POSTGRESQL - QUICK GUIDE

POSTGRESQL OVERVIEW

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.

This tutorial will give you quick start with PostgreSQL and make you comfortable with PostgreSQL programming.

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.

Key features of PostgreSQL

PostgreSQL runs on all major operating systems, including Linux, UNIX AIX, BSD, HP – UX, SGIIRIX, MacOS, 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

You can check official documentation of PostgreSQL to understand above-mentioned features.

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.
To start understanding the PostgreSQL basics, first let's install the PostgreSQL. This chapter explains about installing the PostgreSQL on Linux, Windows and Mac OS platforms.

## 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](https://www.enterprisedb.com/). I downloaded `postgresql-9.2.4-1-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  ]
```

If your installation was correct, you will have PostgreSQL prompt `test=#` shown above.

## Installing PostgreSQL on Windows

Follow the following steps to install PostgreSQL on your Windows machine. Make sure you have turned Third Party Antivirus off while installing.

- Pick the version number of PostgreSQL you want and, as exactly as possible, the platform you
I download postgresql-9.2.4-1-windows.exe for my Windows PC running in 32 bit mode, so let's run `postgresql-9.2.4-1-windows.exe` as administrator to install PostgreSQL. Select the location where you want to install it. By default it is installed within Program Files folder.

The next step of the installation process would be to select the directory where data would be stored, by default it is stored under "data" directory.

The next step, setup asks for password, so you can use your favorite password.
• The next step, keep the port as default

• The next step, when asked for "Locale", I have selected "English, United States".

• It takes a while to install PostgreSQL on your system. On completion of the installation process, you will get the following screen. Uncheck the checkbox and click on Finish button.
After the installation process is completed, you can access pgAdmin III, StackBuilder and PostgreSQL shell from your Program Menu under PostgreSQL 9.2.

**Installing PostgreSQL on Mac**

Follow the following steps to install PostgreSQL on your Mac machine. Make sure you are logged in as **administrator** before you proceed for the installation.

- Pick the latest version number of PostgreSQL for Mac OS available at [EnterpriseDB](http://www.enterprisedb.com).

- I downloaded `postgresql-9.2.4-1-osx.dmg` for my Mac OS running with OS X version 10.8.3. Now, let's open the dmg image in finder and just double click it which will give you PostgreSQL installer in the following window:

![PostgreSQL Installer](image)

- Next, click on `postgres-9.2.4-1-osx` icon which will give a warning message, just accept the warning and proceed for further installation. It will ask for administrator password using the following window:

![PostgreSQL Installer Warning](image)

Enter the password and proceed for the installation and after this step, restart your Mac.
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 instal PostgreSQL at your Mac machine in Application folder which you can check:

Now, you can launch any of the program to start with. Let's start with SQL Shell. When you launch SQL Shell, just use all the default values it displays except, enter your password which you had selected at the time of installation. If everything goes fine, then you will be inside postgres database and you will be given a `postgres#` prompt as shown below:
Here is a list of the PostgreSQL SQL commands, followed by the precise syntax rules for each of these commands. This set of commands is taken from the psql command-line tool. Now that you have Postgres installed, open the psql as:

Program Files > PostgreSQL 9.2 > SQL Shell > psql.

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>
```

### The 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.

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Description</th>
<th>SELECT</th>
<th>id, name</th>
<th>FROM</th>
<th>states</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>Command</td>
<td>id</td>
<td>name</td>
<td>FROM</td>
<td>states</td>
</tr>
<tr>
<td>Identifier</td>
<td>Id and name</td>
<td>columns</td>
<td></td>
<td>Keyword Identifier</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>Clause</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special symbol</td>
<td>Table name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PostgreSQL SQL commands

#### ABORT

Abort the current transaction.

```
ABORT [ WORK | TRANSACTION ]
```

#### ALTER AGGREGATE

Change the definition of an aggregate function.

```
ALTER AGGREGATE name ( type ) RENAME TO new_name
ALTER AGGREGATE name ( type ) OWNER TO new_owner
```
**ALTER CONVERSION**

Change the definition of a conversion.

- `ALTER CONVERSION name RENAME TO new_name`
- `ALTER CONVERSION name OWNER TO new_owner`

**ALTER DATABASE**

Change a database specific parameter.

- `ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }`
- `ALTER DATABASE name RESET parameter`
- `ALTER DATABASE name RENAME TO new_name`
- `ALTER DATABASE name OWNER TO new_owner`

**ALTER DOMAIN**

Change the definition of a domain specific parameter.

- `ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }`
- ` ALTER DOMAIN name { SET | DROP } NOT NULL`
- `ALTER DOMAIN name ADD domain_constraint`
- `ALTER DOMAIN name DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]`
- `ALTER DOMAIN name OWNER TO new_owner`

**ALTER FUNCTION**

Change the definition of a function.

- `ALTER FUNCTION name ( [ type [, ...] ] ) RENAME TO new_name`
- `ALTER FUNCTION name ( [ type [, ...] ] ) OWNER TO new_owner`

**ALTER GROUP**

Change a user group.

- `ALTER GROUP groupname ADD USER username [, ...]`
- `ALTER GROUP groupname DROP USER username [, ...]`
- `ALTER GROUP groupname RENAME TO new_name`

**ALTER INDEX**

Change the definition of an index.

- `ALTER INDEX name OWNER TO new_owner`
- `ALTER INDEX name SET TABLESPACE indexspace_name`
- `ALTER INDEX name RENAME TO new_name`

**ALTER LANGUAGE**

Change the definition of a procedural language.

- `ALTER LANGUAGE name RENAME TO new_name`

**ALTER OPERATOR**

Change the definition of an operator.
ALTER OPERATOR name ([ { lefttype | NONE }, { righttype | NONE } ])
OWNER TO new_owner

**ALTER OPERATOR CLASS**

Change the definition of an operator class.

ALTER OPERATOR CLASS name USING index_method RENAME TO new_name
ALTER OPERATOR CLASS name USING index_method OWNER TO new_owner

**ALTER SCHEMA**

Change the definition of a schema.

ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO new_owner

**ALTER SEQUENCE**

Change the definition of a sequence generator.

ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ]
[ MAXVALUE maxvalue | NO MAXVALUE ]
[ RESTART [ WITH ] start ] [ CACHE cache ] [ NO CYCLE ]

**ALTER TABLE**

Change the definition of a table.

ALTER TABLE [ ONLY ] name [ * ] action [, ... ]
ALTER TABLE [ ONLY ] name [ * ]
RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
RENAME TO new_name

Where *action* is one of the following lines:

ADD [ COLUMN ] column_type [ column_constraint [ ... ] ]
DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
ALTER [ COLUMN ] column TYPE type [ USING expression ]
ALTER [ COLUMN ] column SET DEFAULT expression
ALTER [ COLUMN ] column DROP DEFAULT
ALTER [ COLUMN ] column { SET | DROP } NOT NULL
ALTER [ COLUMN ] column SET STATISTICS integer
ALTER [ COLUMN ] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ADD table_constraint
DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET WITHOUT OIDS
OWNER TO new_owner
SET TABLESPACE tablespace_name

**ALTER TABLESPACE**

Change the definition of a tablespace.

ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO new_owner
ALTER TRIGGER
Change the definition of a trigger.

ALTER TRIGGER name ON table RENAME TO new_name

ALTER TYPE
Change the definition of a type.

ALTER TYPE name OWNER TO new_owner

ALTER USER
Change a database user account.

ALTER USER name [ [ WITH ] option [ ... ] ]
ALTER USER name RENAME TO new_name
ALTER USER name SET parameter { TO | = } { value | DEFAULT }
ALTER USER name RESET parameter

Where option can be:

[ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| VALID UNTIL 'abstime'

ANALYZE
Collect statistics about a database.

ANALYZE [ VERBOSE ] [ table [ (column [, ... ] ) ] ]

BEGIN
Start a transaction block.

BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ... ] ]

Where transaction_mode is one of:

ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
| READ UNCOMMITTED }
READ WRITE | READ ONLY

CHECKPOINT
Force a transaction log checkpoint.

CHECKPOINT

CLOSE
Close a cursor.

CLOSE name

CLUSTER
Cluster a table according to an index.

CLUSTER index_name ON table_name
CLUSTER table_name
CLUSTER

COMMENT

Define or change the comment of an object.

COMMENT ON
{ TABLE object_name | COLUMN table_name.column_name | AGGREGATE agg_name (agg_type) | CAST (source_type AS target_type) | CONSTRAINT constraint_name ON table_name | CONVERSION object_name | DATABASE object_name | DOMAIN object_name | FUNCTION func_name (arg1_type, arg2_type, ...) | INDEX object_name | LARGE OBJECT large_object_oid | OPERATOR op (left_operand_type, right_operand_type) | OPERATOR CLASS object_name USING index_method | PROCEDURAL LANGUAGE object_name | RULE rule_name ON table_name | SCHEMA object_name | SEQUENCE object_name | TRIGGER trigger_name ON table_name | TYPE object_name | VIEW object_name }
IS 'text'

COMMIT

Commit the current transaction.

COMMIT [ WORK | TRANSACTION ]

COPY

Copy data between a file and a table.

COPY table_name [ ( column [, ... ] ) ]
FROM { 'filename' | STDIN }
[ [ WITH ]
[ BINARY ]
[ OIDS ]
[ DELIMITER [ AS ] 'delimiter' ]
[ NULL [ AS ] 'null string' ]
[ CSV [QUOTE [AS ] 'quote' ]
[ ESCAPE [AS ] 'escape' ]
[ FORCE NOT NULL column [, ...] ]
COPY table_name [ ( column [, ... ] ) ]
TO { 'filename' | STDOUT }
[ [ WITH ]
[ BINARY ]
[ OIDS ]
[ DELIMITER [ AS ] 'delimiter' ]
[ NULL [ AS ] 'null string' ]
[ CSV [QUOTE [AS ] 'quote' ]
[ ESCAPE [AS ] 'escape' ]
[ FORCE QUOTE column [, ...] ]

CREATE AGGREGATE

Define a new aggregate function.

```
CREATE AGGREGATE name (
    BASETYPE = input_data_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [, FINALFUNC = ffunc ]
    [, INITCOND = initial_condition ]
)
```

CREATE CAST

Define a new cast.

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION func_name (arg_types)
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

CREATE CONSTRAINT TRIGGER

Define a new constraint trigger.

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    table_name constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func_name (args)
```

CREATE CONVERSION

Define a new conversion.

```
CREATE [DEFAULT] CONVERSION name
    FOR source_encoding TO dest_encoding FROM func_name
```

CREATE DATABASE

Create a new database.

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] db_owner ]
    [ TEMPLATE [=] template ]
    [ ENCODING [=] encoding ]
    [ TABLESPACE [=] tablespace ] ]
```

CREATE DOMAIN

Define a new domain.

```
CREATE DOMAIN name [AS] data_type
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

Where constraint is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```
CREATE FUNCTION

Define a new function.

CREATE [ OR REPLACE ] FUNCTION name ( [ [ arg_name ] arg_type [, ... ] ] )
RETURNS ret_type
{ LANGUAGE lang_name
 | IMMUTABLE | STABLE | VOLATILE
 | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
 | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
 | AS 'definition'
 | AS 'obj_file', 'link_symbol'
}...
[ WITH ( attribute [, ... ] ) ]

CREATE GROUP

Define a new user group.

CREATE GROUP name [ [ WITH ] option [ ... ] ]
Where option can be:
SYSID gid
| USER username [, ...]

CREATE INDEX

Define a new index.

CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
{ { column | ( expression ) } [ opclass ] [, ... ] }
[ TABLESPACE tablespace ]
[ WHERE predicate ]

CREATE LANGUAGE

Define a new procedural language.

CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
HANDLER call_handler [ VALIDATOR val_function ]

CREATE OPERATOR

Define a new operator.

CREATE OPERATOR name ( PROCEDURE = func_name
[, LEFTARG = left_type ] [, RIGHTARG = right_type ]
[, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
[, RESTRICT = res_proc ] [, JOIN = join_proc ]
[, HASHES ] [, MERGES ]
[, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
[, LTCMP = less_than_op ] [, GTCMP = greater_than_op ]
)

CREATE OPERATOR CLASS

Define a new operator class.

CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
USING index_method AS
{ OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ RECHECK ]

CREATE RULE

Define a new rewrite rule.

CREATE [ OR REPLACE ] RULE name AS ON event
TO table [ WHERE condition ]
DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }

CREATE SCHEMA

Define a new schema.

CREATE SCHEMA schema_name
[ AUTHORIZATION username ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION username
[ schema_element [ ... ] ]

CREATE SEQUENCE

Define a new sequence generator.

CREATE [ TEMPORARY | TEMP ] SEQUENCE name
[ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ]
[ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ] [ NO CYCLE ]

CREATE TABLE

Define a new table.

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
table_constraint
LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] [ , ... ]
)
[ INHERITS ( parent_table [ , ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]

Where column_constraint is:

[ CONSTRAINT constraint_name ]
[ NOT NULL ]
[ NULL ]
[ UNIQUE [ USING INDEX TABLESPACE tablespace ] ]
[ PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] ]
[ CHECK ( expression ) ]
[ REFERENCES ref_table [ ( ref_column ) ] ]
[ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
[ ON DELETE action ] [ ON UPDATE action ]
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

And table_constraint is:

[ CONSTRAINT constraint_name ]
[ UNIQUE ( column_name [ , ... ] ) [ USING INDEX TABLESPACE tablespace ] ]
[ PRIMARY KEY ( column_name [ , ... ] ) [ USING INDEX TABLESPACE tablespace ] ]
[ CHECK ( expression ) ]
CREATE TABLE AS

Define a new table from the results of a query.

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
[(column_name [, ... ] )] [ [ WITH | WITHOUT ] OIDS ]
AS query

CREATE TABLESPACE

Define a new tablespace.

CREATE TABLESPACE tablespace_name [ OWNER username ] LOCATION 'directory'

CREATE TRIGGER

Define a new trigger.

CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE func_name ( arguments )

CREATE TYPE

Define a new data type.

CREATE TYPE name AS
( attribute_name data_type [, ... ] )
CREATE TYPE name (
INPUT = input_function,
OUTPUT = output_function
[ , RECEIVE = receive_function ]
[ , SEND = send_function ]
[ , ANALYZE = analyze_function ]
[ , INTERNALLENGTH = { internal_length | VARIABLE } ]
[ , PASSEDBYVALUE ]
[ , ALIGNMENT = alignment ]
[ , STORAGE = storage ]
[ , DEFAULT = default ]
[ , ELEMENT = element ]
[ , DELIMITER = delimiter ]
)

CREATE USER

Define a new database user account.

CREATE USER name [ [ WITH ] option [ ... ] ]

Where option can be:

SYSID uid
| [ [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| IN GROUP group_name [, ...]
<table>
<thead>
<tr>
<th>CREATE VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define a new view.</td>
</tr>
</tbody>
</table>
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ] AS query |

<table>
<thead>
<tr>
<th>DEALLOCATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEALLOCATE [ PREPARE ] plan_name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DECLARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE</td>
</tr>
</tbody>
</table>
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] |
CURSOR [ { WITH | WITHOUT } HOLD ] FOR query |
[ FOR { READ ONLY | UPDATE [ OF column [, ...] ] } ] |

<table>
<thead>
<tr>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
</tr>
</tbody>
</table>
DELETE FROM [ ONLY ] table [ WHERE condition ] |

<table>
<thead>
<tr>
<th>DROP AGGREGATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP AGGREGATE</td>
</tr>
</tbody>
</table>
DROP AGGREGATE name ( type ) [ CASCADE | RESTRICT ] |

<table>
<thead>
<tr>
<th>DROP CAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP CAST</td>
</tr>
</tbody>
</table>
DROP CAST (source_type AS target_type) [ CASCADE | RESTRICT ] |

<table>
<thead>
<tr>
<th>DROP CONVERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP CONVERSION</td>
</tr>
</tbody>
</table>
DROP CONVERSION name [ CASCADE | RESTRICT ] |

<table>
<thead>
<tr>
<th>DROP DATABASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP DATABASE</td>
</tr>
</tbody>
</table>
DROP DATABASE name |

<table>
<thead>
<tr>
<th>DROP DOMAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP DOMAIN</td>
</tr>
</tbody>
</table>
DROP DOMAIN name [, ...] [ CASCADE | RESTRICT ] |
DROP FUNCTION
Remove a function.

DROP FUNCTION name ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]

DROP GROUP
Remove a user group.

DROP GROUP name

DROP INDEX
Remove an index.

DROP INDEX name [, ...] [ CASCADE | RESTRICT ]

DROP LANGUAGE
Remove a procedural language.

DROP [ PROCEDURAL ] LANGUAGE name [ CASCADE | RESTRICT ]

DROP OPERATOR
Remove an operator.

DROP OPERATOR name ( { left_type | NONE }, { right_type | NONE } )
[ CASCADE | RESTRICT ]

DROP OPERATOR CLASS
Remove an operator class.

DROP OPERATOR CLASS name USING index_method [ CASCADE | RESTRICT ]

DROP RULE
Remove a rewrite rule.

DROP RULE name ON relation [ CASCADE | RESTRICT ]

DROP SCHEMA
Remove a schema.

DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]

DROP SEQUENCE
Remove a sequence.

DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]

DROP TABLE
Remove a table.

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

**DROP TABLESPACE**

Remove a tablespace.

```
DROP TABLESPACE tablespace_name
```

**DROP TRIGGER**

Remove a trigger.

```
DROP TRIGGER name ON table [ CASCADE | RESTRICT ]
```

**DROP TYPE**

Remove a data type.

```
DROP TYPE name [, ...] [ CASCADE | RESTRICT ]
```

**DROP USER**

Remove a database user account.

```
DROP USER name
```

**DROP VIEW**

Remove a view.

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

**END**

Commit the current transaction.

```
END [ WORK | TRANSACTION ]
```

**EXECUTE**

Execute a prepared statement.

```
EXECUTE plan_name [ (parameter [, ... ] ) ]
```

**EXPLAIN**

Show the execution plan of a statement.

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

**FETCH**

Retrieve rows from a query using a cursor.

```
FETCH [ direction { FROM | IN } ] cursor_name
```
Where `direction` can be empty or one of:

- NEXT
- PRIOR
- FIRST
- LAST
- ABSOLUTE count
- RELATIVE count
- count
- ALL
- FORWARD
- FORWARD count
- FORWARD ALL
- BACKWARD
- BACKWARD count
- BACKWARD ALL

**GRANT**

Define access privileges.

```sql
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER } [,...] | ALL [ PRIVILEGES ] } ON [ TABLE ] table_name [, ...] TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] } ON DATABASE db_name [, ...] TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] } ON TABLESPACE tablespace_name [, ...] TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] } ON FUNCTION func_name ([type, ...]) [, ...] TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] } ON LANGUAGE lang_name [, ...] TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,,...] | ALL [ PRIVILEGES ] } ON SCHEMA schema_name [, ...] TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

**INSERT**

Create new rows in a table.

```sql
INSERT INTO table [ ( column [, ...] ) ] { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | query }
```

**LISTEN**

Listen for a notification.

```sql
LISTEN name
```

**LOAD**

Load or reload a shared library file.

```sql
LOAD 'filename'
```
LOCK

Lock a table.

LOCK [ TABLE ] name [, ...] [ IN lock_mode MODE ] [ NOWAIT ]

Where lock_mode is one of:

ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE

MOVE

Position a cursor.

MOVE [ direction { FROM | IN } ] cursor_name

NOTIFY

Generate a notification.

NOTIFY name

PREPARE

Prepare a statement for execution.

PREPARE plan_name [ (data_type [, ...] ) ] AS statement

REINDEX

Rebuild indexes.

REINDEX { DATABASE | TABLE | INDEX } name [ FORCE ]

RELEASE SAVEPOINT

Destroy a previously defined savepoint.

RELEASE [ SAVEPOINT ] savepoint_name

RESET

Restore the value of a runtime parameter to the default value.

RESET name
RESET ALL

REVOKE

Remove access privileges.

REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER } 
[ , ... ] | ALL [ PRIVILEGES ] } 
ON [ TABLE ] table_name [, ...] 
FROM { username | GROUP group_name | PUBLIC } [, ...] 
[ CASCADE | RESTRICT ]
REVOKE [ GRANT OPTION FOR ]
{ { CREATE | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
ON DATABASE db_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ CREATE | ALL [ PRIVILEGES ] }
ON TABLESPACE tablespace_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ EXECUTE | ALL [ PRIVILEGES ] }
ON FUNCTION func_name ([type, ...]) [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
ON LANGUAGE lang_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

ROLLBACK
Abort the current transaction.

ROLLBACK [ WORK | TRANSACTION ]

ROLLBACK TO SAVEPOINT
Roll back to a savepoint.

ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name

SAVEPOINT
Define a new savepoint within the current transaction.

SAVEPOINT savepoint_name

SELECT
Retrieve rows from a table or view.

SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
* | expression [ AS output_name ] [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
[ FOR UPDATE [ OF table_name [, ...] ] ]
SELECT INTO

Define a new table from the results of a query.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ] * | expression [ AS output_name ] [, ...]
INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
FROM from_item [, ...]
WHERE condition
GROUP BY expression [, ...]
HAVING condition [, ...]
{ UNION | INTERSECT | EXCEPT } [ ALL ] select
ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
LIMIT { count | ALL }
OFFSET start
FOR UPDATE [ OF table_name [, ...] ]
```

SET

Change a runtime parameter.

```
SET [ SESSION | LOCAL ] name { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { time_zone | LOCAL | DEFAULT }
```

SET CONSTRAINTS

Set constraint checking modes for the current transaction.

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

SET SESSION AUTHORIZATION

Set the session user identifier and the current user identifier of the current session.

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

SET TRANSACTION

Set the characteristics of the current transaction.

```
SET TRANSACTION transaction_mode [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

Where `transaction_mode` is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
```

SHOW
Show the value of a runtime parameter.

```
SHOW name
SHOW ALL
```

**START TRANSACTION**

Start a transaction block.

```
START TRANSACTION [ transaction_mode [, ... ] ]
```

Where `transaction_mode` is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
| READ UNCOMMITTED }
READ WRITE | READ ONLY
```

**TRUNCATE**

Empty a table.

```
TRUNCATE [ TABLE ] name
```

**UNLISTEN**

Stop listening for a notification.

```
UNLISTEN { name | * }
```

**UPDATE**

Update rows of a table.

```
UPDATE [ ONLY ] table SET column = { expression | DEFAULT } [, ...]
[ FROM from_list ]
[ WHERE condition ]
```

**VACUUM**

Garbage-collect and optionally analyze a database.

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

**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
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. They are discussed as below:

### 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. Table below lists the available types.

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

### 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>money</td>
<td>8 bytes</td>
<td>currency amount</td>
<td>-92233720368547758.08 to +92233720368547758.07</td>
</tr>
</tbody>
</table>

### Character Types

The table below lists general-purpose character types available in PostgreSQL.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>character varying(n), varchar(n)</td>
<td>variable-length with limit</td>
</tr>
<tr>
<td>character(n), char(n)</td>
<td>fixed-length, blank padded</td>
</tr>
<tr>
<td>text</td>
<td>variable unlimited length</td>
</tr>
</tbody>
</table>

**Binary Data Types**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytea</td>
<td>1 or 4 bytes plus the actual binary string</td>
<td>variable-length binary string</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Low Value</th>
<th>High Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp ([p]) [without time zone ]</td>
<td>8 bytes</td>
<td>both date and time \ notimezone</td>
<td>4713 BC</td>
<td>294276 AD</td>
</tr>
<tr>
<td>timestamp ([p]) with time zone</td>
<td>8 bytes</td>
<td>both date and time, with time zone</td>
<td>4713 BC</td>
<td>294276 AD</td>
</tr>
<tr>
<td>date</td>
<td>4 bytes</td>
<td>date \ notimeofday</td>
<td>4713 BC</td>
<td>5874897 AD</td>
</tr>
<tr>
<td>time ([p]) [without time zone ]</td>
<td>8 bytes</td>
<td>time of day \ nodate</td>
<td>00:00:00</td>
<td>24:00:00</td>
</tr>
<tr>
<td>time ([p]) with time zone</td>
<td>12 bytes</td>
<td>times of day only, with time zone</td>
<td>00:00:00+1459</td>
<td>24:00:00-1459</td>
</tr>
<tr>
<td>interval ([fields]) ([p])</td>
<td>12 bytes</td>
<td>time interval</td>
<td>-178000000 years</td>
<td>178000000 years</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 byte</td>
<td>state of true or false</td>
</tr>
</tbody>
</table>

**Enumerated 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>16 bytes</td>
<td>Point on a plane</td>
<td>x, y</td>
</tr>
<tr>
<td>line</td>
<td>32 bytes</td>
<td>Infinite line notfullyimplemented</td>
<td>(x1, y1, x2, y2)</td>
</tr>
<tr>
<td>lseg</td>
<td>32 bytes</td>
<td>Finite line segment</td>
<td>(x1, y1, x2, y2)</td>
</tr>
<tr>
<td>box</td>
<td>32 bytes</td>
<td>Rectangular box</td>
<td>(x1, y1, x2, y2)</td>
</tr>
<tr>
<td>path</td>
<td>16+16n bytes</td>
<td>Closed path similartopolygon</td>
<td>(x1, y1,...)</td>
</tr>
<tr>
<td>path</td>
<td>16+16n bytes</td>
<td>Open path</td>
<td>[x1, y1,...]</td>
</tr>
<tr>
<td>polygon</td>
<td>40+16n</td>
<td>Polygon similartoclosedpath</td>
<td>(x1, y1,...)</td>
</tr>
<tr>
<td>circle</td>
<td>24 bytes</td>
<td>Circle</td>
<td>&lt;x, y, r&gt; centerpointradius</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cidr</td>
<td>7 or 19 bytes</td>
<td>IPv4 and IPv6 networks</td>
</tr>
<tr>
<td>inet</td>
<td>7 or 19 bytes</td>
<td>IPv4 and IPv6 hosts and networks</td>
</tr>
<tr>
<td>macaddr</td>
<td>6 bytes</td>
<td>MAC addresses</td>
</tr>
</tbody>
</table>

**Bit String Type**

Bit String Types are used to store bit masks. They are either 0 or 1. There are two SQL bit types: `bitn` and `bit varyingn`, 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsvector</td>
<td>This is a sorted list of distinct words that have been normalized to merge different variants of the same word, called as &quot;lexemes&quot;.</td>
</tr>
</tbody>
</table>
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 UniversallyUniqueIdentifiers 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:

```sql
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 JavaScriptObjectNotation 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:

<table>
<thead>
<tr>
<th>Example</th>
<th>Example Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_to_json</td>
<td>'{1,5,99,100}'</td>
</tr>
<tr>
<td>row_to_json</td>
<td>{'f1':1,'f2':'foo'}</td>
</tr>
</tbody>
</table>

Array Type

PostgreSQL gives opportunity to define a column of a table as a variable length multidimensional array. 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:

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

or by using keyword "ARRAY" as:

```sql
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:

```sql
INSERT INTO monthly_savings
VALUES
    ('Manisha',
    '{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.

```sql
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.

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

or using the ARRAY expression syntax:

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

Searching Arrays

An example of searching arrays is as shown below.

```sql
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.

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

Composite Types

This type represents a list of field names and their data types, i.e., structure of a row or record of a table.

Declaration of Composite Types

The following example shows how to declare a composite type:

```sql
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
);

**Composite Value Input**

Composite values can be inserted as a literal constant, enclosing the field values within parentheses and separating them by commas. An example is as below:

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

This is valid for the `inventory_item` defined above. The ROW keyword is actually optional as long as you have more than one field in the expression.

**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. For example, 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., `allintegervalues1to10` or continuous ranges e.g., `anypointintimebetween10:00amand11:00am`.

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.

**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. The following table lists several alias types. The OID alias types have no operations of their own except for specialized input and output routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>References</th>
<th>Description</th>
<th>Value Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>any</td>
<td>numeric object identifier</td>
<td>564182</td>
</tr>
</tbody>
</table>
Pseudo Types

The PostgreSQL type system contains a number of special-purpose entries that are collectively called pseudo-types. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type. The table below lists the existing pseudo-types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>Indicates that a function accepts any input data type.</td>
</tr>
<tr>
<td>anyelement</td>
<td>Indicates that a function accepts any data type.</td>
</tr>
<tr>
<td>anyarray</td>
<td>Indicates that a function accepts any array data type.</td>
</tr>
<tr>
<td>anynonarray</td>
<td>Indicates that a function accepts any non-array data type.</td>
</tr>
<tr>
<td>anyenum</td>
<td>Indicates that a function accepts any enum data type.</td>
</tr>
<tr>
<td>anyrange</td>
<td>Indicates that a function accepts any range data type.</td>
</tr>
<tr>
<td>cstring</td>
<td>Indicates that a function accepts or returns a null-terminated C string.</td>
</tr>
<tr>
<td>internal</td>
<td>Indicates that a function accepts or returns a server-internal data type.</td>
</tr>
<tr>
<td>language_handler</td>
<td>A procedural language call handler is declared to return language_handler.</td>
</tr>
<tr>
<td>fdw_handler</td>
<td>A foreign-data wrapper handler is declared to return fdw_handler.</td>
</tr>
<tr>
<td>record</td>
<td>Identifies a function returning an unspecified row type.</td>
</tr>
<tr>
<td>trigger</td>
<td>A trigger function is declared to return trigger.</td>
</tr>
<tr>
<td>void</td>
<td>Indicates that a function returns no value.</td>
</tr>
</tbody>
</table>

POSTGRESQL CREATE DATABASE

This chapter discusses about how to create a new database in your PostgreSQL. PostgreSQL provides two ways of creating a new 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 \textit{template1}.

\textbf{Syntax}

The basic syntax of \texttt{CREATE DATABASE} statement is as follows:

\begin{verbatim}
CREATE DATABASE dbname;
\end{verbatim}

where \textit{dbname} is the name of a database to create.

\textbf{Example}

Following is a simple example, which will create \texttt{testdb} in your PostgreSQL schema:

\begin{verbatim}
postgres=# CREATE DATABASE testdb;
postgres-
\end{verbatim}

\textbf{Using \texttt{createdb} Command}

PostgreSQL command line executable \texttt{createdb} is a wrapper around the SQL command \texttt{CREATE DATABASE}. The only difference between this command and SQL command \texttt{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.

\textbf{Syntax}

The syntax for \texttt{createdb} is as shown below:

\begin{verbatim}
createdb [option...] [dbname [description]]
\end{verbatim}

\textbf{Parameters}

Table below lists the parameters with their descriptions.

\begin{table}[h]
\centering
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbname</td>
<td>The name of a database to create.</td>
</tr>
<tr>
<td>description</td>
<td>Specifies a comment to be associated with the newly created database.</td>
</tr>
<tr>
<td>options</td>
<td>command-line arguments, which \texttt{createdb} accepts.</td>
</tr>
</tbody>
</table>
\end{table}

\textbf{Options}

The following table lists the command-line arguments \texttt{createdb} accepts:

\begin{table}[h]
\centering
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D tablespace</td>
<td>Specifies the default tablespace for the database.</td>
</tr>
<tr>
<td>-e</td>
<td>Echo the commands that \texttt{createdb} generates and sends to the server.</td>
</tr>
<tr>
<td>-E encoding</td>
<td>Specifies the character encoding scheme to be used in this database.</td>
</tr>
<tr>
<td>-l locale</td>
<td>Specifies the locale to be used in this database.</td>
</tr>
<tr>
<td>-T template</td>
<td>Specifies the template database from which to build this database.</td>
</tr>
<tr>
<td>--help</td>
<td>Show help about \texttt{createdb} command line arguments, and exit.</td>
</tr>
</tbody>
</table>
\end{table}
-h host            Specifies the host name of the machine on which the server is running.
-p port            Specifies the TCP port or the local Unix domain socket file extension on
                   which the server is listening for connections.
-U username        User name to connect as.
-w                 Never issue a password prompt.
-W                 Force createdb to prompt for a password before connecting to a
                   database.

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.

Once a database is created using either of the above-mentioned methods, you can check it in the
list of databases using \l, i.e., backslash el command as follows:

```
postgres=# \l
        List of databases
    Name | Owner   | Encoding | Collate | Ctype | Access privileges
   -----+---------+----------+---------+-------+-----------------------
 postgres| postgres| UTF8     | C       | C     | =c/postgres
 template0| postgres| UTF8     | C       | C     | =c/postgres=CTc/postgres
 template1| postgres| UTF8     | C       | C     | =c/postgres=CTc/postgres
 testdb  | postgres| UTF8     | C       | C     | (4 rows)

postgres=#
```

**POSTGRESQL - SELECT DATABASE**

This chapter explains various methods of accessing the database. Assume that we have already
created a database in our previous chapter. You can select database using either of the following
methods:

- Database SQL Prompt
- 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 el command as follows:

```
postgres=# \l
        List of databases
    Name | Owner   | Encoding | Collate | Ctype | Access privileges
   -----+---------+----------+---------+-------+-----------------------
 postgres| postgres| UTF8     | C       | C     | =c/postgres
 template0| postgres| UTF8     | C       | C     | =c/postgres=CTc/postgres
 template1| postgres| UTF8     | C       | C     | =c/postgres=CTc/postgres
 testdb  | postgres| UTF8     | C       | C     | (4 rows)
```
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 postgress testdb
Password for user postgress: ****
psql (9.2.4)
Type "help" for help.
You are now connected to database "testdb" as user "postgress".
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 chapter we will discuss how to delete the database in PostgreSQL. They are two options to delete a database:

1. Using DROP DATABASE, an SQL command.
2. 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.

**Using DROP DATABASE**

This command drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. This command cannot be executed while you or anyone else is connected to the target database connecttopostgresoranyotherdatabasetoissuethiscommand.

**Syntax**

The syntax for DROP DATABASE is as below:

```sql
DROP DATABASE [ IF EXISTS ] name
```

**Parameters**

Table below lists the parameters with their descriptions.
**Parameter** | **Description**  
--- | ---  
IF EXISTS | Do not throw an error if the database does not exist. A notice is issued in this case.  
name | The name of the database to remove.  

*We cannot drop a database that has any open connections, including our own connection from psql or pgAdmin III. We must switch to another database or template1 if we want to delete the database we are currently connected to. Thus, it might be more convenient to use the program dropdb instead which is a wrapper around this command.*

**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.

**Syntax**

The syntax for **createdb** is as shown below:

```
dropdb [option...] dbname
```

**Parameters**

Table below lists the parameters with their descriptions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbname</td>
<td>The name of a database to be deleted.</td>
</tr>
<tr>
<td>option</td>
<td>command-line arguments, which dropdb accepts.</td>
</tr>
</tbody>
</table>

**Options**

The following table lists the command-line arguments dropdb accepts:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e</td>
<td>Shows the commands being sent to the server.</td>
</tr>
<tr>
<td>-i</td>
<td>Issues a verification prompt before doing anything destructive.</td>
</tr>
<tr>
<td>-V</td>
<td>Print the dropdb version and exit.</td>
</tr>
<tr>
<td>--if-exists</td>
<td>Do not throw an error if the database does not exist. A notice is issued in this case.</td>
</tr>
</tbody>
</table>
--help  Show help about dropdb command-line arguments, and exit.
-h host  Specifies the host name of the machine on which the server is running.
-p port  Specifies the TCP port or the local UNIX domain socket file extension on which the server is listening for connections.
-U username  User name to connect as.
-w  Never issue a password prompt.
-W  Force dropdb to prompt for a password before connecting to a database.
--maintenance-db=dbname  Specifies the name of the database to connect to in order to drop the target 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 foundundertheyp,olesoftemplate! user name to drop the database.

**POSTGRESQL - CREATE TABLE**

The PostgreSQL CREATE TABLE statement is used to create a new table in any of the given database.

**Syntax**

Basic syntax of CREATE TABLE statement is as follows:

```sql
CREATE TABLE table_name(
  column1 datatype,
  column2 datatype,
  column3 datatype,
  ....
  columnN datatype,
  PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Initially empty table in the current database and will be owned by the user issuing the command.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example below.

**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:

```sql
CREATE TABLE COMPANY(
  ID INT PRIMARY KEY NOT NULL,
  NAME TEXT NOT NULL,
  AGE INT NOT NULL,
  ADDRESS CHAR(50),
  SALARY REAL
);
```
Let us create one more table, which we will use in our exercises in subsequent chapters:

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

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.

```bash
testdb-# \d
```

Above PostgreSQL statement will produce the following result:

```
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:

```bash
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.

**Syntax**

Basic syntax of DROP TABLE statement is as follows.

```sql
DROP TABLE table_name;
```

**Example**

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

```bash
testdb-# \d
```
This would produce the following result:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>company</td>
<td>table</td>
<td>postgres</td>
</tr>
<tr>
<td>public</td>
<td>department</td>
<td>table</td>
<td>postgres</td>
</tr>
</tbody>
</table>

(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.

**Syntax**

The basic syntax CREATE SCHEMA is as follows:

```sql
CREATE SCHEMA name;
```

Where *name* is the name of the schema.

**Syntax to Create table in Schema**

The basic syntax to create table in schema is as follows:

```sql
CREATE TABLE myschema.mytable (...
);
```

**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,
);```
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)
```

**Syntax to Drop schema**

To drop a schema if it's empty *allobjectsinithavebeen dropped*, then use:

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

**Advantages of using a Schema**

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

**POSTGRESQL - INSERT QUERY**

The 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.

**Syntax**

Basic syntax of **INSERT INTO** statement is as follows.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

- Here, column1, column2,...columnN are the names of the columns in the table into which you want to insert data.
- The target column names can be listed in any order. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.

You may not need to specify the columns name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQL INSERT INTO syntax would be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

**Output**

The following table summarizes the output messages and their meaning:
### Output

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT oid 1</td>
<td>Message returned if only one row was inserted. oid is the numeric OID of the inserted row.</td>
</tr>
<tr>
<td>INSERT 0 #</td>
<td>Message returned if more than one rows were inserted. # is the number of rows inserted.</td>
</tr>
</tbody>
</table>

### Examples

Let us create COMPANY table in **testdb** as follows:

```sql
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:

```sql
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:

```sql
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:

```sql
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:

```sql
```

All the above statements would create the following records in COMPANY table. Next chapter will teach you how to display all these records from a table.

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>AGE</th>
<th>ADDRESS</th>
<th>SALARY</th>
<th>JOIN_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paul</td>
<td>32</td>
<td>California</td>
<td>20000.0</td>
<td>2001-07-13</td>
</tr>
<tr>
<td>2</td>
<td>Allen</td>
<td>25</td>
<td>Texas</td>
<td>20000.0</td>
<td>2007-12-13</td>
</tr>
<tr>
<td>3</td>
<td>Teddy</td>
<td>23</td>
<td>Norway</td>
<td>65000.0</td>
<td>2007-12-13</td>
</tr>
<tr>
<td>4</td>
<td>Mark</td>
<td>25</td>
<td>Rich-Mond</td>
<td>85000.0</td>
<td>2007-12-13</td>
</tr>
<tr>
<td>5</td>
<td>David</td>
<td>27</td>
<td>Texas</td>
<td>85000.0</td>
<td>2007-12-13</td>
</tr>
</tbody>
</table>

### 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.
Syntax:
The basic syntax of SELECT statement is as follows:

```sql
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2...are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the table then you can use the following syntax:

```sql
SELECT * FROM table_name;
```

Example:
Consider the table `COMPANY` having records as follows:

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

(7 rows)

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

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

This would produce the following result:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paul</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td>Allen</td>
<td>15000</td>
</tr>
<tr>
<td>3</td>
<td>Teddy</td>
<td>20000</td>
</tr>
<tr>
<td>4</td>
<td>Mark</td>
<td>65000</td>
</tr>
<tr>
<td>5</td>
<td>David</td>
<td>85000</td>
</tr>
<tr>
<td>6</td>
<td>Kim</td>
<td>45000</td>
</tr>
<tr>
<td>7</td>
<td>James</td>
<td>10000</td>
</tr>
</tbody>
</table>

(7 rows)

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

```sql
testdb=# SELECT * FROM COMPANY;
```

This would produce the following result:

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

(7 rows)
The 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., which we would examine in subsequent chapters.

**Syntax**

The basic syntax of SELECT statement with WHERE clause is as follows:

```sql
SELECT column1, column2, columnN
FROM table_name
WHERE [search_condition]
```

You can specify a `search_condition` using comparison or logical operators, like >, <, =, LIKE, NOT, etc. Below examples would make this concept clear.

**Example:**

Consider the table `COMPANY` having records as follows:

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

(7 rows)

Here are 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:

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

Above PostgreSQL statement will produce the following result:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Mark</td>
<td>25</td>
<td>Rich-Mond</td>
<td>65000</td>
</tr>
<tr>
<td>5</td>
<td>David</td>
<td>27</td>
<td>Texas</td>
<td>85000</td>
</tr>
</tbody>
</table>

(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:

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

Above PostgreSQL statement will produce the following result:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paul</td>
<td>32</td>
<td>California</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td>Allen</td>
<td>25</td>
<td>Texas</td>
<td>15000</td>
</tr>
<tr>
<td>4</td>
<td>Mark</td>
<td>25</td>
<td>Rich-Mond</td>
<td>65000</td>
</tr>
<tr>
<td>5</td>
<td>David</td>
<td>27</td>
<td>Texas</td>
<td>85000</td>
</tr>
</tbody>
</table>
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:

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

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:

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paul</td>
<td>32</td>
<td>California</td>
<td>20000</td>
</tr>
</tbody>
</table>
```

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:

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Allen</td>
<td>25</td>
<td>Texas</td>
<td>15000</td>
</tr>
<tr>
<td>4</td>
<td>Mark</td>
<td>25</td>
<td>Rich-Mond</td>
<td>65000</td>
</tr>
<tr>
<td>5</td>
<td>David</td>
<td>27</td>
<td>Texas</td>
<td>85000</td>
</tr>
</tbody>
</table>
```

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:

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paul</td>
<td>32</td>
<td>California</td>
<td>20000</td>
</tr>
<tr>
<td>3</td>
<td>Teddy</td>
<td>23</td>
<td>Norway</td>
<td>20000</td>
</tr>
<tr>
<td>6</td>
<td>Kim</td>
<td>22</td>
<td>South-Hall</td>
<td>45000</td>
</tr>
<tr>
<td>7</td>
<td>James</td>
<td>24</td>
<td>Houston</td>
<td>10000</td>
</tr>
</tbody>
</table>
```

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;
```

(4 rows)
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);

Above PostgreSQL statement will produce the following result:

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>24</td>
</tr>
</tbody>
</table>

(7 rows)

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**

The 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.

**Syntax:**

The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2,..., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

**Example:**

Consider the table **COMPANY** having records as follows:
testdb# select * from COMPANY;

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

(7 rows)

Following is an example, which would update ADDRESS for a customer, whose ID is 6:

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

Now, COMPANY table would have the following records:

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

(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;

Now, COMPANY table will have the following records:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paul</td>
<td>32</td>
<td>Texas</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td>Allen</td>
<td>25</td>
<td>Texas</td>
<td>20000</td>
</tr>
<tr>
<td>4</td>
<td>Mark</td>
<td>25</td>
<td>Texas</td>
<td>20000</td>
</tr>
<tr>
<td>5</td>
<td>David</td>
<td>27</td>
<td>Texas</td>
<td>20000</td>
</tr>
<tr>
<td>6</td>
<td>Kim</td>
<td>22</td>
<td>Texas</td>
<td>20000</td>
</tr>
<tr>
<td>7</td>
<td>James</td>
<td>24</td>
<td>Texas</td>
<td>20000</td>
</tr>
<tr>
<td>3</td>
<td>Teddy</td>
<td>23</td>
<td>Texas</td>
<td>20000</td>
</tr>
</tbody>
</table>

(7 rows)

POSTGRESQL - DELETE QUERY

The 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.

**Syntax:**

The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

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

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

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

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

Now COMPANY table will have following records:

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

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:

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

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