULT clause for the ADDRESS columns rather than specifying a value:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,JOIN_DATE) VALUES (3, 'Teddy', 23, 'Norway', 20000.00, DEFAULT );
```

Following example inserts multiple rows using the multirow VALUES syntax:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,JOIN_DATE) VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00, '2007-12-13' ), (5, 'David', 27, 'Texas', 85000.00 , '2007-12-13');
```

All the above statements would create the following records in COMPANY table.

ID	NAME	AGE	ADDRESS	SALARY	JOIN_DATE
1	Paul	32	California	20000.0	2001-07-13
2	Allen	25	Texas		2007-12-13
3	Teddy	23	Norway	20000.0	
4	Mark	25	Rich-Mond	65000.0	2007-12-13
5	David	27	Texas	85000.0	2007-12-13

PostgreSQL - SELECT Query

PostgreSQL SELECT statement is used to fetch the data from a database table which returns data in the form of result table. These result tables are called result-sets.

Example:

Consider the table COMPANY having records as follows

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is an example, which would fetch ID, Name and Salary fields of the customers available in CUSTOMERS table:

testdb=# SELECT ID, NAME, SALARY FROM COMPANY ;

id	name	salary
1	Paul	20000
2	Allen	15000
3	Teddy	20000
4	Mark	65000
5	David	85000
6	Kim	45000
7	James	10000

(7 rows)

If you want to fetch all the fields of CUSTOMERS table, then use the following query:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

PostgreSQL - WHERE Clause

PostgreSQL WHERE clause is used to specify a condition while fetching the data from single table or joining with multiple tables.

- If the given condition is satisfied only then it returns specific value from the table. You can filter out rows that you don't want included in the result-set by using the WHERE clause.
- The WHERE clause not only is used in SELECT statement, but it is also used in UPDATE, DELETE statement, etc

Example

```
testdb# select * from COMPANY;
 id | name  | age | address  | salary
----+-----+----+-----+-----
  1 | Paul  |  32 | California | 20000
  2 | Allen |  25 | Texas      | 15000
  3 | Teddy |  23 | Norway     | 20000
  4 | Mark  |  25 | Rich-Mond  | 65000
  5 | David |  27 | Texas      | 85000
  6 | Kim   |  22 | South-Hall | 45000
  7 | James |  24 | Houston    | 10000
(7 rows)
```

simple examples showing usage of PostgreSQL Logical Operators. Following SELECT statement will list down all the records where AGE is greater than or equal to 25 AND salary is greater than or equal to 65000.00:

```
testdb=# SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(2 rows)

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 OR salary is greater than or equal to 65000.00:

```
testdb=# SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(4 rows)

Following SELECT statement lists down all the records where AGE is not NULL which means all the records because none of the record is having AGE equal to NULL:

testdb=# SELECT * FROM COMPANY WHERE AGE IS NOT NULL;

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following SELECT statement lists down all the records where NAME starts with 'Pa', does not matter what comes after 'Pa'.

```
testdb=# SELECT * FROM COMPANY WHERE NAME LIKE 'Pa%';
```

Above PostgreSQL statement will produce the following result:

id	 	name	 	age	 	address	 	salary
1		Paul		32		California		20000

Following SELECT statement lists down all the records where AGE value is either 25 or 27:

```
testdb=# SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
2	Allen	25	Texas	15000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(3 rows)

Following SELECT statement lists down all the records where AGE value is neither 25 nor 27:

```
testdb=# SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
1	Paul	32	California	20000
3	Teddy	23	Norway	20000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(4 rows)

Following SELECT statement lists down all the records where AGE value is in BETWEEN 25 AND 27:

```
testdb=# SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
2	Allen	25	Texas	15000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(3 rows)

Following SELECT statement makes use of SQL sub-query *where sub-query finds all the records with AGE field having SALARY > 65000* and later WHERE clause is being used along with EXISTS operator to list down all the records where AGE from the outside query exists in the result returned by sub-query:

```
testdb=# SELECT AGE FROM COMPANY  
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
```

Following SELECT statement makes use of SQL sub-query where subquery finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with > operator to list down all the records where AGE from outside query is greater than the age in the result returned by sub-query:

```
testdb=# SELECT * FROM COMPANY  
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
1	Paul	32	California	20000

PostgreSQL - UPDATE Query

PostgreSQL UPDATE Query is used to modify the existing records in a table. You can use WHERE clause with UPDATE query to update selected rows otherwise all the rows would be updated.

Example:

Consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is an example, which would update ADDRESS for a customer, whose ID is 3 :

```
testdb=# UPDATE COMPANY SET SALARY = 15000 WHERE ID = 3;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
3	Teddy	23	Norway	15000

(7 rows)

If you want to modify all ADDRESS and SALARY column values in COMPANY table, you do not need to use WHERE clause and UPDATE query would be as follows:

```
testdb=# UPDATE COMPANY SET ADDRESS = 'Texas', SALARY=20000;
```

id	name	age	address	salary
1	Paul	32	Texas	20000
2	Allen	25	Texas	20000
4	Mark	25	Texas	20000
5	David	27	Texas	20000
6	Kim	22	Texas	20000
7	James	24	Texas	20000
3	Teddy	23	Texas	20000

(7 rows)

PostgreSQL - DELETE Query

PostgreSQL DELETE Query is used to delete the existing records from a table. You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

Example:

Consider the table COMPANY having records as follows:

```
# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is an example which would DELETE a customer, whose ID is 7:

```
testdb=# DELETE FROM COMPANY WHERE ID = 2;
```

id	name	age	address	salary
1	Paul	32	California	20000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(6 rows)

If you want to DELETE all the records from COMPANY table, you do not need to use WHERE clause with DELETE queries, which would be as follows:

```
testdb=# DELETE FROM COMPANY;
```

Now, COMPANY table does not have any record because all the records have been deleted by DELETE statement.

like

PostgreSQL LIKE operator is used to match text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the LIKE operator will return true, which is 1.

There are two wildcards used in conjunction with the LIKE operator:

- 1) The percent sign (%)
- 2) The underscore (_)

The **percent sign** represents zero, one, or multiple numbers or characters.

The **underscore** represents a single number or character. These symbols can be used in combinations.

If either of these two signs is not used in conjunction with the LIKE clause, then the LIKE acts like the equals operator.

LIKE

```
SELECT FROM table_name  
WHERE column LIKE 'XXXX%'
```

or

```
SELECT FROM table_name  
WHERE column LIKE '%XXXX%'
```

or

```
SELECT FROM table_name  
WHERE column LIKE 'XXXX_'
```

or

```
SELECT FROM table_name  
WHERE column LIKE '_XXXX'
```

or

```
SELECT FROM table_name  
WHERE column LIKE '_XXXX_'
```

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '_' operators:

Statement	Description
WHERE SALARY::text LIKE '200%'	Finds any values that start with 200
WHERE SALARY::text LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY::text LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE SALARY::text LIKE '2_%_ %'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY::text LIKE '%2'	Finds any values that end with 2
WHERE SALARY::text LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3
WHERE SALARY::text LIKE '2__3'	Finds any values in a five-digit number that start with 2 and end with 3

Note: Postgres LIKE is String compare only. Hence, we need to explicitly cast the integer column to string as in the examples.

Let us take a real example, consider the table COMPANY having records as follows:

```
# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is an example, which would display all the records from COMPANY table where AGE starts with 2:

```
testdb=# SELECT * FROM COMPANY WHERE AGE::text LIKE '2%';
```

id	name	age	address	salary
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
8	Paul	24	Houston	20000

(7 rows)

Following is an example, which would display all the records from COMPANY table where ADDRESS will have a hyphen (-) inside the text:

```
testdb=# SELECT * FROM COMPANY WHERE ADDRESS LIKE '%-%';
```

id	name	age	address	salary
4	Mark	25	Rich-Mond	65000
6	Kim	22	South-Hall	45000

(2 rows)

Limit

PostgreSQL LIMIT clause is used to limit the data amount returned by the SELECT statement.

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

Example:

Consider the table COMPANY having records as follows:

```
# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is an example which limits the row in the table according to the number of rows you want to fetch from table:

```
testdb=# SELECT * FROM COMPANY LIMIT 4;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000

(4 rows)

But in certain situation you may need to pick up a set of records from a particular offset. Here is an example which picks up 3 records starting from 3rd position:

testdb=# SELECT * FROM COMPANY LIMIT 3 OFFSET 2;

id	name	age	address	salary
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(3 rows)

Order BY

PostgreSQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns.

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be available in column-list.

Example:

Consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is an example, which would sort the result in descending order by SALARY:

testdb=# SELECT * FROM COMPANY ORDER BY AGE ASC;

id	name	age	address	salary
6	Kim	22	South-Hall	45000
3	Teddy	23	Norway	20000
7	James	24	Houston	10000
8	Paul	24	Houston	20000
4	Mark	25	Rich-Mond	65000
2	Allen	25	Texas	15000
5	David	27	Texas	85000
1	Paul	32	California	20000
9	James	44	Norway	5000
10	James	45	Texas	5000

(10 rows)

Following is an example, which would sort the result in descending order by NAME and SALARY:

```
testdb=# SELECT * FROM COMPANY ORDER BY NAME, SALARY ASC;
```

id	name	age	address	salary
2	Allen	25	Texas	15000
5	David	27	Texas	85000
10	James	45	Texas	5000
9	James	44	Norway	5000
7	James	24	Houston	10000
6	Kim	22	South-Hall	45000
4	Mark	25	Rich-Mond	65000
1	Paul	32	California	20000
8	Paul	24	Houston	20000
3	Teddy	23	Norway	20000

(10 rows)

Following is an example, which would sort the result in descending order by NAME:

```
testdb=# SELECT * FROM COMPANY ORDER BY NAME DESC;
```

id	name	age	address	salary
3	Teddy	23	Norway	20000
1	Paul	32	California	20000
8	Paul	24	Houston	20000
4	Mark	25	Rich-Mond	65000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
9	James	44	Norway	5000
10	James	45	Texas	5000
5	David	27	Texas	85000
2	Allen	25	Texas	15000

(10 rows)

Group by

PostgreSQL GROUP BY clause is used in collaboration with the SELECT statement to group together those rows in a table that have identical data.

This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Example:

Consider the table COMPANY having records as follows:

```
# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

If you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

```
testdb=# SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

```
name  | sum
-----+-----
Teddy  | 20000
Paul   | 20000
Mark   | 65000
David  | 85000
Allen  | 15000
Kim    | 45000
James  | 10000
(7 rows)
```


Now, let us create three more records in COMPANY table using the following INSERT statements:

```
INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00);  
INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00);  
INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00);
```

Now, our table has the following records with duplicate names:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
8	Paul	24	Houston	20000
9	James	44	Norway	5000
10	James	45	Texas	5000

(10 rows)

Again, let us use the same statement to group-by all the records using NAME column as follows:

```
testdb=# SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME  
ORDER BY NAME;
```

name	sum
Allen	15000
David	85000
James	20000
Kim	45000
Mark	65000
Paul	40000
Teddy	20000
(7 rows)	

Let us use ORDER BY clause along with GROUP BY clause as follows:

```
testdb=# SELECT NAME, SUM(SALARY)
        FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
```

This would produce the following result:

name	sum
-----+	
Teddy	20000
Paul	40000
Mark	65000
Kim	45000
James	20000
David	85000
Allen	15000
(7 rows)	

with

PostgreSQL, the WITH query provides a way to write auxiliary statements for use in a larger query.

It helps in breaking down complicated and large queries into simpler forms, which are easily readable.

These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query.

The WITH query being CTE query, is particularly useful when subquery is executed multiple times.

It is equally helpful in place of temporary tables.

It computes the aggregation once and allows us to reference it by its name (may be multiple times) in the queries.

The WITH clause must be defined before it is used in the query.

Recursive WITH

Recursive WITH or Hierarchical queries, is a form of CTE where a CTE can reference to itself, i.e., a WITH query can refer to its own output, hence the name recursive.

Example

Consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Now, let us write a query using the WITH clause to select the records from the above table, as follows:

```
With CTE AS  
(Select  
  ID  
, NAME  
, AGE  
, ADDRESS  
, SALARY  
FROM COMPANY )  
Select * From CTE;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Now, let us write a query using the RECURSIVE keyword along with the WITH clause, to find the sum of the salaries less than 20000, as follows:

```
WITH RECURSIVE t(n) AS (  
    VALUES (0)  
    UNION ALL  
    SELECT SALARY FROM COMPANY WHERE SALARY < 20000  
)  
SELECT sum(n) FROM t;
```

sum
25000
(1 row)

Let us write a query using data modifying statements along with the WITH clause, as follows.

Create a table COMPANY1 similar to the table COMPANY.

The query in the example, effectively moves rows from COMPANY to COMPANY1.

The DELETE in WITH deletes the specified rows from COMPANY, returning their contents by means of its RETURNING clause and then the primary query reads that output and inserts it into COMPANY1 TABLE:

```
CREATE TABLE COMPANY1(  
    ID INT PRIMARY KEY      NOT NULL,  
    NAME          TEXT      NOT NULL,  
    AGE           INT        NOT NULL,  
    ADDRESS       CHAR(50),  
    SALARY        REAL  
);  
  
WITH moved_rows AS (  
    DELETE FROM COMPANY  
    WHERE  
        SALARY >= 30000  
    RETURNING *  
)  
INSERT INTO COMPANY1 (SELECT * FROM moved_rows);
```


Previous slide PostgreSQL statement will produce the following result:

INSERT 0 3

Now, the records in the tables COMPANY and COMPANY1 are as follows:

```
testdb=# SELECT * FROM COMPANY;
 id | name  | age | address  | salary
-----+-----+-----+-----+-----
  1 | Paul  | 32  | California | 20000
  2 | Allen | 25  | Texas      | 15000
  3 | Teddy | 23  | Norway     | 20000
  7 | James | 24  | Houston    | 10000
(4 rows)
```

```
testdb=# SELECT * FROM COMPANY1;
 id | name  | age | address  | salary
-----+-----+-----+-----+-----
  4 | Mark  | 25  | Rich-Mond | 65000
  5 | David | 27  | Texas     | 85000
  6 | Kim   | 22  | South-Hall | 45000
(3 rows)
```

Having

HAVING clause allows us to pick out particular rows where the function's result meets some condition.

Difference – Having & Where clause

The **WHERE** clause places conditions on the selected columns, whereas the **HAVING** clause places conditions on groups created by the **GROUP BY** clause.

Example:

Consider the table **COMPANY** having records as follows:

```
# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is the example, which would display record for which name count is less than 2:

```
testdb-# SELECT NAME FROM COMPANY GROUP BY name HAVING  
count(name) < 2;
```

```
name  
-----  
Teddy  
Paul  
Mark  
David  
Allen  
Kim  
James  
(7 rows)
```

Now, let us create three more records in COMPANY table using the following INSERT statements:

```
INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00);  
INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00);  
INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00);
```

Now, our table has the following records with duplicate names:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
8	Paul	24	Houston	20000
9	James	44	Norway	5000
10	James	45	Texas	5000

(10 rows)

Following is the example, which would display record for which name count is greater than 1:

```
testdb-# SELECT NAME FROM COMPANY GROUP BY name HAVING  
count(name) > 1;
```

This would produce the following result:

```
name  
-----  
Paul  
James  
(2 rows)
```

PostgreSQL DISTINCT keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table.

While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Example:

Consider the table COMPANY having records as follows:

```
# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Let us add two more records to this table as follows:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Paul', 32, 'California', 20000.00 );
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (9, 'Allen', 25, 'Texas', 15000.00 );
```

Now, the records in the COMPANY table would be:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
8	Paul	32	California	20000
9	Allen	25	Texas	15000

(9 rows)

First, let us see how the following SELECT query returns duplicate salary records:

testdb=# SELECT name FROM COMPANY;

```
name
-----
Paul
Allen
Teddy
Mark
David
Kim
James
Paul
Allen
(9 rows)
```


Now, let us use DISTINCT keyword with the above SELECT query and see the result:

```
testdb=# SELECT DISTINCT name FROM COMPANY;
```

This would produce the following result where we do not have any duplicate entry:

```
name
-----
Teddy
Paul
Mark
David
Allen
Kim
James
(7 rows)
```

Advanced PostgreSQL

Constraints

Constraints are the rules enforced on data columns on table.

These are used to prevent invalid data from being entered into the database. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level.

Column level constraints are applied only to one column where as table level constraints are applied to the whole table.

Defining a data type for a column is a constraint in itself.

For example:

a column of type DATE constrains the column to valid dates.

Following are commonly used constraints available in PostgreSQL.

NOT NULL Constraint: Ensures that a column cannot have NULL value.

UNIQUE Constraint: Ensures that all values in a column are different.

PRIMARY Key: Uniquely identifies each row/record in a database table.

FOREIGN Key: Constrains data based on columns in other tables.

CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.

EXCLUSION Constraint: The EXCLUDE constraint ensures that if any two rows are compared on the specified column(s) or expression(s) using the specified operator(s), not all of these comparisons will return TRUE.

NOT NULL Constraint

By default, a column can hold NULL values.

If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NOT NULL constraint is always written as a column constraint.

A NULL is not the same as no data, rather, it represents unknown data.

Example:

The following statement creates a new table called COMPANY1 and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULL values:

```
CREATE TABLE COMPANY1(  
    ID INT PRIMARY KEY      NOT NULL,  
    NAME TEXT               NOT NULL,  
    AGE INT                 NOT NULL,  
    ADDRESS CHAR(50),  
    SALARY REAL  
);
```

UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a particular column.

In the COMPANY table, for example, you might want to prevent two or more people from having identical age.

Example:

The following PostgreSQL statement creates a new table called COMPANY3 and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE COMPANY3(  
    ID INT PRIMARY KEY      NOT NULL,  
    NAME TEXT              NOT NULL,  
    AGE INT                NOT NULL UNIQUE,  
    ADDRESS CHAR(50),  
    SALARY REAL            DEFAULT 50000.00  
);
```

PRIMARY KEY Constraint

PRIMARY KEY constraint uniquely identifies each record in a database table.

There can be more UNIQUE columns, but only one primary key in a table. Primary keys are important when designing the database tables. Primary keys are unique ids.

We use them to refer to table rows. Primary keys become foreign keys in other tables. when creating relations among tables.

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Example:

You already have seen various examples above where we have created COMPANY4 table with ID as primary key:

```
CREATE TABLE COMPANY4(  
    ID INT PRIMARY KEY NOT NULL,  
    NAME TEXT NOT NULL,  
    AGE INT NOT NULL,  
    ADDRESS CHAR(50),  
    SALARY REAL  
);
```


FOREIGN KEY Constraint

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. They are called foreign keys because the constraints are foreign; that is, outside the table.

Foreign keys are sometimes called a referencing key.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY5 and adds five columns.

```
CREATE TABLE COMPANY6(  
    ID INT PRIMARY KEY      NOT NULL,  
    NAME          TEXT      NOT NULL,  
    AGE           INT       NOT NULL,  
    ADDRESS       CHAR(50),  
    SALARY        REAL  
);
```

E.g

The following PostgreSQL statement creates a new table called DEPARTMENT1, which adds three columns.

The column EMP_ID is the foreign key and references the ID field of the table COMPANY6.

```
CREATE TABLE DEPARTMENT1(  
    ID INT PRIMARY KEY      NOT NULL,  
    DEPT          CHAR(50) NOT NULL,  
    EMP_ID        INT       references COMPANY6(ID)  
);
```

CHECK Constraint

CHECK Constraint enables a condition to check the value being entered into a record.

If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

Example:

For example, the following PostgreSQL statement creates a new table called COMPANY5 and adds five columns.

Here, we add a CHECK with SALARY column, so that you can not have any SALARY Zero:

```
CREATE TABLE COMPANY5(  
    ID INT PRIMARY KEY      NOT NULL,  
    NAME                    TEXT    NOT NULL,  
    AGE                    INT      NOT NULL,  
    ADDRESS                CHAR(50),  
    SALARY                 REAL     CHECK(SALARY > 0)  
);
```

EXCLUSION Constraint

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**.

Example

The following PostgreSQL statement creates a new table called COMPANY7 and adds five columns. Here, we add an EXCLUDE constraint:

```
CREATE TABLE COMPANY7(  
  ID INT PRIMARY KEY      NOT NULL,  
  NAME          TEXT ,  
  AGE           INT ,  
  ADDRESS       CHAR(50),  
  SALARY        REAL,  
  EXCLUDE USING gist  
  (NAME WITH =,  
   AGE WITH <>)  
);
```

Here, USING gist is the type of index to build and use for enforcement.

You need to execute the command

CREATE EXTENSION btree_gist;

once per database.

This will install the **btree_gist extension**, which defines the exclusion constraints on plain scalar data types.

As we have enforced the age has to be same, let's see this by inserting records to the table:

```
INSERT INTO COMPANY7 VALUES(1, 'Paul', 32, 'California', 20000.00 );  
INSERT INTO COMPANY7 VALUES(2, 'Paul', 32, 'Texas', 20000.00 );  
INSERT INTO COMPANY7 VALUES(3, 'Allen', 42, 'California', 20000.00 );
```

For the first two INSERT statements the records are added to the COMPANY7 table.

For the third INSERT statement the following error is displayed:

```
ERROR: duplicate key value violates unique constraint "company7_pkey"  
DETAIL: Key (id)=(3) already exists.
```

Dropping Constraints:

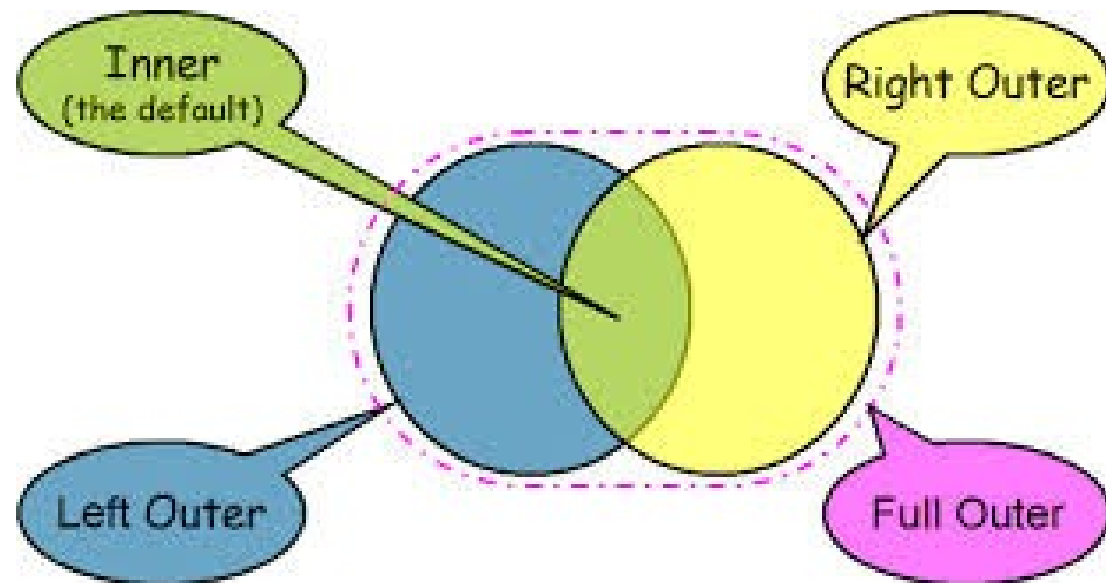
To remove a constraint you need to know its name. If the name is known, it's easy to drop. Else you need to find out the system generated name.

The psql command `\d tablename` can be helpful here.

The general syntax is:

```
ALTER TABLE table_name DROP CONSTRAINT some_name;
```

Joins



Joins

PostgreSQL Joins clause is used to combine records from two or more tables in a database.

A JOIN is a means for combining fields from two tables by using values common to each.

Join Types in PostgreSQL are:

CROSS JOIN

INNER JOIN

LEFT OUTER JOIN

RIGHT OUTER JOIN

FULL OUTER JOIN

We already have seen INSERT statements to populate COMPANY table. So just let's assume the list of records available in COMPANY table:

id	name	age	address	salary	join_date
1	Paul	32	California	20000	2001-07-13
3	Teddy	23	Norway	20000	
4	Mark	25	Rich-Mond	65000	2007-12-13
5	David	27	Texas	85000	2007-12-13
2	Allen	25	Texas		2007-12-13
8	Paul	24	Houston	20000	2005-07-13
9	James	44	Norway	5000	2005-07-13
10	James	45	Texas	5000	2005-07-13

```
CREATE TABLE DEPARTMENT(  
  ID INT PRIMARY KEY      NOT NULL,  
  DEPT          CHAR(50) NOT NULL,  
  EMP_ID        INT       NOT NULL  
);
```

Here is the list of INSERT statements to populate DEPARTMENT table:

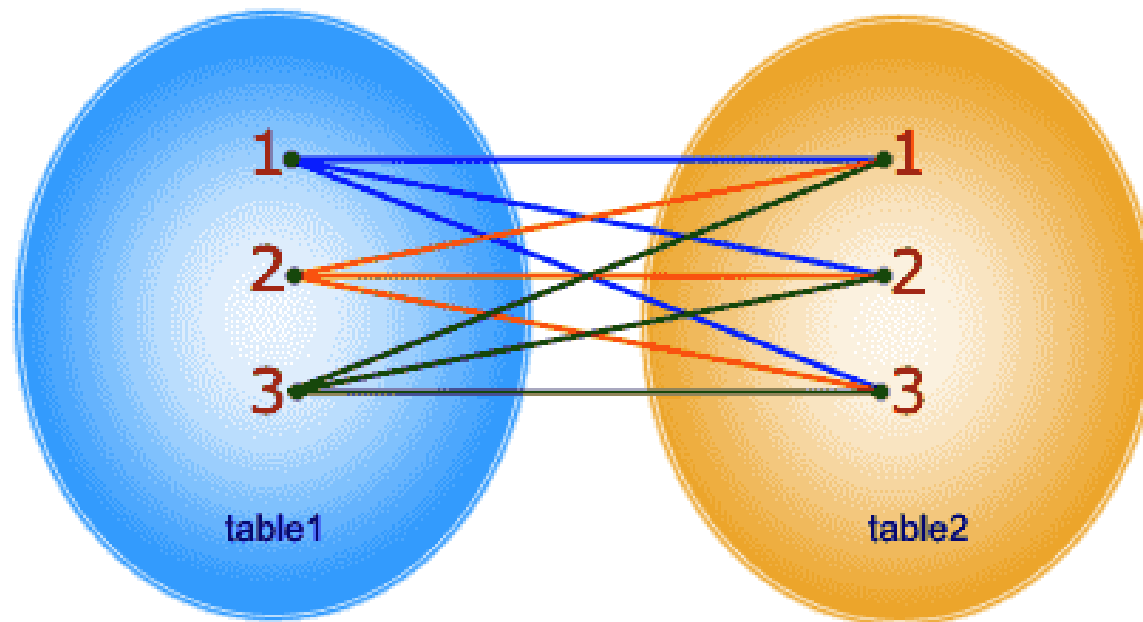
```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID) VALUES (1, 'IT Billing', 1 );  
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID) VALUES (2, 'Engineering', 2 );  
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID) VALUES (3, 'Finance', 7 );
```

Finally, we have the following list of records available in DEPARTMENT table:

id	dept	emp_id
1	IT Billing	1
2	Engineering	2
3	Finance	7

Cross Join

```
SELECT * FROM table1 CROSS JOIN table2;
```



In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be $x * y$ rows.

Cross Join

CROSS JOIN matches every row of the first table with every row of the second table. If the input tables have x and y columns, respectively, the resulting table will have x+y columns.

Because CROSS JOINS have the potential to generate extremely large tables, care must be taken to only use them when appropriate.

we can write a CROSS JOIN as follows:

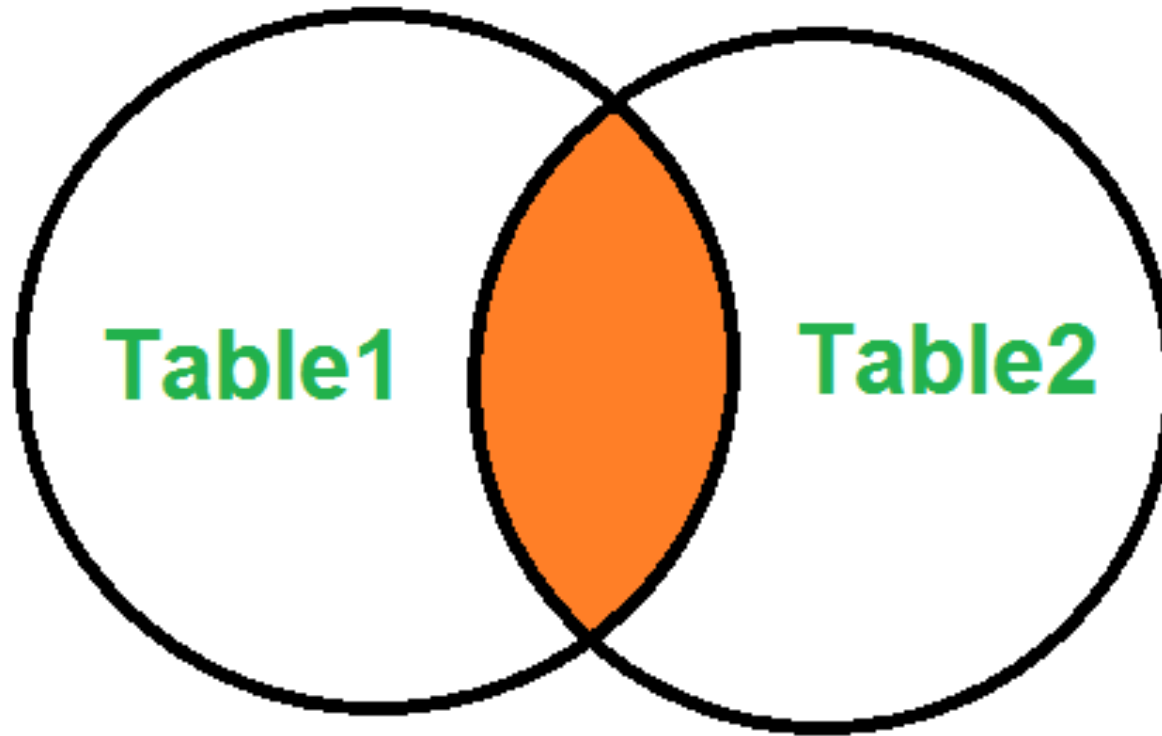
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;

Above query will produce the following result:

emp_id	name	dept
1	Paul	IT Billing
1	Teddy	IT Billing
1	Mark	IT Billing
1	David	IT Billing
1	Allen	IT Billing
1	Paul	IT Billing
1	James	IT Billing
1	James	IT Billing
2	Paul	Engineering

.....

INNER JOIN



```
SELECT * FROM Table1 INNER JOIN Table2 ON Table1.name = Table2.name
```

Inner join

INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows, which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of table1 and table2 are combined into a result row.

An INNER JOIN is the most common type of join and is the default type of join. You can use INNER keyword optionally.

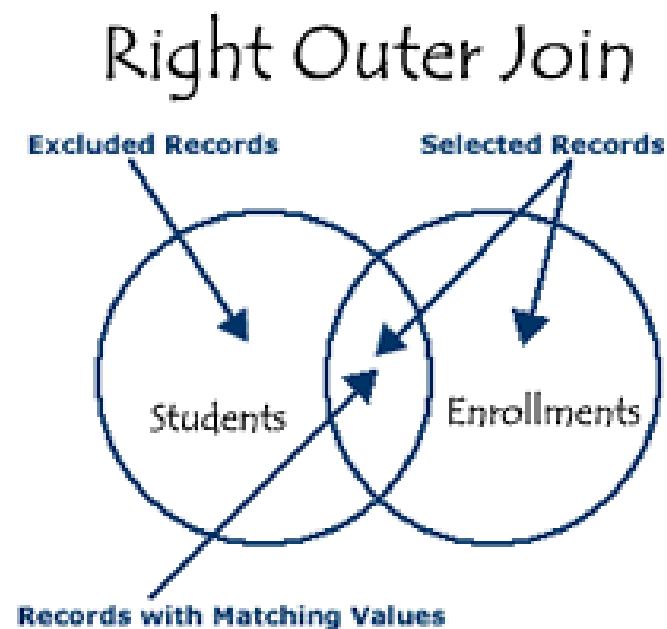
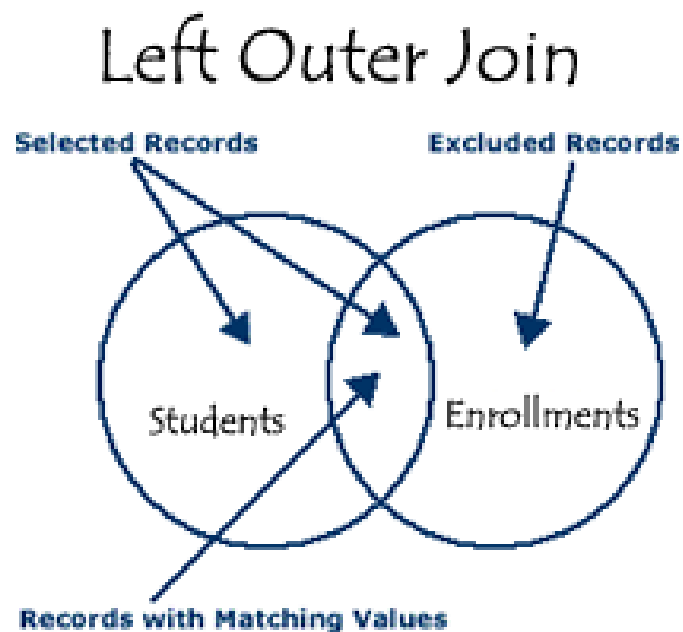
Based on the above tables, we can write an INNER JOIN as follows:

```
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN
          DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Above query will produce the following result:

emp_id	name	dept
1	Paul	IT Billing
2	Allen	Engineering

LEFT OUTER JOIN & RIGHT OUTER JOIN



LEFT OUTER JOIN

SQL standard defines three types of OUTER JOINS: LEFT, RIGHT, and FULL and PostgreSQL supports all of these.

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

Based on the above tables, we can write a inner join as follows:

```
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN  
DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Above query will produce the following result:

emp_id	name	dept
1	Paul	IT Billing
2	Allen	Engineering
	James	
	David	
	Paul	
	Mark	
	Teddy	
	James	

RIGHT OUTER JOIN

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

Based on the above tables, we can write a inner join as follows:

```
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY RIGHT OUTER JOIN  
DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Above query will produce the following result:

emp_id	name	dept
1	Paul	IT Billing
2	Allen	Engineering
7		Finance

FULL OUTER JOIN

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

Based on the above tables, we can write a inner join as follows:

```
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY FULL OUTER JOIN  
DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Above query will produce the following result:

emp_id	name	dept
1	Paul	IT Billing
2	Allen	Engineering
7		Finance
	James	
	David	
	Paul	
	Mark	
	Teddy	
	James	

Union

PostgreSQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order but they do not have to be the same length.

Example:

Consider following two tables,

(a) COMPANY table is as follows:

```
testdb=# SELECT * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

```
testdb=# SELECT * from DEPARTMENT;
 id | dept          | emp_id
-----+-----+-----
  1 | IT Billing     |      1
  2 | Engineering   |      2
  3 | Finance       |      7
  4 | Engineering   |      3
  5 | Finance       |      4
  6 | Engineering   |      5
  7 | Finance       |      6
(7 rows)
```

Now let us join these two tables using SELECT statement along with UNION clause as follows:

```
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
UNION
        SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Union

UNION ALL Clause

UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows. The same rules that apply to UNION apply to the UNION ALL operator as well.\

Example:

Now, let us join above-mentioned two tables in our SELECT statement as follows:

```
testdb=# SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
UNION ALL
        SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

emp_id	name	dept
1	Paul	IT Billing
2	Allen	Engineering
7	James	Finance
3	Teddy	Engineering
4	Mark	Finance
5	David	Engineering
6	Kim	Finance
1	Paul	IT Billing
2	Allen	Engineering
7	James	Finance
3	Teddy	Engineering
4	Mark	Finance
5	David	Engineering
6	Kim	Finance

(14 rows)

NULL

PostgreSQL NULL is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

The basic syntax of using NULL while creating a table is as follows:

```
CREATE TABLE COMPANY(  
  ID INT PRIMARY KEY     NOT NULL,  
  NAME           TEXT     NOT NULL,  
  AGE            INT       NOT NULL,  
  ADDRESS        CHAR(50),  
  SALARY         REAL  
);
```

NOT NULL signifies that column should always accept an explicit value of the given data type.

There are two columns where we did not use NOT NULL. Hence this means these columns could be NULL.

A field with a NULL value is one that has been left blank during record creation.

Example:

The NULL value can cause problems when selecting data, however, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results.

Consider the following table, COMPANY having the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	Paul	32	California	20000.0
2	Allen	25	Texas	15000.0
3	Teddy	23	Norway	20000.0
4	Mark	25	Rich-Mond	65000.0
5	David	27	Texas	85000.0
6	Kim	22	South-Hall	45000.0
7	James	24	Houston	10000.0

Let us use UPDATE statement to set few nullable values as NULL as follows:

testdb=# UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID IN(6,7);

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22		
7	James	24		
(7 rows)				

Next, let us see the usage of **IS NOT NULL** operator to list down all the records where SALARY is not NULL:

```
testdb=# SELECT ID, NAME, AGE, ADDRESS, SALARY FROM COMPANY WHERE SALARY IS NOT NULL;
```

Above PostgreSQL statement will produce the following result:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(5 rows)

Following is the usage of **IS NULL** operator which will list down all the records where SALARY is NULL:

```
testdb=# SELECT ID, NAME, AGE, ADDRESS, SALARY FROM COMPANY WHERE SALARY IS NULL;
```

id	name	age	address	salary
6	Kim	22		
7	James	24		

(2 rows)

Alias

We can rename a table or a column temporarily by giving another name, which is known as ALIAS. The use of table aliases means to rename a table in a particular PostgreSQL statement. Renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular PostgreSQL query.

Example:

Consider the following two tables,

(a) COMPANY table is as follows

```
testdb=# select * from COMPANY;
 id | name  | age | address  | salary
-----+-----+-----+-----+-----
  1 | Paul  |  32 | California | 20000
  2 | Allen |  25 | Texas     | 15000
  3 | Teddy |  23 | Norway    | 20000
  4 | Mark  |  25 | Rich-Mond | 65000
  5 | David |  27 | Texas     | 85000
  6 | Kim   |  22 | South-Hall | 45000
  7 | James |  24 | Houston   | 10000
(7 rows)
```

id	dept	emp_id
1	IT Billing	1
2	Engineering	2
3	Finance	7
4	Engineering	3
5	Finance	4
6	Engineering	5
7	Finance	6

(7 rows)

Now, following is the usage of TABLE ALIAS where we use C and D as aliases for COMPANY and DEPARTMENT tables, respectively:

```
testdb=# SELECT C.ID, C.NAME, C.AGE, D.DEPT FROM COMPANY AS C,
        DEPARTMENT AS D WHERE C.ID = D.EMP_ID;
```

Above PostgreSQL statement will produce the following result:

id	name	age	dept
1	Paul	32	IT Billing
2	Allen	25	Engineering
7	James	24	Finance
3	Teddy	23	Engineering
4	Mark	25	Finance
5	David	27	Engineering
6	Kim	22	Finance

(7 rows)

Let us see an example for the usage of COLUMN ALIAS where COMPANY_ID is an alias of ID column and COMPANY_NAME is an alias of name column:

```
testdb=# SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D.DEPT FROM  
COMPANY AS C, DEPARTMENT AS D WHERE C.ID = D.EMP_ID;
```

Above PostgreSQL statement will produce the following result:

company_id	company_name	age	dept
1	Paul	32	IT Billing
2	Allen	25	Engineering
7	James	24	Finance
3	Teddy	23	Engineering
4	Mark	25	Finance
5	David	27	Engineering
6	Kim	22	Finance

(7 rows)

Triggers

Triggers are database callback functions, which are *automatically performed/invoked when a specified database event occurs*.

Following are important points about PostgreSQL triggers:

- PostgreSQL trigger can be specified to fire before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE or DELETE is attempted); or after the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed); or instead of the operation (in the case of inserts, updates or deletes on a view).
- A trigger that is marked **FOR EACH ROW** is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.
- Both the WHEN clause and the trigger actions may access elements of the row being inserted, deleted or updated using references of the form NEW.column-name and OLD.column-name, where column-name is the name of a column from the table that the trigger is associated with.
- If a WHEN clause is supplied, the PostgreSQL statements specified are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the PostgreSQL statements are executed for all rows.

- If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
- The BEFORE, AFTER or INSTEAD OF keyword determines when the trigger actions will be executed relative to the insertion, modification or removal of the associated row.
- Triggers are automatically dropped when the table that they are associated with is dropped.
- The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just tablename, not database.tablename.
- A CONSTRAINT option when specified creates a constraint trigger. This is the same as a regular trigger except that the timing of the trigger firing can be adjusted using SET CONSTRAINTS. Constraint triggers are expected to raise an exception when the constraints they implement are violated.



E.g

Let us consider a case where we want to keep audit trial for every record being inserted in COMPANY table, which we will create newly as follows (Drop COMPANY table if you already have it):

```
testdb=# CREATE TABLE COMPANY(  
  ID INT PRIMARY KEY      NOT NULL,  
  NAME          TEXT      NOT NULL,  
  AGE           INT       NOT NULL,  
  ADDRESS       CHAR(50),  
  SALARY        REAL  
);
```

To keep audit trial, we will create a new table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record:

```
testdb=# CREATE TABLE AUDIT(  
  EMP_ID INT NOT NULL,  
  ENTRY_DATE TEXT NOT NULL  
);
```


- ID is the AUDIT record ID, and EMP_ID is the ID which will come from COMPANY table and DATE will keep timestamp when the record will be created in COMPANY table.

let's create a trigger on COMPANY table as follows:

```
testdb=# CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY FOR  
EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

Where auditlogfunc() is a PostgreSQL procedure and has the following definition:

```
CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example_table$  
BEGIN  
    INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, current_timestamp);  
    RETURN NEW;  
END;  
$example_table$ LANGUAGE plpgsql;
```

Now, we will start actual work, let's start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. So let's create one record in COMPANY table as follows:

```
testdb=# INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

This will create one record in COMPANY table, which is as follows:

id	name	age	address	salary
1	Paul	32	California	20000

Same time, one record will be created in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation on COMPANY table.

Similar way you can create your triggers on UPDATE and DELETE operations based on your requirements.

emp_id	entry_date
1	2013-05-05 15:49:59.968+05:30

(1 row)

Listing TRIGGERS

You can list down all the triggers in the current database from pg_trigger table as follows:

```
testdb=# SELECT * FROM pg_trigger;
```

Above PostgreSQL statement will list down all triggers.

If you want to list down triggers on a particular table, then use AND clause with table name as follows:

```
testdb=# SELECT tgname FROM pg_trigger, pg_class WHERE  
tgrelid=pg_class.oid AND relname='company';
```

Above PostgreSQL statement will also list down only one entry as follows:

```
tgname  
-----  
example_trigger  
(1 row)
```

Dropping TRIGGERS

Following is the DROP command, which can be used to drop an existing trigger:

```
testdb=# DROP TRIGGER trigger_name;
```

INDEXES

Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval.

An index is a pointer to data in a table.

An index in a database is very similar to an index in the back of a book.

E.g

- if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers.

- An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.
- Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.
- Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

CREATE INDEX Command:

The basic syntax of CREATE INDEX is as follows:

```
CREATE INDEX index_name ON table_name;
```


Index Types

PostgreSQL provides several index types:

B-tree,

Hash,

GiST,

SP-GiST

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*

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

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

Multicolumn Indexes:

- A multicolumn index is defined **on more than one column of a table**. The basic syntax is as follows:

```
CREATE INDEX index_name ON table_name (column1_name, column2_name);
```

- Whether to create a single-column index or a multicolumn index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.
- Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the multicolumn index would be the best choice.

Unique Indexes:

Unique indexes are used for performance and also data integrity.

A unique index does not allow any duplicate values to be inserted into the table.

The basic syntax is as follows:

```
CREATE UNIQUE INDEX index_name on table_name (column_name);
```

Partial Indexes

A partial index is an **index built over a subset of a table**; the subset is defined by a conditional expression (called the predicate of the partial index).

The index contains entries only for those table rows that satisfy the predicate.

The basic syntax is as follows:

```
CREATE INDEX index_name on table_name (conditional_expression);
```

Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created.

Indexes are automatically created for primary key constraints and unique constraints.

- Following is an example where we will create an index on COMPANY table for salary column:

```
# CREATE INDEX salary_index ON COMPANY (salary);
```

- Now, let's list down all the indices available on COMPANY table using \d company command as follows:

```
# \d company
```

- This will produce the following result, where company_pkey is an implicit index which got created when the table was created.

Table "public.company"		
Column	Type	Modifiers
id	integer	not null
name	text	not null
age	integer	not null
address	character(50)	
salary	real	
Indexes:		
"company_pkey" PRIMARY KEY, btree (id)		
"salary_index" btree (salary)		

You can list down the entire indexes database wide using the \di command:

DROP INDEX Command

An index can be dropped using PostgreSQL DROP command. Care should be taken when dropping an index because performance may be slowed or improved.

- The basic syntax is as follows:

```
DROP INDEX index_name;
```

- You can use following statement to delete previously created index:

```
# DROP INDEX salary_index;
```


When should indexes be avoided?

The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

Alter table

ALTER TABLE command is used to add, delete or modify columns in an existing table.

You would also use ALTER TABLE command to add and drop various constraints on an existing table.

The basic syntax of ALTER TABLE to add a new column in an existing table is as follows:

```
ALTER TABLE table_name ADD column_name datatype;
```

The basic syntax of ALTER TABLE to DROP COLUMN in an existing table is as follows:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

The basic syntax of ALTER TABLE to change the DATA TYPE of a column in a table is as follows:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

- The basic syntax of ALTER TABLE to add a NOT NULL constraint to a column in a table is as follows:

ALTER TABLE table_name MODIFY column_name datatype NOT NULL;

- The basic syntax of ALTER TABLE to ADD UNIQUE CONSTRAINT to a table is as follows:

ALTER TABLE table_name ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);

- The basic syntax of ALTER TABLE to ADD CHECK CONSTRAINT to a table is as follows:

ALTER TABLE table_name ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);

The basic syntax of ALTER TABLE to ADD PRIMARY KEY constraint to a table is as follows:

```
ALTER TABLE table_name ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
```

The basic syntax of ALTER TABLE to DROP CONSTRAINT from a table is as follows:

```
ALTER TABLE table_name DROP CONSTRAINT MyUniqueConstraint;
```

The basic syntax of ALTER TABLE to DROP PRIMARY KEY constraint from a table is as follows:

```
ALTER TABLE table_name DROP CONSTRAINT MyPrimaryKey;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

Following is the example to ADD a new column in an existing table:

testdb=# ALTER TABLE COMPANY ADD GENDER char(1);

Now, COMPANY table is changed and following would be output from SELECT statement:

id	name	age	address	salary	gender
1	Paul	32	California	20000	
2	Allen	25	Texas	15000	
3	Teddy	23	Norway	20000	
4	Mark	25	Rich-Mond	65000	
5	David	27	Texas	85000	
6	Kim	22	South-Hall	45000	
7	James	24	Houston	10000	

(7 rows)

- **testdb=# ALTER TABLE COMPANY DROP GENDER;**

Now, COMPANY table is changed and following would be output from SELECT statement:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

Truncate table

TRUNCATE TABLE command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure from the database and you would need to re-create this table once again if you wish to store some data.

It has the same effect as an DELETE on each table, but since it does not actually scan the tables, it is faster.

It reclaims disk space immediately, rather than requiring a subsequent VACUUM operation.

This is most useful on large tables.

Consider COMPANY table is having the following records:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Following is the example to truncate:

```
testdb=# TRUNCATE TABLE COMPANY;
```

Now, COMPANY table is truncated and following would be output from SELECT statement:

```
testdb=# SELECT * FROM CUSTOMERS;  
id | name | age | address | salary  
----+-----+-----+-----+-----  
(0 rows)
```


Views

Views are pseudo-tables. That is, they are not real tables, but nevertheless appear as ordinary tables to SELECT.

A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table.

A view can even represent joined tables. Because views are assigned separate permissions, you can use them to restrict table access so that users see only specific rows or columns of a table.

A view can contain all rows of a table or selected rows from one or more tables.

A view can be created from one or many tables which depends on the written PostgreSQL query to create a view.

Views, - virtual tables, allow users to do the following:

Structure data in a way that users or classes of users find natural or intuitive.

Restrict access to the data such that a user can only see limited data instead of complete table.

Summarize data from various tables which can be used to generate reports.

Because views are not ordinary tables, so you may not execute a DELETE, INSERT, or UPDATE statement on a view. But you can create a RULE to correct this problem of using DELETE, INSERT or UPDATE on a view.

Creating Views

PostgreSQL views are created using the CREATE VIEW statement.

The PostgreSQL views can be created from a single table, multiple tables, or another view.

CREATE VIEW syntax is as follows:

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS  
SELECT column1, column2.....  
FROM table_name  
WHERE [condition];
```

We can include multiple tables in our SELECT statement in very similar way as we use them in normal PostgreSQL SELECT query.

If the optional TEMP or TEMPORARY keyword is present, the view will be created in the temporary space.

Temporary views are automatically dropped at the end of the current session.

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

Now, following is an example to create a view from COMPANY table.

This view would be used to have only few columns from COMPANY table:

```
testdb=# CREATE VIEW COMPANY_VIEW AS SELECT ID, NAME, AGE  
        FROM COMPANY;
```

Now, we can query COMPANY_VIEW in similar way as we query an actual table. Following is the example:

```
testdb=# SELECT * FROM COMPANY_VIEW;
```

id	name	age
1	Paul	32
2	Allen	25
3	Teddy	23
4	Mark	25
5	David	27
6	Kim	22
7	James	24

(7 rows)

Dropping Views

To drop a view, simply use the DROP VIEW statement with the view_name.

The basic DROP VIEW syntax is as follows:

```
testdb=# DROP VIEW view_name;
```

Following command will delete COMPANY_VIEW view, which we created in last section:

```
testdb=# DROP VIEW COMPANY_VIEW;
```

Materialized Views

Materialized views in PostgreSQL use the rule system like views do, but persist the results in a table like form.

Difference:

Materialized view cannot subsequently be directly updated and that the query used to create the materialized view is stored in exactly the same way that a view's query is stored, so that fresh data can be generated for the materialized view with:

- **REFRESH MATERIALIZED VIEW rental_by_category;**

Where to Use -

Materialized views are useful in many cases that require fast data access therefore they are often used in data warehouses or business intelligent applications.

E.g

```
CREATE MATERIALIZED VIEW rental_by_category
AS
SELECT c.name AS category,
       sum(p.amount) AS total_sales
FROM (((((payment p
        JOIN rental r ON ((p.rental_id = r.rental_id)))
        JOIN inventory i ON ((r.inventory_id = i.inventory_id)))
        JOIN film f ON ((i.film_id = f.film_id)))
        JOIN film_category fc ON ((f.film_id = fc.film_id)))
        JOIN category c ON ((fc.category_id = c.category_id)))
GROUP BY c.name
ORDER BY sum(p.amount) DESC
WITH NO DATA;
```


Because we used the WITH NO DATA option, we cannot query data from the view. If we try to do so, we will get an error message as follows:

```
dvdrental=# select * from rental_by_category;
```

```
ERROR: materialized view "rental_by_category" has not been  
populated
```

HINT: Use the REFRESH MATERIALIZED VIEW command.

PostgreSQL is very nice to give us a hint to ask for loading data into the view. Let's do it by executing the following statement:

REFRESH MATERIALIZED VIEW rental_by_category;

```
dvdrental=# select * from rental_by_category;
```

category	total_sales
Sports	4892.19
Sci-Fi	4336.01
Animation	4245.31
Drama	4118.46
Comedy	4002.48
New	3966.38
Action	3951.84
Foreign	3934.47
Games	3922.18
Family	3830.15
Documentary	3749.65
Horror	3401.27
Classics	3353.38
Children	3309.39
Travel	3227.36
Music	3071.52

(16 rows)

From now on, we can refresh the data in the rental_by_category view using the REFRESH MATERIALIZED VIEW statement.

However, to refresh it with CONCURRENTLY option, we need to create a UNIQUE index for the view first.

```
CREATE UNIQUE INDEX rental_category ON rental_by_category (category);
```

Let's refresh data concurrently for the rental_by_category view.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY rental_by_category;
```

Transactions



Transactions

A transaction is a unit of work that is performed against a database.

Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database.

E.g

- if you are creating a record or updating a record or deleting a record from the table, then you are performing transaction on the table. It is important to control transactions to ensure data integrity and to handle database errors.

Practically, you will club many PostgreSQL queries into a group and you will execute all of them together as a part of a transaction.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID:

- **Atomicity**: ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
- **Consistency**: ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation**: enables transactions to operate independently of and transparent to each other.
- **Durability**: ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control

There are following commands used to control transactions:

- **BEGIN TRANSACTION**: to start a transaction.
- **COMMIT**: to save the changes, alternatively you can use **END TRANSACTION** command.
- **ROLLBACK**: to rollback the changes.
- Transactional control commands are only used with the DML commands **INSERT**, **UPDATE** and **DELETE** only. They can not be used while creating tables or dropping them because these operations are automatically committed in the database.

BEGIN TRANSACTION Command

Transactions can be started using BEGIN TRANSACTION or simply BEGIN command.

Such transactions usually persist until the next COMMIT or ROLLBACK command is encountered.

But a transaction will also ROLLBACK if the database is closed or if an error occurs.

Following is the simple syntax to start a transaction:

BEGIN;

or

BEGIN TRANSACTION;

COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows:

COMMIT;

or

END TRANSACTION;

ROLLBACK Command

*ROLLBACK command is the transactional command used to **undo** transactions that have not already been saved to the database.*

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for ROLLBACK command is as follows:

ROLLBACK;

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

Now, let's start a transaction and delete records from the table having age = 25 and finally we use ROLLBACK command to undo all the changes.

```
testdb=# BEGIN;  
DELETE FROM COMPANY WHERE AGE = 25;  
ROLLBACK;
```

If you will check COMPANY table is still having the following records:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

Now, let's start another transaction and delete records from the table having age = 25 and finally we use COMMIT command to commit all the changes.

```
testdb=# BEGIN;
```

```
DELETE FROM COMPANY WHERE AGE = 25;
```

```
COMMIT;
```

If you will check COMPANY table is still having the following records:

id	name	age	address	salary
1	Paul	32	California	20000
3	Teddy	23	Norway	20000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
(5 rows)				

Locks

- Locks or Exclusive Locks or Write Locks prevent users from modifying a row or an entire table.
- Rows modified by UPDATE and DELETE are then exclusively locked automatically for the duration of the transaction.
- **This prevents other users from changing the row until the transaction is either committed or rolled back.**
- **The only time when users must wait for other users is when they are trying to modify the same row. If they modify different rows, no waiting is necessary. SELECT queries never have to wait.**
- The database performs locking automatically.
- In certain cases, however, locking must be controlled manually. Manual locking can be done by using the LOCK command. It allows specification of a transaction's lock type and scope.

Syntax for LOCK command is as follows:

LOCK [TABLE]

name

IN

lock_mode

name: The name (optionally schema-qualified) of an existing table to lock. If ONLY is specified before the table name, only that table is locked. If ONLY is not specified, the table and all its descendant tables (if any) are locked.

lock_mode: The lock mode specifies which locks this lock conflicts with. If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive mode, is used.

Possible values are: ACCESS SHARE, ROW SHARE , ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE.

Note: Once obtained, the lock is held for the remainder of the current transaction. There is no UNLOCK TABLE command; locks are always released at transaction end.

DeadLocks

Deadlocks can occur when two transactions are waiting for each other to finish their operations. While PostgreSQL can detect them and end them with a ROLLBACK, deadlocks can still be inconvenient.

To prevent your applications from running into this problem, make sure to design them in such a way that they will lock objects in the same order.

Advisory Locks

PostgreSQL provides a means for creating locks that have application-defined meanings. These are called advisory locks.

As the system does not enforce their use, it is up to the application to use them correctly.

Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model.

E.g

A common use of advisory locks is to emulate pessimistic locking strategies typical of so called "flat file" data management systems. While a flag stored in a table could be used for the same purpose, advisory locks are faster, avoid table bloat, and are automatically cleaned up by the server at the end of the session.

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

The following example locks the COMPANY table within the testdb database in ACCESS EXCLUSIVE mode. The LOCK statement works only in a transaction mode:

```
testdb=#BEGIN;  
LOCK TABLE company1 IN ACCESS EXCLUSIVE MODE;
```

Above PostgreSQL statement will produce the following result:

LOCK TABLE

The above message indicates that the table is locked until the transaction ends and to finish the transaction you will have to either rollback or commit the transaction.

Sub Queries

A subquery or Inner query or Nested query is a query within another PostgreSQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE and DELETE statements along with the operators like =, <, >, >=, <=, IN, etc.

Rules of Sub queries

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN, EXISTS, NOT IN, ANY/SOME, ALL operator.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN can be used within the subquery.

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Now, let us check following sub-query with SELECT statement:

```
testdb=# SELECT *
        FROM COMPANY
        WHERE ID IN (SELECT ID
                    FROM COMPANY
                    WHERE SALARY > 45000) ;
```

id	name	age	address	salary
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000

(2 rows)

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date, or number functions. The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
```

```
    SELECT [ *|column1 [, column2 ]
```

```
    FROM table1 [, table2 ]
```

```
    [ WHERE VALUE OPERATOR ]
```

Example:

Consider a table COMPANY_BKP with similar structure as COMPANY table and can be created using same CREATE TABLE using COMPANY_BKP as table name.

Now to copy complete COMPANY table into COMPANY_BKP, following is the syntax:

```
testdb=# INSERT INTO COMPANY_BKP
        SELECT * FROM COMPANY
        WHERE ID IN (SELECT ID
                     FROM COMPANY);
```



Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

UPDATE table

SET column_name = new_value

[WHERE OPERATOR [VALUE]

(SELECT COLUMN_NAME

FROM TABLE_NAME)

[WHERE)]

Following example updates SALARY by 0.50 times in COMPANY table for all the customers, whose AGE is greater than or equal to 27:

```
testdb=# UPDATE COMPANY
```

```
SET SALARY = SALARY * 0.50
```

```
WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
```

```
WHERE AGE >= 27 );
```

This would impact two rows and finally COMPANY table would have the following records:

id	name	age	address	salary
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
1	Paul	32	California	10000
5	David	27	Texas	42500

(7 rows)

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME  
[ WHERE OPERATOR [ VALUE ]  
(SELECT COLUMN_NAME  
FROM TABLE_NAME)  
[ WHERE) ]
```

Assuming, we have COMPANY_BKP table available which is backup of COMPANY table.

Following example deletes records from COMPANY table for all the customers, whose AGE is greater than or equal to 27:

```
testdb=# DELETE FROM COMPANY
```

```
WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
```

```
WHERE AGE > 27 );
```

This would impact two rows and finally COMPANY table would have the following records:

id	name	age	address	salary
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000
5	David	27	Texas	42500

(6 rows)

AUTO INCREMENT

PostgreSQL has the data types `smallserial`, `serial` and `bigserial`; these are not true types, but merely a notational convenience for creating unique identifier columns.

These are similar to `AUTO_INCREMENT` property supported by some other databases.

If you wish a serial column to have a unique constraint or be a primary key, it must now be specified, just like any other data type.

The type name `serial` create integer columns. The type name `bigserial` create a `bigint` column. `bigserial` should be used if you anticipate the use of more than 231 identifiers over the lifetime of the table. The type name `smallserial` create a `smallint` column.

Consider COMPANY table to be created as follows:

```
testdb=# CREATE TABLE COMPANY(  
  ID SERIAL PRIMARY KEY,  
  NAME          TEXT      NOT NULL,  
  AGE           INT       NOT NULL,  
  ADDRESS       CHAR(50),  
  SALARY        REAL  
);
```

Now, insert following records into table COMPANY:

This will insert 7 tuples into the table COMPANY and COMPANY will have the following records:

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

```
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'Paul', 32, 'California', 20000.00 );  
  
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'Allen', 25, 'Texas', 15000.00 );  
  
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'Teddy', 23, 'Norway', 20000.00 );  
  
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'Mark', 25, 'Rich-Mond ', 65000.00 );  
  
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'David', 27, 'Texas', 85000.00 );  
  
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'Kim', 22, 'South-Hall', 45000.00 );  
  
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)  
VALUES ( 'James', 24, 'Houston', 10000.00 );
```

PRIVILEGES

The owner is usually the one who executed the creation statement.

For most kinds of objects, the initial state is that only the owner (or a superuser) can modify or delete the object.

To allow other roles or users to use it, privileges or permission must be granted.

Different kinds of privileges in PostgreSQL are:

**SELECT, INSERT,
UPDATE, DELETE,
TRUNCATE, REFERENCES,
TRIGGER, CREATE,
CONNECT, TEMPORARY,
EXECUTE, USAGE**

```
testdb=# CREATE USER ranger WITH PASSWORD 'ilg007';
```

CREATE ROLE

The message CREATE ROLE indicates that the USER "ranger" is created.

Consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Next, let's grants all privileges on a table COMPANY to the user "ranger" as follows:

```
testdb=# GRANT ALL ON COMPANY TO ranger;
```

```
GRANT
```

The message GRANT indicates that all privileges are assigned to the USER.

Next, let's revoke the privileges from the USER "ranger" as follows:

```
testdb=# REVOKE ALL ON COMPANY FROM ranger;
```

```
REVOKE
```

The message REVOKE indicates that all privileges are revoked from the USER.

You can even delete the user as follows:

```
testdb=# DROP USER ranger;
```

```
DROP ROLE
```

The message DROP ROLE indicates USER ranger is deleted from the database.







DATE/TIME Functions and Operators

The following table lists the behaviors of the basic arithmetic 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'

Operator	Example	Result
-	- 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'

Operator	Example	Result
*	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'

Function	Description
AGE() 	Subtract arguments
CURRENT DATE/TIME() 	Current date and time
DATE_PART() 	Get subfield (equivalent to extract)
EXTRACT() 	Get subfield
ISFINITE() 	Test for finite date,time and interval (not +/-infinity)
JUSTIFY 	Adjust interval

AGE(timestamp, timestamp), AGE(timestamp)

Function	Description
AGE(timestamp, timestamp)	When invoked with the <code>TIMESTAMP</code> form of the second argument, <code>AGE()</code> subtract arguments, producing a "symbolic" result that uses years and months and is of type <code>INTERVAL</code> .
AGE(timestamp)	When invoked with only the <code>TIMESTAMP</code> as argument, <code>AGE()</code> subtracts from the <code>current_date</code> (at midnight).

Example for function AGE(timestamp, timestamp) is:

```
testdb=# SELECT AGE(timestamp '2015-09-27', timestamp '1957-06-13');
```

Above PostgreSQL statement will produce the following result:

```
      age
-----
58 years 3 mons 14 days
```

Example for function AGE(timestamp) is:

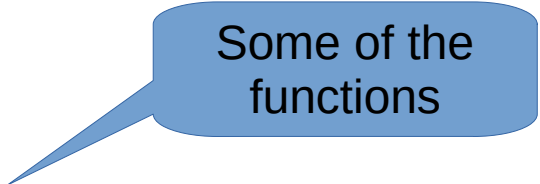
```
testdb=# select age(timestamp '1973-06-13');
```

Above PostgreSQL statement will produce the following result:

```
      age
-----
42 years 3 mons 14 days
```

CURRENT DATE/TIME()

Function	Description
CURRENT_DATE	Delivers current date.
CURRENT_TIME	Delivers values with time zone.
CURRENT_TIMESTAMP	Delivers values with time zone.
CURRENT_TIME(precision)	Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.
CURRENT_TIMESTAMP(precision)	Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.
LOCALTIME	Delivers values without time zone.
LOCALTIMESTAMP	Delivers values without time zone.
LOCALTIME(precision)	Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.
LOCALTIMESTAMP(precision)	Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.



Some of the functions

```
postgres=# select current_time;  
          timetz
```

```
-----  
06:26:29.269939+05:30  
(1 row)
```

```
postgres=# select current_date;  
          date
```

```
-----  
2015-09-27  
(1 row)
```

```
postgres=# select localtimestamp;  
          timestamp
```

```
-----  
2015-09-27 06:30:57.390938  
(1 row)
```

```
postgres=# select current_timestamp;  
          now
```

```
-----  
2015-09-27 06:28:09.815444+05:30  
(1 row)
```

```
postgres=# select current_timestamp(2);  
          timestampz
```

```
-----  
2015-09-27 06:29:22.64+05:30  
(1 row)
```

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. These functions are:

Function	Description
<code>transaction_timestamp()</code>	It is equivalent to <code>CURRENT_TIMESTAMP</code> , but is named to clearly reflect what it returns.
<code>statement_timestamp()</code>	It returns the start time of the current statement.
<code>clock_timestamp()</code>	It returns the actual current time, and therefore its value changes even within a single SQL command.
<code>timeofday()</code>	It returns the actual current time, but as a formatted text string rather than a timestamp with time zone value.
<code>now()</code>	It is a traditional PostgreSQL equivalent to <code>transaction_timestamp()</code> .

Functions

PostgreSQL functions, also known as Stored Procedures, allow you to carry out operations that would normally take several queries and round trips in a single function within the database.

Functions allow database reuse as other applications can interact directly with your stored procedures instead of a middle-tier or duplicating code.

Functions can be created in language of your choice like SQL, PL/pgSQL, C, Python, ...

```
CREATE [OR REPLACE] FUNCTION
function_name (arguments)
RETURNS return_datatype AS
$variable_name$
  DECLARE
    declaration;
  [...]
BEGIN
  < function_body >
  [...]
  RETURN { variable_name | value }
END; LANGUAGE plpgsql;
```



Usual syntax

Example

The following example illustrates creating and calling a standalone function. This function returns the total number of records in the COMPANY table. We will use the COMPANY table, which has the following records:

```
testdb=# create or replace function totalrecords() returns int as $total$ declare
testdb$$ total integer;
testdb$$ begin
testdb$$ select count(*) into total from company;
testdb$$ return total;
testdb$$ end;
testdb$$ $total$ language plpgsql;
CREATE FUNCTION
testdb=# select totalrecords();
totalrecords
-----
          7
(1 row)
```

Useful Functions - inbuilt

- PostgreSQL COUNT function is the simplest function and very useful in counting the number of records, which are expected to be returned by a SELECT statement.
- To understand COUNT function consider the table COMPANY having records as follows:

```
testdb=# SELECT COUNT(*) FROM COMPANY;
```

```
testdb=# SELECT COUNT(*) FROM COMPANY WHERE name='Paul';
```



count the number of records for Paul

MAX function

PostgreSQL MAX function is used to find out the record with maximum value among a record set.

To understand MAX function, consider the table COMPANY having records as follows:

suppose based on the above table you want to fetch maximum value of SALARY, then we can do so simply using the following command:

```
testdb=# SELECT MAX(salary) FROM COMPANY;
```

We can find all the records with maximum value for each name using GROUP BY clause as follows:

```
testdb=# SELECT id, name, MAX(salary) FROM COMPANY GROUP BY id, name;
```

We can use MIN Function along with MAX function to find out minimum value as well. Try out following example:

```
testdb=# SELECT MIN(salary), MAX(salary) max FROM company;
```

MIN Function

PostgreSQL MIN function is used to find out the record with minimum value among a record set.

To understand MIN function consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000

(7 rows)

Now suppose based on the above table you want to fetch minimum value of salary, then you can do so simply using the following command:

```
testdb=# SELECT MIN(salary) FROM company;
```

You can find all the records with minimum value for each name using GROUP BY clause as follows:

```
testdb=# SELECT id, name, MIN(salary) FROM company GROUP BY id, name;
```

AVG Function

To understand AVG function consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
id | name  | age | address  | salary
---+---+---+---+---
1  | Paul  | 32  | California | 20000
2  | Allen | 25  | Texas     | 15000
3  | Teddy | 23  | Norway    | 20000
4  | Mark  | 25  | Rich-Mond | 65000
5  | David | 27  | Texas     | 85000
6  | Kim   | 22  | South-Hall | 45000
7  | James | 24  | Houston   | 10000
(7 rows)
```

Now suppose based on the above table we want to calculate average of all the SALARY, then we can do so by using the following command:

```
testdb=# SELECT AVG(SALARY) FROM COMPANY;
```

```
testdb=# SELECT name, AVG(SALARY) FROM COMPANY GROUP BY name;
```

Sum Function

PostgreSQL SUM function is used to find out the sum of a field in various records.

To understand SUM function consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```

id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000











(7 rows)












Now suppose based on the above table we want to calculate total of all the salary, then we can do so by using the following command:




```
testdb# SELECT SUM(salary) FROM company;
```

```
testdb# SELECT name, SUM(salary) FROM company GROUP BY name;
```

Numeric functions

Name	Description
ABS() 	Returns the absolute value of numeric expression.
ACOS() 	Returns the arccosine of numeric expression. Returns NULL if the value is not in the range -1 to 1.
ASIN() 	Returns the arcsine of numeric expression. Returns NULL if value is not in the range -1 to 1
ATAN() 	Returns the arctangent of numeric expression.
ATAN2() 	Returns the arctangent of the two variables passed to it.
CEIL() 	Returns the smallest integer value that is not less than passed numeric expression
CEILING() 	Returns the smallest integer value that is not less than passed numeric expression
COS() 	Returns the cosine of passed numeric expression. The numeric expression should be expressed in radians.
COT() 	Returns the cotangent of passed numeric expression.
DEGREES() 	Returns numeric expression converted from radians to degrees.

Name	Description
EXP() 	Returns the base of the natural logarithm (e) raised to the power of passed numeric expression.
FLOOR() 	Returns the largest integer value that is not greater than passed numeric expression.
GREATEST() 	Returns the largest value of the input expressions.
LEAST() 	Returns the minimum-valued input when given two or more.
LOG() 	Returns the natural logarithm of the passed numeric expression.
MOD() 	Returns the remainder of one expression by dividing by another expression.
PI() 	Returns the value of pi
POW() 	Returns the value of one expression raised to the power of another expression
POWER() 	Returns the value of one expression raised to the power of another expression
RADIANS() 	Returns the value of passed expression converted from degrees to radians.
ROUND() 	Returns numeric expression rounded to an integer. Can be used to round an expression to a number of decimal points

Name	Description
SIN() 	Returns the sine of numeric expression given in radians.
SQRT() 	Returns the non-negative square root of numeric expression.
TAN() 	Returns the tangent of numeric expression expressed in radians.

Array_Agg function

- PostgreSQL ARRAY_AGG function is used to concatenate the input values including null into an array.
- To understand ARRAY_AGG function, consider the table COMPANY having records as follows:

```
testdb# select * from COMPANY;
```















id	name	age	address	salary
1	Paul	32	California	20000
2	Allen	25	Texas	15000
3	Teddy	23	Norway	20000
4	Mark	25	Rich-Mond	65000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
7	James	24	Houston	10000













(7 rows)

Now suppose based on the above table we want to use the ARRAY_AGG, we can do so by using the following command:

```
testdb=# SELECT ARRAY_AGG(SALARY) FROM COMPANY;
```

String Functions

Name	Description
ASCII() 	Returns numeric value of left-most character
BIT_LENGTH() 	Returns length of argument in bits
CHAR_LENGTH() 	Returns number of characters in argument
CHARACTER_LENGTH() 	A synonym for CHAR_LENGTH()
CONCAT_WS() 	Returns concatenate with separator
CONCAT() 	Returns concatenated string
LCASE() 	Synonym for LOWER()
LEFT() 	Returns the leftmost number of characters as specified
LENGTH() 	Returns the length of a string in bytes
LOWER() 	Returns the argument in lowercase
LPAD() 	Returns the string argument, left-padded with the specified string
LTRIM() 	Removes leading spaces
MID() 	Returns a substring starting from the specified position
POSITION() 	A synonym for LOCATE()

Name	Description
QUOTE() 	Escapes the argument for use in an SQL statement
REGEXP 	Pattern matching using regular expressions
REPEAT() 	Repeats a string the specified number of times
REPLACE() 	Replaces occurrences of a specified string
REVERSE() 	Reverse the characters in a string
RIGHT() 	Returns the specified rightmost number of characters
RPAD() 	Appends string the specified number of times
RTRIM() 	Removes trailing spaces
SUBSTRING(), SUBSTR() 	Returns the substring as specified
TRIM() 	Removes leading and trailing spaces
UCASE() 	Synonym for UPPER()
UPPER() 	Converts to uppercase

WINDOW FUNCTIONS

A window function performs a calculation across a set of table rows that are somehow related to the current row.

Usage:

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.

employees

last_name	salary	department
Jones	45000	Accounting
Adams	50000	Sales
Johnson	40000	Marketing
Williams	37000	Accounting
Smith	55000	Sales



Highest paid
person

First we can rank each individual over a certain grouping:

```
SELECT last_name,  
       salary,  
       department,  
       rank() OVER (  
         PARTITION BY department  
         ORDER BY salary  
         DESC  
       )  
FROM employees
```

last_name	salary	department	rank
Jones	45000	Accounting	1
Williams	37000	Accounting	2
Smith	55000	Sales	1
Adams	50000	Sales	2
Johnson	40000	Marketing	1


```
SELECT *
FROM (
  SELECT
    last_name,
    salary,
    department,
    rank() OVER (
      PARTITION BY department
      ORDER BY salary
      DESC
    )
  FROM employees) sub_query
WHERE rank = 1;
```

its clear from here how we can filter and find only the top paid employee in each department:

last_name	salary	department	rank
Jones	45000	Accounting	1
Smith	55000	Sales	1
Johnson	40000	Marketing	1

<https://www.postgresql.org/docs/9.1/static/functions-window.html>

<https://www.postgresql.org/docs/9.1/static/tutorial-window.html>

References

Postgresql manuals

Reference examples manuals and websites.

