**POSTGRESQL – II**

**Limit**

**Offset**

**Ex:**

select \* from employee limit 5

select \* from employee offset 2

select \* from employee offset 2 limit 2

**Like:** Used to match pattern

Wild Chars \_ , %

S% - starting with s

%s - ending with s

%s% - contains s

\_a% -second letter is a

\_ \_s% -third char is s

Ex:

select \* from employee where ename like '\_a%'

select \* from employee where ename like 'R%'

select \* from employee where ename like '%a%'

**In**

Used to check the condition from a given list of values

Ex: select \* from employee where city in ('Hyd','bang','Delhi')

**Serial**

Automatically inserts serial numbers into a column

Ex:

create table products(

pno serial,

pname varchar(30),

price int

)

insert into products (pname, price) values('Chatger', 350)

select \* from products

**Working with Joins**

Joins is the concept of joining one or more table columns into a single result set

Joins used to retrieve the data from more than one table

For joining two tables , they must have atleast one common column.

**Inner Join**

**Outer Join – left outer join , right outer join , full outer join**

**Cross Join**

**Self Join**

**Equi join**

**Inner Join**

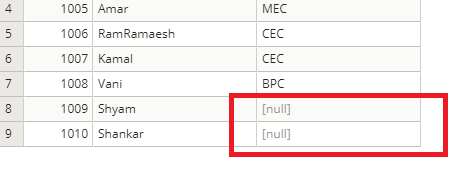
A INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. Display the records form the two tables which satisfy the join condition

Ex:

select student.sid, student.sname, course.cname from student join course on student.cid = course.cid

**Left Outer Join :**

Combines the two tables columns basing on the join condition and retrieves all rows from first table , if no matching value for the second table column value displays Null



**Right outer join**

Combines the two tables columns basing on the join condition and retrieves all rows from second table , if no matching value for the first table column value displays Null

Ex:

select student.sid, student.sname, course.cname from student right outer join course on student.cid = course.cid

**Full Outer Join**

Combines the two tables columns basing on the join condition and retrieves all rows from two tables , if no matching value displays Null

Ex:

select student.sid, student.sname, course.cname from student Full outer join course on student.cid = course.cid

Examples :

select student.sid, student.sname, course.cname

from student join course

on student.cid = course.cid

select student.sid, student.sname, course.cname

from student left outer join course

on student.cid = course.cid

select student.sid, student.sname, course.cname

from student right outer join course

on student.cid = course.cid

select student.sid, student.sname, course.cname

from student full outer join course

on student.cid = course.cid

select s.sid, s.sname, c.cname

from student as s full outer join course as c

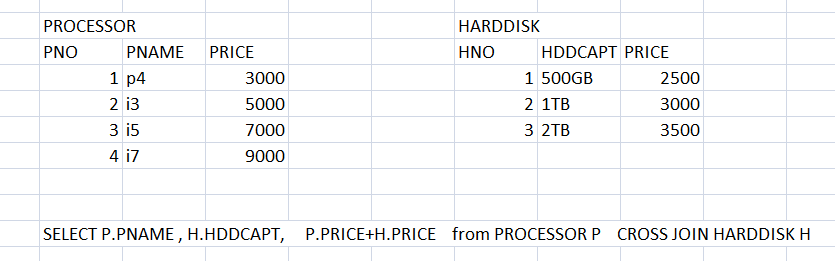
on s.cid = c.cid

**Cross Join**

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

Ex:

Select processor.pname, hdd.disk, processor.price+ hdd.price from processor cross join hdd



**Self join**

Joins the same table twice to get précised data

To join the same table twice we requires alias names for the table to avoid ambiguity

Ex:

select e.eno, e.ename, e1.ename from employee as e join employee e1 on e.mngrid = e1.eno

**UNION**

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

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

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.

Ex:

SELECT \* FROM STUDENT1 UNION SELECT \* FORM STUDENT2

SELECT \* FROM STUDENT1 UNIONALL SELECT \* FORM STUDENT2

**Joining three tables ex:**

select s.sname, c.cname, l.city

from student as s

join course as c on s.cid = c.cid

join clocation as l on c.cid= l.cid

**Working with SubQueries**

A subquery-also referred to as an *inner query* or*inner select*-is a SELECT statement embedded within a data manipulation language (DML) statement or nested within another subquery. You can use subqueries in SELECT, INSERT, UPDATE, and DELETE statements wherever expressions are allowed. For instance, you can use a subquery as one of the column expressions in a SELECT list or as a table expression in the FROM clause.

A DML statement that includes a subquery is referred to as the *outer query*. The following guidelines provide details about how to implement subqueries in your outer queries or in other subqueries:

* You must enclose a subquery in parenthesis.
* A subquery must include a SELECT clause and a FROM clause.
* A subquery can include optional WHERE, GROUP BY, and HAVING clauses.

Ex:

**select \* from employee where mngrid= (select mngrid from employee where eno =1004)**

Will display the employees whose manager id is same as eno 1004

First retrives the mngrid of the employee with eno 1004 and then retrives the employees basing on the mngrid

**select \* from employee where city in (select city from employee where eno = 1004 or eno = 1006)**

**Working with Views**

Views are sub tables . views are not real tables. A view never stores the data in it. A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table. A view can even represent joined tables. Because views are assigned separate permissions, you can use them to restrict table access so that the users see only specific rows or columns of a table.

A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables, which depends on the written PostgreSQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following −

* Structure data in a way that users or classes of users find natural or intuitive.
* Restrict access to the data such that a user can only see limited data instead of complete table.
* Summarize data from various tables, which can be used to generate reports.

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.

## Creating Views

The PostgreSQL views are created using the **CREATE VIEW** statement. The PostgreSQL views can be created from a single table, multiple tables, or another view.

The basic CREATE VIEW syntax is as follows −

**CREATE VIEW view\_name AS**

**SELECT column1, column2.....**

**FROM table\_name**

**WHERE Condition**

You can include multiple tables in your SELECT statement in very similar way as you use them in normal PostgreSQL SELECT query.

Consider, the [COMPANY](https://www.tutorialspoint.com/postgresql/company.sql) table having the columns id, name, age , address, salary

Now, following is an example to create a view from COMPANY table. This view would be used to have only few columns from COMPANY table −

CREATE VIEW COMPANY\_VIEW AS

SELECT ID, NAME, AGE

FROM COMPANY;

Now, you can query COMPANY\_VIEW in a similar way as you query an actual table. Following is the example −

SELECT \* FROM COMPANY\_VIEW;

## Dropping Views

To drop a view, simply use the DROP VIEW statement with the **view\_name**. The basic DROP VIEW syntax is as follows −

DROP VIEW view\_name;

The following command will delete COMPANY\_VIEW view, which we created in the last section −

DROP VIEW COMPANY\_VIEW;

**Working with Indexes**

CREATE INDEX constructs an index on the specified column(s) of the specified relation, which can be a table or a materialized view. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

PostgreSQL provides the index methods B-tree, hash, GiST, SP-GiST, GIN, and BRIN. Users can also define their own index methods, but that is fairly complicated.

**Partial Indexes**

When the WHERE clause is present, a partial index is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table. See [Section 11.8](https://www.postgresql.org/docs/current/static/indexes-partial.html) for more discussion.

The expression used in the WHERE clause can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in WHERE. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be “immutable”, that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or WHERE clause, remember to mark the function immutable when you create it.

**To create an index**

Create index index\_name on tableName(colName);

Ex:

create index ind\_eno on employee(eno)

create unique index ind\_uniq\_ename on employee(ename)

create index ind\_prt on employee(deptid) where deptid>30

To create a B-tree index on the column title in the table films:

CREATE UNIQUE INDEX title\_idx ON films (title);

To create an index on the expression lower(title), allowing efficient case-insensitive searches:

CREATE INDEX ON films ((lower(title)));

(In this example we have chosen to omit the index name, so the system will choose a name, typically films\_lower\_idx.)

To create an index with non-default sort ordering of nulls:

CREATE INDEX title\_idx\_nulls\_low ON films (title NULLS FIRST);

To create an index with non-default fill factor:

CREATE UNIQUE INDEX title\_idx ON films (title) WITH (fillfactor = 70);

**Working with User Functions**

PostgreSQL **functions**, also known as Stored Procedures, allow you to carry out operations that would normally take several queries and round trips in a single function within the database. Functions allow database reuse as other applications can interact directly with your stored procedures instead of a middle-tier or duplicating code.

We can declare variables and return values form the function

**To declare a variable**

declare

varName datatype ;

Ex:

Declare

totemp int ;

in order to create new user defined function

**create or replace function myFunc() returns integer as $$**

**declare**

**tot integer;**

**begin**

**select count(\*) into tot from employee;**

**return tot;**

**end**

**$$ language plpgsql**

Returns indicate what type the function is returning

$$ - indicates start of the function

Last statement is saying we are using plpgsql to write the fuction . we can write the fuction in other languages

Ex2 :

create or replace function countemployeebycity(c varchar(30)) returns integer as $$

declare

tot integer;

begin

select count(\*) into tot from employee where city =c;

return tot;

end

$$ language plpgsql

**Working with triggers**

PostgreSQL **Triggers** are functions, which automatically invoked when a specified database command fired

* PostgreSQL trigger can be specified to fire
  + Before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE or DELETE is attempted)
  + After the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed)
  + Instead of the operation (in the case of inserts, updates or deletes on a view)
* A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.
* Magic tables can be accessible when updating the data . can accessed by using NEW and OLD Keywords
* The BEFORE, AFTER or INSTEAD OF keyword determines when the trigger actions will be executed relative to the insertion, modification or removal of the associated row.
* Triggers are automatically dropped when the table that they are associated with is dropped.

The basic syntax of creating a **trigger** is as follows −

CREATE TRIGGER trigger\_name BEFORE|AFTER|INSTEAD OF SQLCOMMAND

ON table\_name

-- Trigger logic goes here....

testdb=# CREATE TRIGGER example\_trigger AFTER INSERT ON COMPANY

FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();

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

CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example\_table$

BEGIN

INSERT INTO AUDIT(EMP\_ID, ENTRY\_DATE) VALUES (new.ID, current\_timestamp);

RETURN NEW;

END;

$example\_table$ LANGUAGE plpgsql;

Now, we will start the actual work. Let us start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. So let us create one record in COMPANY table as follows −

testdb=# INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Paul', 32, 'California', 20000.00 );

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

id | name | age | address | salary

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

1 | Paul | 32 | California | 20000

Same time, one record will be created in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation on COMPANY table. Similarly, you can create your triggers on UPDATE and DELETE operations based on your requirements.

emp\_id | entry\_date

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

1 | 2013-05-05 15:49:59.968+05:30

(1 row)

Listing TRIGGERS

You can list down all the triggers in the current database from **pg\_trigger**table as follows −

testdb=# SELECT \* FROM pg\_trigger;

The above given PostgreSQL statement will list down all triggers.

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

testdb=# SELECT tgname FROM pg\_trigger, pg\_class WHERE tgrelid=pg\_class.oid AND relname='company';

The above given PostgreSQL statement will also list down only one entry as follows −

tgname

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

example\_trigger

(1 row)

Dropping TRIGGERS

The following is the DROP command, which can be used to drop an existing trigger −