Select \* from car OFFSET 5 LIMT 5;

OFFSET – starts taking data after ignoring first 5 rows;

LIMIT – returns only 5 rows

In this case, returns rows from 6-10

ORDER of query execution

FROM 🡪 WHERE 🡪 SELECT 🡪ORDER BY

Threrefore

(SELECT name , marks\*2 as “percentage” FROM students WHERE percentage > 80)

will fail

We should give  
(SELECT name , marks\*2 as “percentage” FROM students WHERE marks\*2 > 80)

Can also give

(SELECT name , marks\*2 as “percentage” FROM students WHERE marks\*2 > 80 ORDER BY percentage)

\? – help

\l – list all DB

\q –quit

\c- connect to a database

psql –h localhost –p 5050 –U postgres testDB - connect to DB {

-h = hostname

-p = port

-U = username

testDB is the DB that I wanna connect to

}

\i /Users/shakthi/Downloads - execute the contents in the file {

/Users/shakthi/Downloads - filepath

}

Press q and then enter \x and then press enter to expand the display

Table = person

Select \* from person WHERE date\_of\_birth BETWEEN DATE ‘1990 -04-01’ AND ‘2004-08-29’;

Select \* from person WHERE country\_of\_birth IN (‘FRANCE’ , ‘ITALY’ , ‘GERMANY’);

Select \* from person WHERE email\_id LIKE ‘%.com’; {

Gets all records with email ending in “.com”

}

Select \* from person WHERE email\_id LIKE ‘%@google%’; {

Gets all records with email having @google in them

}

Select \* from person WHERE user\_name LIKE ‘%sha\_th%’; {

Gets all records with user\_name having a single character between sha and th(shakthi , shanthi) in them

}

Select \* from person WHERE user\_name LIKE ‘%sha$\_th%’ escape ‘$’; {

Gets all records with user\_name exactly with sha\_th in them

}

Select country\_of\_birth , count(\*) FROM person GROUP BY country\_of\_birth HAVING COUNT(\*) >= 40 ORDER BY country\_of\_birth;

Having clause is used to specify conditions on aggregate functions

Table = car

Select MAX(price) FROM car;

Select AVG(price) FROM car;

Select ROUND(AVG(price)) FROM car;

Select SUM(price) FROM car;

Select id,make,model, price AS original\_price , ROUND(price \* .10 , 2) AS ten\_percent , ROUND(price – (price \*.10) , 2) AS discount\_after\_10\_percent\_from\_car;

NULLIF returns null, if argument 1 = argument 2, else it returns argument 1

SELECT NULLIF(0 , 0) - null

SELECT NULLIF(10 , 10) - null

SELECT NULLIF(10 , 1) - 10

SELECT 10 /0 - ERROR can not divide by 0

SELECT 10/NULL - NULL

SELECT 10/NULLIF(2 , 9) - 5

SELECT 10/NULLIF(0 , 0) - null

SELECT COALESCE(NULL , 8) - 8

SELECT COALESCE(10,NULL,8) - 10

SELECT COALESCE(10, 8) - 10

SELECT COALESCE(NULLIF(0,0) , 7) - 7

SELECT NOW(); - gives timestamp

now

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

2022-08-15 16:40:57.894242+05:30

SELECT NOW()::DATE; - gives DATE

now

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

2022-08-15

SELECT NOW()::TIME; - gives TIME

now

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

16:40:57.894242+05:30

postgres=# SELECT NOW() - interval '10 years';

?column?

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

2012-08-15 18:33:59.528471+05:30

postgres=# SELECT NOW() + INTERVAL '10 years';

? column?

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

2032-08-15 18:35:54.356071+05:30

postgres=# SELECT (NOW() + INTERVAL '10 years')::DATE;

date

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

2032-08-15

postgres=# SELECT EXTRACT(YEAR FROM NOW());

extract

---------

2022

postgres=# SELECT EXTRACT(MONTH FROM NOW());

extract

---------

8

postgres=# SELECT AGE(NOW() , '2001-04-22'::DATE);

age

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

21 years 3 mons 23 days 18:48:10.681282

Primary key constraint

ALTER TABLE person ADD PRIMARY KEY (id);

ALTER TABLE person DROP CONSTRAINT person\_pkey;

Unique constraint

ALTER TABLE person ADD CONSTRAINT unique\_email\_address UNIQUE (email);

ALTER TABLE person DROP CONSTRAINT unique\_email\_address;

Check constraint

ALTER TABLE person ADD CONSTRAINT gender\_constraint CHECK (gender = ‘FEMALE’ OR gender = ‘MALE’);

AUTO\_INCREMENT (use constraint near a column during CREATE)

DEFAULT value (use constraint near a column during CREATE)

ALTER TABLE table\_name DROP CONSTRAINT some\_name;

DELETE FROM person ; deletes all rows

DELETE FROM person WHERE gender = ‘FEMALE’;

UPDATE person SET email = ‘omar@gmail.com’ WHERE name = ‘omar’;

UPSERT(update + insert)

ON CONFLICT DO - to avoid error when PK is already present or UNIQUE constraint is violated

* To insert a new row. If a row already exists with given values it updates it

Record with id =1 exists already in the table

1. INSERT INTO person(id ,name , email , date\_of\_birth , country\_of\_birth) VALUES(1 , ‘RUSS’ , ‘MALE’ , ‘rr@gmail.com’ , DATE ‘1995-04-22’ , ‘Norway’) ON CONFLICT DO NOTHING;

Output : INSERT

1. 0
2. INSERT INTO person(id ,name , email , date\_of\_birth , country\_of\_birth) VALUES(1 , ‘RUSS’ , ‘MALE’ , ‘rr@gmail.com’ , DATE ‘1995-04-22’ , ‘Norway’) ON CONFLICT DO UPDATE SET email = EXCLUDED.email , name = EXCLUDED.name ;

Output : INSERT

1. 1

(name and email are updated for the record with id=1)

FOREIGN KEY

CREATE TABLE person(

id BIGSERIAL NOT NULL PRIMARY KEY,

name VARCHAR(50) NOT NULL,

gender VARCHAR(7),

email VARCHAR(100),

date\_of\_birth DATE NOT NULL,

country\_of\_birth VARCHAR(20) NOT NULL,

car\_id BIGINT REFERENCES car(id),

UNQUE(car\_id)

);

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

);

Table with 2 primary keys

Table having a column which is both PRIMARY and FOREIGN key

CREATE TABLE branch\_supplier(

branch\_id INT ,

supplier\_name VARCHAR(40),

suplply\_type VARCHAR(40),

PRIMARY KEY(branch\_id , supplier\_name),

FOREIGN KEY(branch\_id) REFERENCES branch(id) ON DELETE CASCADE

);

ON DELETE CASCADE – deletes the rows in branch\_supplier table when that particular id is deleted in branch table

ON DELETE SET NULL – sets the branch\_id column value to NULL when that particular id is deleted in branch table

JOINS

INNER JOIN

SELECT \* FROM person JOIN car ON person.car\_id = car.id;

LEFT JOIN

SELECT \* FROM person LEFT JOIN car ON person.car\_id = car.id;

Gives all records from PERSON combined with empty records on CAR table

SELECT \* FROM person LEFT JOIN car ON person.car\_id = car.id WHERE car.\* IS NULL;

car.\* IS NULL = ( car.id IS NULL AND car.make IS NULL AND car.model IS NULL car.price IS NULL )

Question : display the employees that earn more than avg salary of their respective department

SELECT e.ename, d.dname FROM emp e

JOIN dept d ON e.deptno = d.deptno

WHERE e.sal > ( SELECT AVG(sal) FROM emp WHERE deptno = e.deptno );

**TRIGGER**

- Gets executed immediately after certain action takes place

- USES

1. **Data Validation**: Ensuring specific conditions are met before data is committed.
2. **Auditing and Logging**: Tracking changes to important records for history or logging purposes.
3. **Enforcing Referential Integrity**: Managing data relationships in complex systems, even beyond basic foreign key constraints.
4. **Automated Calculations**: Updating derived fields automatically when data changes.

Downsides :

1. **Recursive Triggers**: Be careful with recursive triggers. For example, if you create a trigger on table A that updates table B, and then create a trigger on table B that updates A, this can lead to infinite loops. Some databases handle recursive triggers differently or provide options to limit recursion.
2. **Performance**: **Overusing triggers** can slow down performance. Triggers run as part of the transaction, so they can delay operations **if they involve heavy computations** or complex queries.
3. **Debugging**: Triggers can be challenging to debug since they execute automatically and behind the scenes. It’s often useful to log actions in a separate audit table to monitor trigger behavior.
4. **Database-Specific Syntax**: Trigger syntax and functionality can vary across different database management systems. Always refer to your specific database documentation (like PostgreSQL, MySQL, or SQL Server) for syntax and capabilities.

**Syntax** :

CREATE TRIGGER trigger\_name

AFTER | BEFORE INSERT | UPDATE | DELETE

ON table\_name

FOR EACH ROW

WHEN (-- condition)

BEGIN

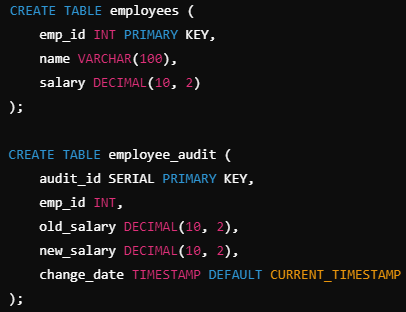
-- SQL statements to execute when the trigger fires

END;

-- condition examples :

1) OLD.salary IS DISTINCT FROM NEW.salary

2) OLD.stock > 0



CREATE TRIGGER audit\_salary\_changes

BEFORE UPDATE ON employees

FOR EACH ROW

WHEN (OLD.salary IS DISTINCT FROM NEW.salary)

BEGIN

INSERT INTO employee\_audit (emp\_id, old\_salary, new\_salary)

VALUES (OLD.emp\_id, OLD.salary, NEW.salary);

END;

Similarly

CREATE TABLE orders (

order\_id SERIAL PRIMARY KEY,

order\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TRIGGER set\_updated\_at

BEFORE UPDATE ON orders

FOR EACH ROW

BEGIN

NEW.updated\_at = CURRENT\_TIMESTAMP;

END;

CREATE TRIGGER audit\_balance\_changes

AFTER UPDATE ON bank\_account

FOR EACH ROW

WHEN (OLD.balance IS DISTINCT FROM NEW.balance)

BEGIN

INSERT INTO account\_transaction\_audit (account\_id, old\_balance, new\_balance, transaction\_type)

VALUES (OLD.account\_id, OLD.balance, NEW.balance,

CASE

WHEN NEW.balance > OLD.balance THEN 'Deposit'

ELSE 'Withdrawal'

END);

END;

CREATE TRIGGER check\_minimum\_salary

BEFORE INSERT OR UPDATE ON employee

FOR EACH ROW

BEGIN

IF NEW.designation = 'Manager' AND NEW.salary < 15000 THEN

RAISE EXCEPTION 'Managers must have a salary of at least 15,000';

ELSIF NEW.designation = 'Salesman' AND NEW.salary < 8000 THEN

RAISE EXCEPTION 'Salesmen must have a salary of at least 8,000';

END IF;

END;

CREATE TRIGGER delete\_employees\_on\_department\_delete

AFTER DELETE ON department

FOR EACH ROW

BEGIN

DELETE FROM employee WHERE dept\_id = OLD.dept\_id;

END;

**STORED PROCEDURE**

* Precompiled SQL statements (like functions in java)
* Can be invoked by DB users, apps, and other procedures

**USES :**

 **Performance**: Since they are precompiled, the database doesn’t need to re-parse and optimize the query each time. This can result in better performance, especially for complex logic.

 **Reusability**: You can write logic once and reuse it in multiple parts of your application. This reduces code duplication.

 **Maintainability**: When the business logic is encapsulated in stored procedures, you can change the logic in one place, rather than modifying it across multiple application layers.

 **Security**: Access to underlying data can be controlled. You can grant permissions to execute stored procedures without allowing direct access to the tables, improving data security.

 **Reduced Network Traffic**: Rather than sending multiple SQL queries from the application to the database, you can just call a stored procedure that performs a series of operations on the server.

**Syntax**:

CREATE OR REPLACE PROCEDURE example\_procedure(IN dept\_id INT, OUT avg\_salary DECIMAL)

LANGUAGE plpgsql

AS $$

BEGIN

-- Calculate average salary for the given department

SELECT AVG(sal) INTO avg\_salary

FROM emp

WHERE deptno = dept\_id;

END;

$$;

Example :

CREATE OR REPLACE PROCEDURE example\_procedure(IN dept\_id INT, OUT avg\_salary DECIMAL)

LANGUAGE plpgsql

AS $$

BEGIN

-- Calculate average salary for the given department

SELECT AVG(sal) INTO avg\_salary

FROM emp

WHERE deptno = dept\_id;

END;

$$;

CALL example\_procedure(10, avg\_salary);

CREATE OR REPLACE PROCEDURE get\_employees\_by\_department(IN dept\_id INT, OUT total\_salary DECIMAL)

LANGUAGE plpgsql

AS $$

BEGIN

-- Initialize total\_salary

total\_salary := 0;

-- Loop through employees in the specified department

FOR employee IN

SELECT eno, ename, sal FROM emp WHERE deptno = dept\_id

LOOP

-- Print employee details (for demonstration)

RAISE NOTICE 'Employee ID: %, Name: %, Salary: %', employee.eno, employee.ename, employee.sal;

-- Add employee's salary to total\_salary

total\_salary := total\_salary + employee.sal;

END LOOP;

END;

$$;

CALL get\_employees\_by\_department(10, total\_salary);

SERIAL AND SEQUENCES

SELECT \*FROM person\_id\_seq;

SELECT nextval(‘person\_id\_sequence’);

ALTER SEQUENCE test\_id\_sequence RESTART WITH 2;

UNIONS

UNION Operator

used to combine the results of two or more SELECT statements without returning any duplicate rows.

testdb=# SELECT EMP\_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID

UNION

SELECT EMP\_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT

ON COMPANY.ID = DEPARTMENT.EMP\_ID;

UNION ALL Operator

Same as UNION but also returns duplicate rows

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;