

WORLD'S MOST ADVANCED OPEN SOURCE object-RELATIONAL DATABASE

Applications & Tools:

PGAdmin 4 (PostgreSQL GUI)

Dummy & Random Data Generator Tool: <https://www.mockaroo.com/>

<https://dbdiagram.io/home> //Data Model Designing

/? //for help in psql

\help <command_name>

Install Postgre on linux

1. sudo apt-get update
 2. sudo apt-get upgrade
 3. sudo apt-get install postgresql postgresql-contrib
-

Start PostgreSQL CLI

- service postgresql status //running status of PostgreSQL
 - sudo su postgres //login as postgres user
 - psql //start PostgreSQL cli in Terminal
 - \q //to exit PostgreSQL cli
-

In PostgreSQL CLI (psql //start PostgreSQL cli in Terminal)

- \l (small L) //list of databases
 - \du //list of users of PSQL DBMS
 - CREATE DATABASE test ;
 - DROP DATABASE test ;
 -
 - ALTER USER postgres PASSWORD 'admin'; \\ to alter password of a user
 - \c test //to connect to a database;
 -
-

Connect to database (after logging as sudo su postgres)

psql --help

Default Hostname: "/var/run/postgresql"

Default Port: "5432"

Default Username: "postgres"

~~**Default Password:**~~ I had set my postgresql password admin

CMD: psql -h localhost -p 5432 -U postgres DB_name

Exit: \q

or

\l

\c DB_name

In a Database (\c db_name //to connect to a database;)

- **CREATE TABLE** employee(
 id **BIGSERIAL NOT NULL PRIMARY KEY**,
 age **INT NOT NULL**,
 full_name **VARCHAR(60) NOT NULL**,
 gender **VARCHAR(7) NOT NULL**,
 dob **DATE NOT NULL**,
);
 BIGSERIAL == BIGINT it increment by themselves.
- \d employee; //structure of table
in pgadmin: Servers-> learn->db_name->Schemas->public->Tables
- **DROP TABLE** table_name;
- **INSERT INTO** table_name(col1,col2,...) **VALUES**(102, 'val2' ,) ;
INSERT INTO table_name **VALUES**(102, 'val2' ,, 'valn') ; //no need to mention col_name if we are inserting in all columns;
 EX: **INSERT INTO** person (first_name,last_name,gender, dob) **VALUES** ('Anne', 'Smith', DATE '1988-01-09') ;
- **SELECT * FROM** table_name;
SELECT col1, col2 **FROM** table_name;
SELECT DISTINCT col1, col2 **FROM** table_name;
- **SELECT DISTINCT** col1, col2 **FROM** table_name **WHERE** col1='fddfff';
- **AND || OR || ORDER BY** col_name | col1, col2 | **ASC** (default) | **DESC**
- **LIMIT** 9 | 2*3-1;
- **UPDATE** table_name **SET** col_name = 'new updated val' **WHERE** col_name2 = 'val to search' ;
- **DELETE FROM** table_name **WHERE** col_name2 = 'val to search' ;
- **DROP TABLE** table_name; (erase the table from db + unrestoreable + no log is maintained)
- **TRUNCATE TABLE** table_name; (delete all the records of the data + log is maintained)
- **ALTER TABLE:**
 ADD NEW COL =>**ALTER TABLE** table_name **ADD** newcol_name datatype;
 DROP A COL =>**ALTER TABLE** table_name **DROP** col_name;
 MODIFY A COL=>**ALTER TABLE** table_name **MODIFY** col_name newdatatype;
- **WHERE** col_name **BETWEEN** val1 **AND** val2; ===== colname>=val1 **AND** colname<=val2 ;
- Comparison Operators: **Equal to (==): =** , **Not Equal to (!=): <>** , rest....is same
- **WHERE** col1 **IN** (val1, val2, val3);
- **WHERE** col1 **LIKE** 'p%' ; OR '_p' //LIKE is CASE SENSITIVE
WHERE col1 **ILIKE** 'p%' ; OR '_p' ==(LIKE 'P%' + LIKE 'p%') //ILIKE is CASE INSENSITIVE
- **GROUP BY:**
SELECT country, COUNT(*) **FROM** person **GROUP BY** country **ORDER BY** country;

- **HAVING:** (must be after GROUP BY and before ORDER BY)
SELECT country, COUNT(*) FROM person GROUP BY country **HAVING** COUNT(*) > 40 ORDER BY country;
- **AGGREGATORS : MIN, MAX, COUNT, etc**
MAX: SELECT **MAX**(price) FROM car;
Eg: SELECT make, **MAX**(price) FROM car GROUP BY make;

MIN: SELECT **MIN**(price) FROM car;
AVG: SELECT **AVG**(price) FROM car;

ROUND: SELECT **ROUND**(AVG(price)) FROM car;
EG: SELECT make, price, **ROUND**(price*.10 , 2) **AS** discount FROM car;
//it will show 10% price of cars upto 2 precision.

SUM: SELECT make, **SUM**(price) FROM car GROUP BY make;
- **Handling Null Values:**
SELECT email FROM person; //it will print values all records on email col and if it is NULL blank will be print.
SELECT **COALESCE**(email, "<Default Value>") FROM person; //it will print Default in place of NULL.
- **Handling Divide by 0 Error:**

```

ERROR: division by zero
test=# SELECT NULLIF(10, 10);
 nullif
-----
(1 row)


test=# SELECT NULLIF(10, 1);
 nullif
-----
      10
(1 row)

test=# SELECT NULLIF(10, 19);
 nullif
-----
      10
(1 row)

test=# SELECT NULLIF(100, 19);
 nullif
-----
     100
(1 row)

test=# SELECT NULLIF(100, 100);
 nullif
-----

```



```

      }
(1 row)

```

```
test=# SELECT COALESCE(10 / NULLIF(0, 0), 0);
 coalesce
-----
(1 row)

test=#
```

- **Timestamp and Date**

NOW()

```
test=# SELECT NOW();
          now
-----
2018-12-02 22:43:41.774892+00
(1 row)

test=# SELECT NOW()::DATE;
          now
-----
2018-12-02
(1 row)

test=# SELECT NOW()::TIME;
          now
-----
22:44:35.645348
(1 row)
```

INTERVAL

```
test=# SELECT (NOW() + INTERVAL '10 MONTHS')::DATE;
          date
-----
2019-10-02
(1 row)
```

AGE()

```
test=# SELECT first_name, last_name, gender, country_of_birth, date_of_birth, AGE(NOW(), date_of_birth) AS age FROM person;
```

```

|          age
+-----+
| 65 years 1 mon 5 days 23:01:03.572283
| 97 years 7 mons 29 days 23:01:03.572283

```

- **Handling CONSTRAINT:**

Drop all CONSTRAINTs including UNIQUE and PRIMARY KEY.:

```
test=# ALTER TABLE person DROP CONSTRAINT person_pkey;  
ALTER TABLE
```

Add Back Primary Key:

```
test=# ALTER TABLE person ADD PRIMARY KEY (id);  
ALTER TABLE
```

Add UNIQUE Constraints:

```
test=# ALTER TABLE person ADD UNIQUE (email);  
ALTER TABLE
```

```
test=# ALTER TABLE person ADD CONSTRAINT unique_email_address UNIQUE (email);
```

Add CHECK Constraints:

```
test=# ALTER TABLE person ADD CONSTRAINT gender_constraint CHECK (gender = 'Female' OR gender = 'Male');  
ALTER TABLE
```

Add ON CONFLICT Constraints:

```
test=# ON CONFLICT (id) DO NOTHING;
```

```
VALUES (2011, 'RUSS', 'Radduch', 'Male', 'Radduch@nhs.uk');  
ON CONFLICT (id) DO UPDATE SET email = EXCLUDED.email;
```

- **Update Records:**

```
test=# UPDATE person SET first_name = 'Omar', last_name = 'Montana', email = 'omar.montana@hotmail.com' WHERE id = 2011;  
UPDATE 1
```

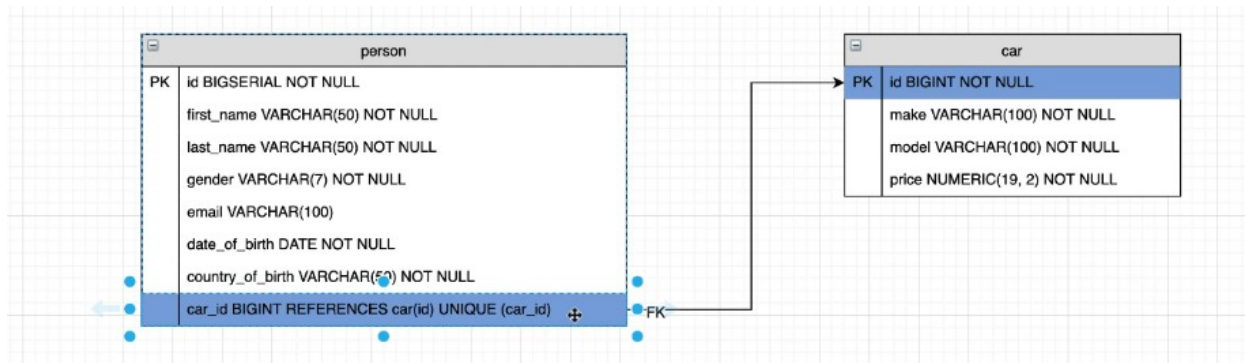
- **Execute a file in PostgreSQL**

```
test=# \i /Users/amigoscode/Downloads/person.sql
```

- **DELETE a record:**

```
test=# DELETE FROM person WHERE gender = 'Female' AND country_of_birth = 'Nigeria';  
DELETE 3
```

Relationship:



```

create table car (
    id BIGSERIAL NOT NULL PRIMARY KEY,
    make VARCHAR(100) NOT NULL,
    model VARCHAR(100) NOT NULL,
    price NUMERIC(19, 2) NOT NULL
);

create table person (
    id BIGSERIAL NOT NULL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    gender VARCHAR(7) NOT NULL,
    email VARCHAR(100),
    date_of_birth DATE NOT NULL,
    country_of_birth VARCHAR(50) NOT NULL,
    car_id BIGINT REFERENCES car(id),
    UNIQUE(car_id)
);
  
```

```

test=# SELECT * FROM person;
 id | first_name | last_name | gender | email | date_of_birth | country_of_birth | car_id
-----+-----+-----+-----+-----+-----+-----+-----
  1 | Fernanda  | Beardon  | Female | fernandab@is.gd | 1953-10-28 | Comoros | 
  2 | Omar      | Colmore  | Male   | | 1921-04-03 | Finland | 
  3 | Adriana   | Matuschek | Female | amatuschek2@feedburner.com | 1965-02-28 | Cameroon | 
(3 rows)

test=# SELECT * FROM car;
 id | make | model | price
-----+-----+-----+-----
  1 | Land Rover | Sterling | 87665.38
  2 | GMC | Acadia | 17662.69
(2 rows)

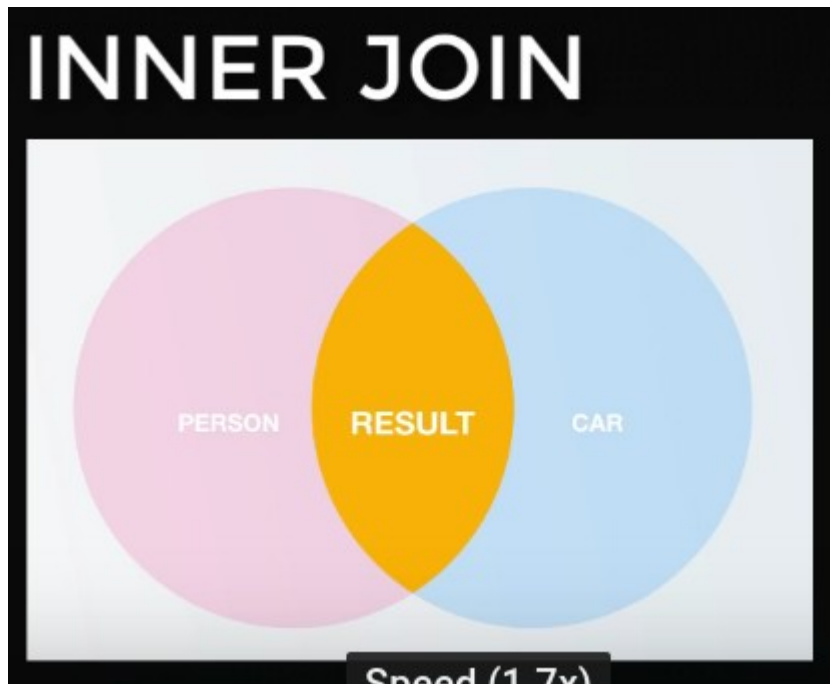
test=# UPDATE person SET car_id = 2 WHERE id = 1;
UPDATE 1
test=# SELECT * FROM car;
 id | make | model | price
-----+-----+-----+-----
  1 | Land Rover | Sterling | 87665.38
  2 | GMC | Acadia | 17662.69
(2 rows)

test=# SELECT * FROM person;
 id | first_name | last_name | gender | email | date_of_birth | country_of_birth | car_id
-----+-----+-----+-----+-----+-----+-----+-----
  2 | Omar      | Colmore  | Male   | | 1921-04-03 | Finland | 
  3 | Adriana   | Matuschek | Female | amatuschek2@feedburner.com | 1965-02-28 | Cameroon | 
  1 | Fernanda  | Beardon  | Female | fernandab@is.gd | 1953-10-28 | Comoros | 2
(3 rows)
  
```

- JOINS:

Inner Join

```
test=# SELECT * FROM person
test=# JOIN car ON person.car_id = car.id;
```



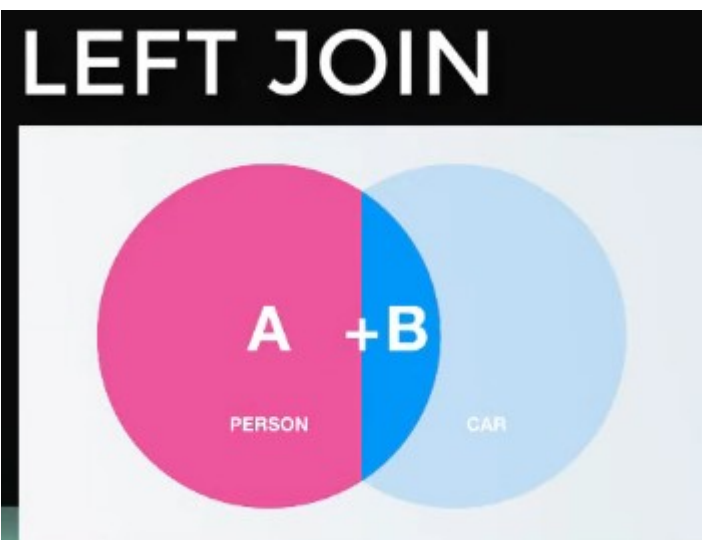
```
test=# SELECT person.first_name, car.make, car.model, car.price
test=# FROM person
test=# JOIN car ON person.car_id = car.id;
```

LEFT JOIN

```
test=# SELECT * FROM person
test=# LEFT JOIN car ON car.id = person.car_id;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id	id	make	model	price
2	Omar	Colmore	Male		1921-04-03	Finland	1	1	Land Rover	Sterling	87665.38
1	Fernanda	Beardon	Female	fernandab@is.gd	1953-10-28	Comoros	2	2	GMC	Acadia	17662.69
3	Adriana	Matuschek	Female	amatuschek2@feedburner.com	1965-02-28	Cameroon					

(3 rows)




```
test=# SELECT * FROM person
LEFT JOIN car ON car.id = person.car_id;
id | first_name | last_name | gender | email | date_of_birth | country_of_birth | car_id | id | make | model | price
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
2 | Omar | Colmore | Male | | 1921-04-03 | Finland | 1 | 1 | Land Rover | Sterling | 87665.38
1 | Fernanda | Beardon | Female | fernandab@is.gd | 1953-10-28 | Comoros | 2 | 2 | GMC | Acadia | 17662.69
3 | Adriana | Matuschek | Female | amatuschek2@feedburner.com | 1965-02-28 | Cameroon | | | | | 
(3 rows)

test=# SELECT * FROM person
JOIN car ON person.car_id = car.id;
id | first_name | last_name | gender | email | date_of_birth | country_of_birth | car_id | id | make | model | price
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
2 | Omar | Colmore | Male | | 1921-04-03 | Finland | 1 | 1 | Land Rover | Sterling | 87665.38
1 | Fernanda | Beardon | Female | fernandab@is.gd | 1953-10-28 | Comoros | 2 | 2 | GMC | Acadia | 17662.69
```

DELETING RECORDS IN REFERENCED AND REFRENCING TABLE:

```
test=# DELETE FROM car WHERE id = 13;
ERROR: update or delete on table "car" violates foreign key constraint "person_car_id_fkey" on table "person"
DETAIL: Key (id)=(13) is still referenced from table "person".
test=# DELETE FROM person WHERE id = 9000;
DELETE 1
test=# SELECT * FROM person WHERE id = 9000;
id | first_name | last_name | gender | email | date_of_birth | country_of_birth | car_id
-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

test=# DELETE FROM car WHERE id = 13;
DELETE 1
test=# SELECT * FROM car WHERE id = 13;
id | make | model | price
-----+-----+-----+-----
(0 rows)
```

we can also use **CASCADE DELETE**, it will delete other linked records also (but it is a bad practice)

- **Exporting Query Result to CSV file**

```
test=# \copy (SELECT * FROM person LEFT JOIN car ON car.id = person.car_id) TO '/Users/amigoscode/Desktop/results.csv' DELIMITER ',' CSV HEADER;
COPY 3
test=#
```


Data Types

Numeric Types:-

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

Monetary Types

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

Character Types

S. No.	Name & Description
	character varying(n), varchar(n)
1	variable-length with limit
	character(n), char(n)
2	fixed-length, blank padded
	text
3	variable unlimited length

Binary Data Types

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

Date/Time Types

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
TIMESTAMPTZ	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

SELECT the last day of month:

```
SELECT (DATE_TRUNC('MONTH', ('201608'||'01')::DATE) + INTERVAL '1 MONTH - 1 day')::DATE;
```

Cast a timestamp or interval to a string:

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'DD Mon YYYY HH:MI:SSPM');
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP,
' "Today is "FMDay", the "DDth" day of the month of "FMMonth" of "YYYY " );
```

Count the number of records per week

```
SELECT DATE_TRUNC('week', <>) AS "Week" , COUNT(*)
FROM <>
GROUP BY 1
ORDER BY 1;
```

Boolean Type

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

Enumerated Type

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

Geometric Type

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	<(x,y),r> (center point and radius)

Network Address Type

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

UUID Type

A UUID (Universally Unique Identifiers) is written as a sequence of lower-case hexadecimal digits,

An example of a UUID is – 550e8400-e29b-41d4-a716-446655440000

Array Type

Declaration of Arrays

```
SELECT INTEGER[];
```

```
SELECT INTEGER[3];
SELECT INTEGER[][];
SELECT INTEGER[3][3];
SELECT INTEGER ARRAY;
SELECT INTEGER ARRAY[3];
```

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

By default PostgreSQL **uses a one-based numbering convention for arrays**, that is, an array of n elements starts

with ARRAY[1] and ends with ARRAY[n].

Inserting values

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

Accessing Arrays

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

Modifying Arrays

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

or using the ARRAY expression syntax –

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

Searching Arrays

If Size of Array is known:

```
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 Size of Array is **not known**:

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

Composite Types

Declaration of Composite Types

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

Using:

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

Composite Value Input

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

Accessing Composite Types

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

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

PostgreSQL - CREATE Database

```
CREATE DATABASE dbname;
```

or

```
CREATEDB dbname;    //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.
```

Syntax

```
createdb [option...] [dbname [description]]
```

Parameters

S. No.	Parameter & Description
	dbname
1	The name of a database to create.
	description
2	Specifies a comment to be associated with the newly created database.
	options
3	command-line arguments, which createdb accepts.

Options

S. No.	Option & Description
	-D tablespace
1	Specifies the default tablespace for the database.
	-e
2	Echo the commands that createdb generates and sends to the server.
	-E encoding
3	Specifies the character encoding scheme to be used in this database.
	-l locale
4	Specifies the locale to be used in this database.
	-T template
5	Specifies the template database from which to build this database.
	--help
6	Show help about createdb command line arguments, and exit.
	-h host
7	Specifies the host name of the machine on which the server is running.
8	-p port

Specifies the TCP port or the local Unix domain socket file extension on which the server is listening for connections.

-U username

9 User name to connect as.

-w

10 Never issue a password prompt.

-W

11 Force createdb to prompt for a password before connecting to a database.

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

//The above given command will prompt you for password of the PostgreSQL admin user, which is **postgres**, by default. Hence, provide a password and proceed to create your new database

list of databases using \l
postgres=# \l

Command to connect/select a desired database; here, we will connect to the *testdb* database.

```
postgres=# \c testdb;
```

select your database from the command prompt itself at the time when you login to your database.

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

To exit from the database, you can use the command \q.

PostgreSQL - DROP 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 (connect to postgres or any other database to issue this command).

Syntax

The syntax for DROP DATABASE is given below –

DROP DATABASE [IF EXISTS] name

Parameters

S. No.	Parameter & Description
	IF EXISTS

- 1 Do not throw an error if the database does not exist. A notice is issued in this case.

name

- 2 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

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

Using dropdb Command

PostgreSQL command line executable **dropdb** is a 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 super user or the owner of the database.

Syntax

The syntax for *dropdb* is as shown below –

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

Parameters

S. No.	Parameter & Description
	dbname

- 1 The name of a database to be deleted.

option

- 2 command-line arguments, which *dropdb* accepts.

Options

S.	Option & Description
----	----------------------

No.

-e

- 1 Shows the commands being sent to the server.

-i

- 2 Issues a verification prompt before doing anything destructive.

-V

- 3 Print the dropdb version and exit.

--if-exists

- 4 Do not throw an error if the database does not exist. A notice is issued in this case.

--help

- 5 Show help about dropdb command-line arguments, and exit.

-h host

- 6 Specifies the host name of the machine on which the server is running.

-p port

- 7 Specifies the TCP port or the local UNIX domain socket file extension on which the server is listening for connections.

-U username

- 8 User name to connect as.

-w

- 9 Never issue a password prompt.

-W

- 10 Force dropdb to prompt for a password before connecting to a database.

--maintenance-db=dbname

- 11 Specifies the name of the database to connect to in order to drop the target database.

Example

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

Password for user postgres: ****

The above command drops the database **testdb**. Here, I have used the **postgres** (found under the pg_roles of template1) username to drop the database.

PostgreSQL - CREATE Table

Syntax

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

Examples

```
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
```

```
testdb-# \d tablename
```

PostgreSQL - DROP 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 –

```
DROP TABLE table_name;
```

Example

```
testdb-# \d
```

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

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.

Syntax

```
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 –

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

Example

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

```
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)
);
```

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

Syntax to Drop Schema

```
DROP SCHEMA myschema;    //if it is empty (all objects in it have been dropped)
```

```
DROP SCHEMA myschema CASCADE; //To drop a schema including all contained
                                objects,
```

```
????????????????????????????????????????????????????????????????????????????????????????????
```

PostgreSQL - INSERT Query

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

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table.

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

Example:

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

```
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');
```

PostgreSQL - SELECT Query

Syntax

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

```
SELECT * FROM table_name;
```

PostgreSQL - Operators.

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators

PostgreSQL Arithmetic Operators

Assume variable **a** holds 2 and variable **b** holds 3, then –

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 5
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -1
*	Multiplication - Multiplies values on either side of the operator	a * b will give 6
/	Division - Divides left hand operand by right hand operand	b / a will give 1
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 1
^	Exponentiation - This gives the exponent value of the right hand operand	a ^ b will give 8
/	square root	/ 25.0 will give 5
/	Cube root	/ 27.0 will give 3
!	factorial	5 ! will give 120
!!	factorial (prefix operator)	!! 5 will give 120

PostgreSQL Comparison Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

PostgreSQL Logical Operators

Here is a list of all the logical operators available in PostgreSQL.

S.

No.

Operator & Description

AND

- 1 The AND operator allows the existence of multiple conditions in a PostgreSQL statement's WHERE clause.

NOT

- 2 The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. **This is negate operator.**

OR

- 3 The OR operator is used to combine multiple conditions in a PostgreSQL statement's WHERE clause.

PostgreSQL Bit String Operators

The Bitwise operators supported by PostgreSQL are listed in the following table –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111
#	bitwise XOR.	A # B will give 49 which is 00110001

PostgreSQL - Expressions

```
SELECT * FROM COMPANY WHERE SALARY = 10000;
```

```
SELECT (15 + 6) AS ADDITION ;
```

```
SELECT COUNT(*) AS "RECORDS" FROM COMPANY;
```

```
SELECT CURRENT_TIMESTAMP;
```

PostgreSQL - WHERE Clause

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

```
SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
```

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

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

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

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

PostgreSQL - AND and OR Conjunctive Operators

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

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

PostgreSQL - UPDATE Query

Syntax

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

Example

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

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;
```

PostgreSQL - DELETE Query

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

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;
```

PostgreSQL - LIKE Clause

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

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

Syntax

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

```
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%3'
```

Finds any values that have 2 in the second position and end with a 3

Example:

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

PostgreSQL - LIMIT Clause

Syntax

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

Example

```
# 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 COMPANY LIMIT 4;
```

This would 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
(4 rows)
```

```
testdb=# SELECT * FROM COMPANY LIMIT 3 OFFSET 2; // Limit will display 3 records
Offset will skip first 2 records.
```

This would produce the following result –

```
id | name  | age | address    | salary
---+-----+---+-----+-----
 3 | Teddy | 23  | Norway     | 20000
 4 | Mark  | 25  | Rich-Mond  | 65000
 5 | David | 27  | Texas      | 85000
(3 rows)
```

PostgreSQL - ORDER BY Clause

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

```
SELECT * FROM COMPANY ORDER BY AGE ASC;
```

```
SELECT * FROM COMPANY ORDER BY NAME DESC;
```

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

PostgreSQL - GROUP BY

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

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

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

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

PostgreSQL - WITH Clause

The WITH query being CTE query, is particularly useful when subquery is executed multiple times. **It is equally helpful in place of temporary tables.**

Syntax

```
WITH
    name_for_summary_data AS (
        SELECT Statement)
SELECT columns
FROM name_for_summary_data
WHERE conditions <=> (
    SELECT column
    FROM name_for_summary_data)
[ORDER BY columns]
```

WITH

```
cte_name AS (
    CTE Query)
Main Query using CTE query result
```

EXAMPLE:

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

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.

[illegible]

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause –

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

DISTINCT keyword to eliminate duplicate records

```
SELECT DISTINCT name FROM COMPANY;
```

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.

```

CREATE TABLE COMPANY7(
  ID INT PRIMARY KEY NOT NULL,
  NAME TEXT,
  AGE INT DEFAULT 0,
  ADDRESS CHAR(50) UNIQUE,
  EMP_ID INT references COMPANY6(ID)
  SALARY REAL CHECK(SALARY > 0),
  EXCLUDE USING gist
  (NAME WITH =,
  AGE WITH <>)
);

```

```

ERROR:  conflicting key value violates exclusion constraint
"company7_name_age_excl"
DETAIL:  Key (name, age)=(Paul, 42) conflicts with existing key (name,
age)=(Paul, 32).

```

PostgreSQL - JOINS

A. INNER JOIN

Syntax:

```

SELECT table1.column1, table1.column2, table2.column1, ...
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;

```

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.

B. LEFT JOIN

Syntax:

```

SELECT table1.column1, table1.column2, table2.column1, ...
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;

```

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.

C. RIGHT JOIN

Syntax:

```

SELECT table1.column1, table1.column2, table2.column1, ...
FROM table1
RIGHT JOIN table2

```

ON table1.matching_column = table2.matching_column;

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.

D. FULL JOIN

Syntax:

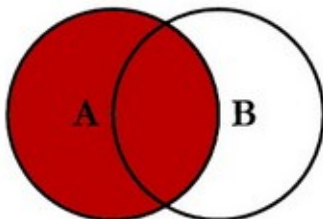
```
SELECT table1.column1, table1.column2, table2.column1, ....  
FROM table1  
FULL JOIN table2  
ON table1.matching_column = table2.matching_column;
```

table1: First table.

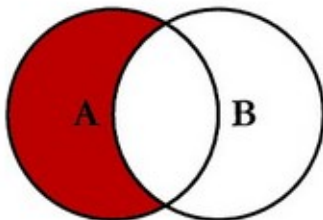
table2: Second table

matching_column: Column common to both the tables.

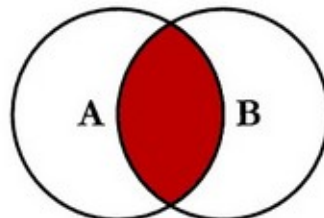
SQL JOINS



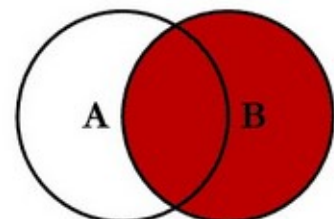
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



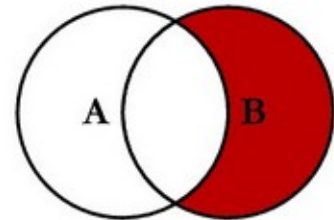
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL.
```



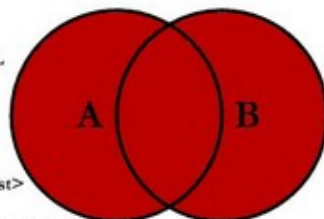
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



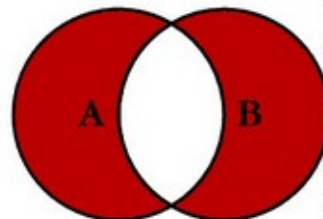
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL.
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```

MySQL JOIN Types


Created by Steve Stedman



SELECT *
FROM Table1;


SELECT *
FROM Table2;

SELECT from two tables



SELECT *
FROM Table1 t1
INNER JOIN Table2 t2
ON t1.fk = t2.id;

INNER JOIN



SELECT *
FROM Table1 t1
LEFT OUTER JOIN Table2 t2
ON t1.fk = t2.id;

LEFT OUTER JOIN




SELECT *
FROM Table1 t1
RIGHT OUTER JOIN Table2 t2
ON t1.fk = t2.id;

RIGHT OUTER JOIN




SELECT *
FROM Table1 t1
WHERE EXISTS (SELECT 1
FROM Table2 t2
WHERE t1.fk = t2.id
);

SEMI JOIN – Similar to INNER JOIN, with less duplication.




SELECT *
FROM Table1 t1
WHERE NOT EXISTS (SELECT 1
FROM Table2 t2
WHERE t1.fk = t2.id
);

ANTI SEMI JOIN



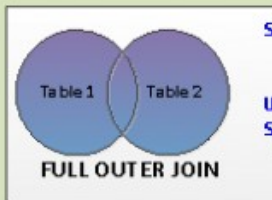
SELECT *
FROM Table1 t1
LEFT OUTER JOIN Table2 t2
ON t1.fk = t2.id
WHERE t2.id is null;

LEFT OUTER JOIN with exclusion



SELECT *
FROM Table1 t1
RIGHT OUTER JOIN Table2 t2
ON t1.fk = t2.id
WHERE t1.fk is null;

RIGHT OUTER JOIN with exclusion



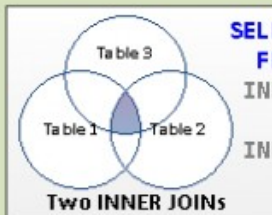
SELECT * FROM Table1 t1
LEFT OUTER JOIN Table2 t2
ON t1.fk = t2.id
UNION
SELECT * FROM Table1 t1
RIGHT OUTER JOIN Table2 t2
ON t1.fk = t2.id;

FULL OUTER JOIN



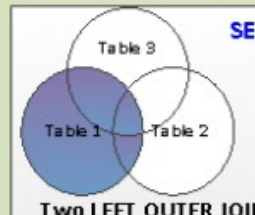
SELECT * FROM Table1 t1
LEFT OUTER JOIN Table2 t2
ON t1.fk = t2.id
WHERE t2.id IS NOT NULL
UNION
SELECT * FROM Table1 t1
RIGHT OUTER JOIN Table2 t2
ON t1.fk = t2.id
WHERE t1.id IS NOT NULL;

FULL OUTER JOIN with exclusion




SELECT *
FROM Table1 t1
INNER JOIN Table2 t2
ON t1.fk = t2.id
INNER JOIN Table3 t3
ON t1.fk_table3 = t3.id;

Two INNER JOINS



SELECT *
FROM Table1 t1
LEFT OUTER JOIN Table2 t2
ON t1.fk = t2.id
LEFT OUTER JOIN Table3 t3
ON t1.fk_table3 = t3.id;

Two LEFT OUTER JOINS



SELECT *
FROM Table1 t1
INNER JOIN Table2 t2
ON t1.fk = t2.id
LEFT OUTER JOIN Table3 t3
ON t1.fk_table3 = t3.id;

INNER JOIN and a LEFT OUTER JOIN

NATURAL JOIN:

```
SELECT *
FROM Student NATURAL JOIN Marks; //common attribute must have same name
```

INNER JOIN:

```
SELECT *
FROM student S INNER JOIN Marks M ON S.Roll_No = M.Roll_No;
```

Difference between Natural JOIN and INNER JOIN in SQL :

SR.NO.	NATURAL JOIN	INNER JOIN
1.	Natural Join joins two tables based on same attribute name and datatypes.	Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause.
2.	In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column	In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also
3.	In Natural Join, If there is no condition specifies then it returns the rows based on the common column	In Inner Join, only those records will return which exists in both the tables
4.	SYNTAX: SELECT * FROM table1 NATURAL JOIN table2;	SYNTAX: SELECT * FROM table1 INNER JOIN table2 ON table1.Column_Name = table2.Column_Name;

Join = cross product + some condition;

CARTESIAN JOIN:

- In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT . i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
- In the presence of WHERE condition this JOIN will function like a INNER JOIN.
- Generally speaking, Cross join is similar to an inner join where the join-condition will always evaluate to True

Syntax:

```
SELECT table1.column1 , table1.column2, table2.column1...
FROM table1
CROSS JOIN table2;
```

Example:

```
SELECT Student.NAME, Student.AGE, StudentCourse.COURSE_ID
FROM Student
CROSS JOIN StudentCourse;
```

SELF JOIN:

```
SELECT a.ROLL_NO , b.NAME
FROM Student a, Student b
WHERE a.ROLL_NO < b.ROLL_NO;
```

EQUI JOIN :

Syntax :

```
SELECT column_list
FROM table1, table2....
WHERE table1.column_name =
table2.column_name;
```

Example –

```
SELECT student.name, student.id, record.class, record.city
FROM student, record WHERE student.city = record.city;
```

PostgreSQL - UNION

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

EXAMPLE:

```
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;
```

The UNION ALL Clause

The **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

```
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 JOIN DEPARTMENT
ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

PostgreSQL - NULL value

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

EXAMPLE:

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

PostgreSQL - ALIAS

```
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;
```

PostgreSQL - ALTER TABLE

Syntax

ALTER TABLE to **add a new column** in an existing table

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

DROP COLUMN in an existing table –

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

to change the **DATA TYPE** of a column in a table is as follows –

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

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;
```

to **ADD UNIQUE CONSTRAINT** to a table is as follows –

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

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...);
```

to **DROP CONSTRAINT** from a table is as follows –

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

to **DROP PRIMARY KEY** constraint from a table is as follows –

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

PostgreSQL - TRUNCATE TABLE

The PostgreSQL **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 DELETE on each table, but since it does not actually scan the tables, it is faster.

Syntax

```
TRUNCATE TABLE table_name;
```

EXAMPLE:

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

PostgreSQL -VIEWS

Views are pseudo-tables.

Since views are not ordinary tables, you may not be able to execute a DELETE, INSERT, or UPDATE statement on a view. However, you can create a RULE to correct this problem of using DELETE, INSERT or UPDATE on a view.

```
testdb=# CREATE VIEW COMPANY_VIEW AS
```

```
SELECT ID, NAME, AGE  
FROM COMPANY;
```

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

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

PostgreSQL - Transaction Control

The following commands are 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 cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The 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.

BEGIN;

or

BEGIN TRANSACTION;

The 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.

COMMIT;

or

END TRANSACTION;

The ROLLBACK Command

The 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.

ROLLBACK;

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

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

PostgreSQL - LOCKS

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

```
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.

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

PostgreSQL - Subqueries

Subqueries with the INSERT Statement

```
testdb=# INSERT INTO COMPANY_BKP  
SELECT * FROM COMPANY
```

```
WHERE ID IN (SELECT ID
FROM COMPANY) ;
```

Subqueries with the UPDATE Statement

```
testdb=# UPDATE COMPANY
SET SALARY = SALARY * 0.50
WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
WHERE AGE >= 27 );
```

Subqueries with the DELETE Statement

```
testdb=# DELETE FROM COMPANY
WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
WHERE AGE > 27 );
```

PostgreSQL - PRIVILEGES

Syntax for GRANT

```
GRANT privilege [, ...]
ON object [, ...]
TO { PUBLIC | GROUP group | username }
```

- privilege** – values could be: SELECT, INSERT, UPDATE, DELETE, RULE, ALL.
- object** – The name of an object to which to grant access. The possible objects are: table, view, sequence
- PUBLIC** – A short form representing all users.
- GROUP group** – A group to whom to grant privileges.
- username** – The name of a user to whom to grant privileges. PUBLIC is a short form representing all users.

The privileges can be revoked using the REVOKE command.

Syntax for REVOKE

Basic syntax for REVOKE command is as follows –


```
REVOKE privilege [, ...]
ON object [, ...]
FROM { PUBLIC | GROUP groupname | username }
```

- **privilege** – values could be: SELECT, INSERT, UPDATE, DELETE, RULE, ALL.
- **object** – The name of an object to which to grant access. The possible objects are: table, view, sequence
- **PUBLIC** – A short form representing all users.
- **GROUP group** – A group to whom to grant privileges.
- **username** – The name of a user to whom to grant privileges. PUBLIC is a short form representing all users.

Example

```
testdb=# CREATE USER manisha WITH PASSWORD 'password';
```

```
CREATE ROLE
```

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

```
GRANT
```

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

```
REVOKE
```

```
testdb=# DROP USER manisha;
```

```
DROP ROLE
```

PostgreSQL - DATE/TIME Functions and Operators

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

*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

The following is the list of all important Date and Time related functions available.

S. No.	Function & Description
1	AGE() Subtract arguments
2	CURRENT DATE/TIME() Current date and time
3	DATE_PART() Get subfield (equivalent to extract)
4	EXTRACT() Get subfield
5	ISFINITE() Test for finite date, time and interval (not +/-infinity)
6	JUSTIFY Adjust interval

AGE(timestamp, timestamp), AGE(timestamp)

S. No.	Function & Description
1	AGE(timestamp, timestamp) When invoked with the TIMESTAMP form of the second argument, AGE() subtract arguments, producing a "symbolic" result that uses years and months and is of type INTERVAL.
2	AGE(timestamp) When invoked with only the TIMESTAMP as argument, AGE() subtracts from the current_date (at midnight).

Example of the function AGE(timestamp, timestamp) is –

```
testdb=# SELECT AGE(timestamp '2001-04-10', timestamp '1957-06-13');
```

The above given PostgreSQL statement will produce the following result –

age
43 years 9 mons 27 days

Example of the function AGE(timestamp) is –

```
testdb=# select age(timestamp '1957-06-13');
```

The above given PostgreSQL statement will produce the following result –

age
55 years 10 mons 22 days

CURRENT DATE/TIME()

PostgreSQL provides a number of functions that return values related to the current date and time. Following are some functions –

S. No.	Function & Description
1	CURRENT_DATE Delivers current date.
2	CURRENT_TIME Delivers values with time zone.
3	CURRENT_TIMESTAMP Delivers values with time zone.
4	CURRENT_TIME(precision) Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.
5	CURRENT_TIMESTAMP(precision) Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.
6	LOCALTIME Delivers values without time zone.
7	LOCALTIMESTAMP Delivers values without time zone.
8	LOCALTIME(precision)

	Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.
9	LOCALTIMESTAMP(precision) Optionally takes a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field.

Examples using the functions from the table above –

```
testdb=# SELECT CURRENT_TIME;
      timetz
```

```
-----
 08:01:34.656+05:30
(1 row)
```

```
testdb=# SELECT CURRENT_DATE;
      date
```

```
-----
2013-05-05
(1 row)
```

```
testdb=# SELECT CURRENT_TIMESTAMP;
      now
```

```
-----
2013-05-05 08:01:45.375+05:30
(1 row)
```

```
testdb=# SELECT CURRENT_TIMESTAMP(2);
      timestamptz
```

```
-----  
2013-05-05 08:01:50.89+05:30  
(1 row)  
  
testdb=# SELECT LOCALTIMESTAMP;  
         timestamp  
-----  
2013-05-05 08:01:55.75  
(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 –

S. No.	Function & Description
1	transaction_timestamp() It is equivalent to CURRENT_TIMESTAMP, but is named to clearly reflect what it returns.
2	statement_timestamp() It returns the start time of the current statement.
3	clock_timestamp() It returns the actual current time, and therefore its value changes even within a single SQL command.
4	timeofday()

	It returns the actual current time, but as a formatted text string rather than a timestamp with time zone value.
5	now() It is a traditional PostgreSQL equivalent to transaction_timestamp().

DATE_PART(text, timestamp), DATE_PART(text, interval), DATE_TRUNC(text, timestamp)

S. No .	Function & Description
1	DATE_PART('field', source) <p>These functions get the subfields. The <i>field</i> parameter needs to be a string value, not a name.</p> <p>The valid field names are: <i>century, day, decade, dow, doy, epoch, hour, isodow, isoyear, microseconds, millennium, milliseconds, minute, month, quarter, second, timezone, timezone_hour, timezone_minute, week, year.</i></p>
2	DATE_TRUNC('field', source) <p>This function is conceptually similar to the <i>trunc</i> function for numbers. <i>source</i> is a value expression of type timestamp or interval. <i>field</i> selects to which precision to truncate the input value. The return value is of type <i>timestamp</i> or <i>interval</i>.</p> <p>The valid values for <i>field</i> are : <i>microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year, decade, century, millennium</i></p>

The following are examples for DATE_PART(*field*, source) functions –

```
testdb=# SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
         16
(1 row)
```

```
testdb=# SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
date_part
-----
         4
(1 row)
```

The following are examples for DATE_TRUNC(*field*, source) functions –

```
testdb=# SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
date_trunc
-----
2001-02-16 20:00:00
(1 row)
```

```
testdb=# SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
date_trunc
-----
2001-01-01 00:00:00
(1 row)
```

EXTRACT(field from timestamp), EXTRACT(field from interval)

The **EXTRACT(field FROM source)** function retrieves subfields such as year or hour from date/time values. The *source* must be a value expression of

type *timestamp*, *time*, or *interval*. The *field* is an identifier or string that selects what field to extract from the source value. The EXTRACT function returns values of type *double precision*.

The following are valid field names (similar to DATE_PART function field names): century, day, decade, dow, doy, epoch, hour, isodow, isoyear, microseconds, millennium, milliseconds, minute, month, quarter, second, timezone, timezone_hour, timezone_minute, week, year.

The following are examples of EXTRACT(*field*, source) functions –

```
testdb=# SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
date_part
-----
         20
(1 row)

testdb=# SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
        16
(1 row)
```

ISFINITE(date), ISFINITE(timestamp), ISFINITE(interval)

S. No.	Function & Description
1	ISFINITE(date) Tests for finite date.
2	ISFINITE(timestamp) Tests for finite time

	stamp.
3	ISFINITE(interval) Tests for finite interval.

The following are the examples of the ISFINITE() functions –

```
testdb=# SELECT isfinite(date '2001-02-16');
```

```
isfinite
```

```
-----
```

```
t
```

```
(1 row)
```

```
testdb=# SELECT isfinite(timestamp '2001-02-16 21:28:30');
```

```
isfinite
```

```
-----
```

```
t
```

```
(1 row)
```

```
testdb=# SELECT isfinite(interval '4 hours');
```

```
isfinite
```

```
-----
```

```
t
```

```
(1 row)
```

**JUSTIFY_DAYS(interval), JUSTIFY_HOURS(interval),
JUSTIFY_INTERVAL(interval)**

S. No.	Function & Description
1	JUSTIFY_DAYS(interval)

	Adjusts interval so 30-day time periods are represented as months. Return the interval type
2	JUSTIFY_HOURS(interval) Adjusts interval so 24-hour time periods are represented as days. Return the interval type
3	JUSTIFY_INTERVAL(interval) Adjusts interval using JUSTIFY_DAYS and JUSTIFY_HOURS, with additional sign adjustments. Return the interval type

The following are the examples for the ISFINITE() functions –

```
testdb=# SELECT justify_days(interval '35 days');
```

```
justify_days
```

```
-----
1 mon 5 days
(1 row)
```

```
testdb=# SELECT justify_hours(interval '27 hours');
```

```
justify_hours
```

```
-----
1 day 03:00:00
(1 row)
```

```
testdb=# SELECT justify_interval(interval '1 mon -1 hour');
```

```
justify_interval
```

```
-----
29 days 23:00:00
```

(1 row)

PostgreSQL - Functions

Syntax

The basic syntax to create a function is as follows –

```
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;
```

Where,

- **function-name** specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The function must contain a **return** statement.
- **RETURN** clause specifies that data type you are going to return from the function. The **return_datatype** can be a base, composite, or domain type, or can reference the type of a table column.
- **function-body** contains the executable part.
- The AS keyword is used for creating a standalone function.
- **plpgsql** is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL, it can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example

Function totalRecords() is as follows –

```
sCREATE OR REPLACE FUNCTION totalRecords ()  
RETURNS integer AS $total$  
declare  
    total integer;  
BEGIN  
    SELECT count(*) into total FROM COMPANY;  
    RETURN total;  
END;  
$total$ LANGUAGE plpgsql;
```

```
testdb# CREATE FUNCTION  
testdb=# select totalRecords();
```

PostgreSQL - Triggers

PostgreSQL - Indexing

PostgreSQL - EXTRA

ADD REFERENCE USING ALTER:

```
ALTER TABLE sample.public.employee  
ADD FOREIGN KEY (col_name1) REFERENCES public.department  
(col_name2);
```

ADD NOT NULL CONSTRAINT:

```
ALTER TABLE table_name  
ALTER COLUMN column_name SET NOT NULL;
```

DROP NOT NULL CONSTRAINT:

```
ALTER TABLE YourTable ALTER COLUMN YourColumn DROP NOT NULL;
```

Rename Database:

```
ALTER DATABASE db RENAME TO newdb; //if it doesnot work close terminal and retry
```
