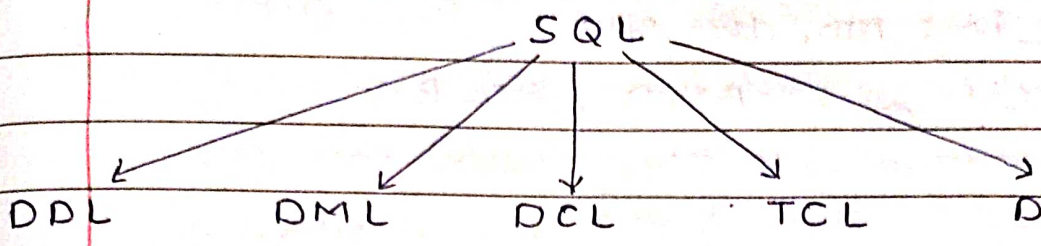
#SQL #Basics
#Intro Level

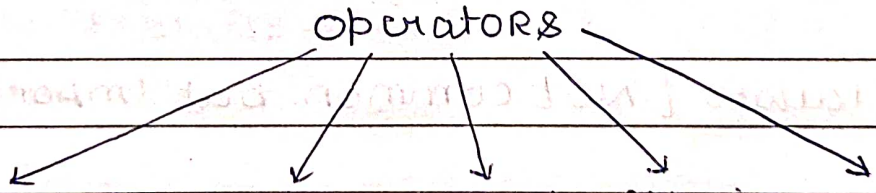
DOMS | Page No.
Date / /

SQL (Structured Query Language)



DDL	DML	DCL	TCL	DQL
CREATE	INSERT	GRANT	COMMIT	SELECT
ALTER	UPDATE	REVOKE	ROLLBACK	
DROP	DELETE		SAVE POINT	
TRUNCATE				
change schema OR change table structure	Modification & insertion into database	control access of data.	Manage transactions of database	used for querying database.

(Above are just basic commands)



Arithmetic	Comparision	Logical	Bitwise	String
+, -, *, /, %	=, >, <, <>, !=	AND	&, |, ^,	||, +
	<=, >=	OR	~, <<, >>	↓
		NOT		(Concatenation)

- Other operators:

IN - checks within a set of values.

BETWEEN - checks within a range of values.

LIKE - Search for specified pattern.

% : 0, 1 or multiple characters
_ : single character

\ - escape character

Basics
precedence

Example:

%ANN% : Janny, Flannted

_is : his, dis, 8is

escape character → _\%_ : A%A , A%B

Precedence in SQL

FROM > JOIN > ON > USING > WHERE

> GROUP BY > HAVING > SELECT

> DISTINCT > WINDOW FUNCTIONS >

> ORDER BY > LIMIT/OFFSET

> FETCH FIRST / NEXT > OFFSET

Clauses [Not common but Important]

1. DISTINCT : remove duplicates from result set.
2. LIMIT : return maximum no. of rows from result set. (N) – parameter
3. COUNT (column_name) : count no. of rows. in column.

* → also count null values.

1, 2, Etc. → doesn't reference any column, same as *

advance.
CASE

DOMS | Page No.
Date / /

(4.) CASE + WHEN + THEN + ELSE :

- switch and if/else of SQL.
- allows you to create expression, return different value based on evaluation of conditions.

Where to use: SELECT - return columns/value
WHERE - apply condition.
ORDER BY - sort result by cond.
HAVING - apply condition.

Examples:

(1.)
```
SELECT A, B,
    CASE COLC
        WHEN 'HR' THEN 'Human'
        WHEN 'IT' THEN 'Tech'
        ELSE 'OTHER'
    END AS COL_C_CASE
FROM TableA;
```

(2.)
```
Select id, name, department FROM employees
WHERE CASE
        WHEN department = 'HR' THEN
        WHEN  "  = 'IT' THEN id < 5
        ELSE TRUE                id < 3
    END;
```

(3.)
```
ORDER BY CASE
        WHEN department = 'HR' THEN 1
        WHEN  "  = 'IT' THEN 2
        ELSE 3
END;
```

basics

DOMS | Page No.
Date / /

(5.) Two ways to write SQL Table Queries.

using , (Comma)

```
SELECT id, name, d.d_name
FROM employees, department d
where id = d.did;
```

using JOIN

```
SELECT e.id, e.name, d.dname
FROM employees e
JOIN departments d ON e.id = d.id;
```

↑ I prefer JOIN method as it is more intituive, good and easy to grasp.

(6.) UNION & UNION ALL

UNION	UNION ALL
not include duplicate	also include duplicates.

Both used to combine two resultsets.

Conditions : → Datatype of column must match
→ No. of columns must match.

advance
Coalesce
aggregation functions

DOMS | Page No.
Date / /

⑦ COALESCE / NVL

- NVL is specific to oracle and takes only two arguments
- NVL (exp., replace_val)

return the first non-null value from a list of expressions.

COALESCE (Ex1, Ex2, Ex3 .... ExN)

eg:

```
SELECT id,
COALESCE (middlename, fname, lname) AS name
From employees;
```

Aggregate fn) — mostly used with Group By clause

perform calculation over multiple rows & return a single value.

COUNT() SUM() AVG() MIN() MAX() VARIANCE() STDDEV()

- applied over columns.
- with GROUP BY clause to calculate value according to groups.
- with HAVING / WHERE clause to filter out values.

advance
JOINS

DOMS | Page No.
Date / /

Joins

- they are used to combine two or more tables or result sets.

1. JOIN / INNER JOIN

return record having matching value in both tables.

2. LEFT OUTER / RIGHT OUTER / FULL OUTER

LEFT OUTER	RIGHT OUTER	FULL OUTER
INNER JOIN + all records from left table, non match value replace by NULL.	INNER JOIN + all records from right table, NON match value replace by NULL	INNER JOIN + all records from left table + all records from right table

3. CROSS JOIN

return cartesian product of two tables i.e., combine each row of left table with each row of right table

4. NATURAL JOIN

- It's INNER JOIN but ON condition is determined by SQL, itself. based on column names, primary keys etc.
- Not recommended, inconsistent results.

advance # JOINS

DOMS | Page No.
Date / /

5. SELF JOIN

- INNER JOIN but it is join with itself i.e.. same table.
- use case when you have to link one records of table with another records of same table.

Key words

1. JOIN ON
2. LEFT JOIN ON / RIGHT / FULL
3. CROSS JOIN ON
4. NATURAL JOIN ON
5. SELF JOIN ON

advance
subquery

DOMS | Page No.
Date / /

Sub Queries

1. Scalar
   - return single cell
   - Both are inter dependent.
   - not related to outer subquery.
   - use ( ) in where clause to compare multiple checks or conditions.

   Multiple
   - either multiple rows & columns or single column multiple rows.

2. Correlated
   - a subquery which is related to outer subquery.
   - for each outer query record, inner sub query is executed. like nested loops.
   - you can nest as many as queries you want but keep in mind it is computationally expensive.

- All subqueries can be used anywhere like in WHERE, JOIN, HAVING, SELECT.

(imp)
- In SELECT clause, you must do only scalar subqueries otherwise will throw error.
  OR Basically. coprelated query throw error in SELECT.

advance
CTE # with clause

DOMS | Page No.
Date / /

WITH clause / CTE (common table expressions)

one time

- It works like ^variables in other languages like CPP, python etc.
- It runs a query and store its result in RAM until the execution of that particular query.
- returns the result set.
- increase readiblity and speed.
- Time span : current Query Execution only.
- very helpful in recursive queries.

Syntax:

```
WITH CTE_NAME AS (
    -- query
) main query.
```

Eg:

```
WITH employeedepartments AS (
        SELECT e.id, e.name
        FROM employee AS e
        JOIN department d ON e.id = d.id
    )
    SELECT * FROM employeedepartments.;
```

- CTE are also used to write recursive queries.

advance
recursion.

DOMS | Page No.
Date / /

Recursive Queries

- Syntax:

```
WITH RECURSIVE cte_name AS (
    -- Base / Anchor member
    [ Query
    UNION OR UNION ALL
    -- Recursive member
    [ Query (inside FROM there must be
             cte_name as it reference
             same result set)
)
main Query.
```

- Anchor member
  - provide base result and starting point

- Recursive member
  - references CTE itself & iteratively adds rows to result set.
  - It also have termination condition, or query will end when no result is found.

Recursive query

DOMS | Page No.
Date / /

Eg:

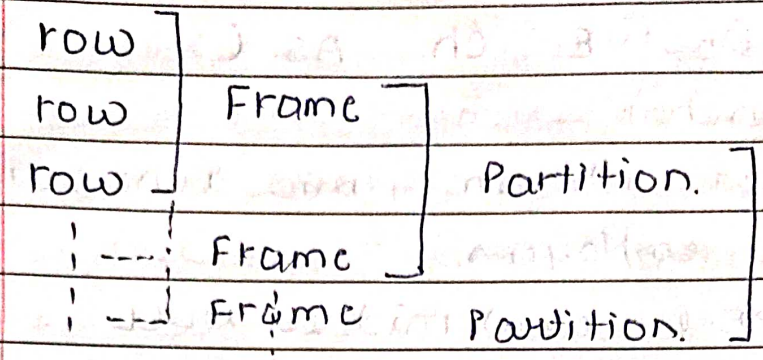
```
WITH RECURSIVE eh AS (
    -- Anchor
    SELECT id, name, mid, 1 AS level
    FROM employees
    WHERE (manager) mid IS NULL

    UNION ALL

    -- Recursive
    SELECT e.id, e.name, e.mid, eh.level+1
    FROM employees e
    JOIN eh AS eh ON e.mid = eh.id
)
SELECT id, name, mid, level
FROM eh;
```

- It is same like other recursion concept of other languages.

advance # window fcn
concept

DOMS | Page No.
Date / /

Window Functions.



- running rows makes a frame
  1 row = 1 frame
  2 row = frame 2
  3 row = frame 3 etc
  or it can be depend on condition.
  but the point is 0 — ith current frame
  0 — (i+1)th next frame.

- window fcns are used to perform calculations across a set of table (window) that are related to current row

OVER	PARTITION BY	ORDER BY
↓	↓	↓
Defines the window to operate on.	like a group by for window over which other fcns can be applied.	order rows with in partition.

window fns

DOMS | Page No.
Date / /

1. RANK()
   - assign rank to each row in partition.
   - assign same rank to duplicates but skip next values.

   Eg:

value:	10	20	20	43
RANK:	1	2	2	3

2. DENSE_RANK()
   - same as Rank fn) but do not skip values. It goes like 1 2 2 3

3. ROW_NUMBER()
   - assign each row a unique row number

4. Aggregate fns can also be used using OVER clause to apply them over a partition.

5. LAG() — access data from previous rows.
   - LAG(column, N)

   LEAD() — access data from next rows.
   - LEAD(column, N)

6. FIRST_VALUE(column)
   LAST_VALUE(column)
   NTH_VALUE(column, N)
   — respective value from each partition.

7. NTILE(N): SQL try to divide in equal partition.

window fun) # concept
frame clause

DOMS | Page No.
Date / /

8. CUME_DIST() | PERCENT_RANK()

↓ | ↓

It gives the cumulative distribution of row within set of rows. i.e. How much distribution current row till now holds. | It gives the relative rank of row within partition i.e. How many rows are below or higher relative to current row.

FORMULA: (No. of row with value ≤ current) / total rows. | (current ROW NO. - 1) / (total - 1)

Range: 0 < value ≤ 1 | 0 ≤ value ≤ 1

Note: In case of duplicate records, row of last record is considered.

9. FRAME Clause.

- FRAME word is not used.
- used at end of OVER clause as last arguments.

Syntax:

(A) BETWEEN (B) PRECEDING AND (C) Following.

window fns
frame clause

DOMS | Page No.
Date / /

(A) RANGE : take account of duplicates.
eg: last 2 : 5 [4 4] 3 2
ROWS : specific to rows only.
eg: last 2 : 5 [4 4] 3 2

(B) → unbounded : from starting of partition.
→ n : from last n of partition. (range/rows)

* (there are many more)

(C) unbounded : till end of current partition.
n : next n (range / rows) of current partition.
current row : till current row (omit word following)

WHY?

Because in fns like last_value()
frame considers the running rows.

	C1	C2	O	We want
Partition.	A	1	1	4
	B	2	2	4
	C	3	3	4
	D	4	4	4

F1 : A ∴ O = 1
F2 : AB ∴ O = 2
F3 : ABC ⋮
F4 : ABCD ↓

window fcts
concept

DOMS | Page No.
Date / /

as these fcts are applied frame by frame. even though they are applied over partition, as per saying, but executed frame by frame.

By default FRAME clause of LAST_VALUE is
* Range between unbounded preceding and current row.
∴ it will generate wrong results.
Hence, we need frame clause.

Examples:

1. Select id, name, salary,
   RANK() OVER (ORDER BY salary desc)as R
   FROM employees;

2. Select id, date, amount,
   AVG(amount) OVER (
   order By date
   ROWS between 2 preceding and current row
) as moving_avg
   FROM orders;

3. Select id, salary,
   PERCENT_RANK() OVER() as PR
   FROM employees.

window fun
window keyword

Note: if you have common window condition. i.e. inside over brackets. than you can omit common conditions and write it in window keyword fun

Example:

At end of query mention.

:

WINDOW W as (conditions)

URC.

Query.

```
Select *,
FIRST_value (Pn) OVER W as FV,
Last_value (Pn) OVER W as LV
From products.
where PC = 'Phone'
WINDOW W as ( partition by PC
              order by price desc
              range between 2 preceding
              and 2 following );
```

crosstab # pivot
rows to columns

DOMS | Page No.
Date / /

CROSSTAB / PIVOT in other DB

used to transform rows into columns.

In crosstab must add Unc to create extension.

Unc:

```
CREATE Extension IF NOT Exists tablefunc
```

Syntax:

```
Select *
FROM crosstab (
    'SELECT [row identifier], *
            [column to pivot], *
            [data column] *
     FROM [table]
     ORDER BY 1, 2'
) AS ct (
     [row identifier] [data type],
     [value 1] [data type],
     [value 2] [data type],
     :
);
```

important all 3

rows to columns

Example:

```
CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT *
FROM crosstab(
    'SELECT salesperson, month, sales
     FROM sales_data
     ORDER BY 1, 2'
) AS ct(salesperson text, Jan int,
        Feb int, Mar int);
```

this line works as magic

why?

because postgres just pick values and place it according to new columns Hence, we have to sort data accordingly.

views

DOMS | Page No.
Date / /

VIEWS

→ views are like CTE's but CTE associated to only single query. & SQL don't store schema of CTE as it is used in one query only.

they are like virtual tables.
- schema is stored
- permanent object
- always present until explicitly dropped
- will run every time, it is used
  it is like stored variable but evaluated everytime it is used.

* Basic DDL

syntax:
create view v_name
AS (query)

1. CREATE OR REPLACE
   if already present then
   - can't change column name, datatype, order.
   - add new column only at end.

2. ALTER VIEW
   change structure

3. DROP VIEW
   delete view

Note: (if table structure changed view needs to be updated as it stores previous version of table)

Updatable views
Concept

DOMS | Page No.
Date / /

Updatable views:

only views which

1. Created by only one table can be updated.
2. Not have DISTINCT, Aggregate fns
3. Not have GROUP BY.
4. Not have CTE/ WITH clause.
5. Not have JOINS
6. Not have subqueries.

Eg: Update VIEW_A
SET d = "Market"
WHERE id = 1;

- changes reflected back into table also.
- you can also add check option so that where condition must met before updation.

Eg:

Query
:
WHERE condition.
WITH CHECK OPTION;

materialized views
optimization.

DOMS | Page No.
Date / /

Materialized view.

use word MATERIALIZED before view
save both schema and resultset,
Hence,
performance very fast
but takes extra space.

→ we can manually refresh them
if new data is added to table.
CMD:
Refresh Materialized view MVA;

Syntax:

CREATE MATERIALIZED VIEW V_A AS
(
Query
)
;

A. Trigger

In SQL, a trigger is a special type of stored procedure that automatically executes or "fires" in response to specific events or changes in a database. Triggers are used to enforce business rules, automate tasks, and maintain data integrity.

Features

1. Automatic Execution:
   Triggers are executed automatically in response to events like `INSERT`, `UPDATE`, or `DELETE` on a table.
2. Event-Driven:
   They are triggered by specific database events and can be set to run before or after the event occurs.
3. Maintain Data Integrity:
   Triggers can enforce rules and constraints to ensure data consistency and integrity.
4. Complex Logic:
   They allow for the execution of complex logic that cannot be easily handled with constraints alone.

Types

1. **BEFORE** Triggers:
   Executed before an `INSERT`, `UPDATE`, or `DELETE` operation on a table.
2. **AFTER** Triggers:
   Executed after an `INSERT`, `UPDATE`, or `DELETE` operation on a table.
3. **INSTEAD OF** Triggers:
   Executed in place of the triggering operation, allowing for custom logic to replace the standard operation.

Syntax

```
CREATE FUNCTION trigger_function()
RETURNS TRIGGER
AS $$
BEGIN
   -- trigger logic here
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_name
BEFORE INSERT ON table_name
FOR EACH ROW
EXECUTE FUNCTION trigger_function();
```

Example

```
CREATE OR REPLACE FUNCTION log_salary_changes()
RETURNS TRIGGER AS $$
BEGIN
   -- Insert a record into salary_log for each update
   INSERT INTO salary_log (employee_id, old_salary, new_salary)
   VALUES (OLD.employee_id, OLD.salary, NEW.salary);

   -- Return the new row
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trigger_salary_changes
AFTER UPDATE OF salary
ON employees
FOR EACH ROW
EXECUTE FUNCTION log_salary_changes();
```

B. Stored Procedures

A stored procedure is a set of SQL statements that perform a specific task, which is stored in the database and can be executed by calling the procedure. Unlike functions, stored procedures do not return a value but can perform operations such as modifying data or managing transactions.

Syntax

```
CREATE PROCEDURE procedure_name (param1 type1, param2 type2, ...)
LANGUAGE plpgsql
AS $$

DECLARE
   Var1 DATATYPE;
   var2 DATATYPE;
   …
   var_n DATATYPE;

BEGIN
   -- Procedure logic here
END;
$$;
```

Example

```
CREATE PROCEDURE update_department_average(department_name VARCHAR)
LANGUAGE plpgsql
AS $$

DECLARE
   total_salary DECIMAL;
   employee_count INT;
   avg_salary DECIMAL;

BEGIN
   -- Calculate total salary and employee count for the department
   SELECT SUM(salary), COUNT(*)
   INTO total_salary, employee_count
   FROM employees
   WHERE department = department_name;

   -- Calculate average salary
   IF employee_count > 0 THEN
      avg_salary := total_salary / employee_count;
   ELSE
      avg_salary := 0;
   END IF;
```

```
    -- Update department table with average salary (assuming there's a department table)
    UPDATE departments
    SET average_salary = avg_salary
    WHERE department_name = department_name;
END;
$$;

-- Calling the procedure
CALL update_department_average('Engineering');
```

Parameters

1. IN Parameters: Used to pass values into the procedure.
2. OUT Parameters: Used to return values from the procedure to other procedure.
3. INOUT Parameters: Can be used to pass values in and return values out.

C. Functions

A function in SQL is a stored routine that returns a value. Functions can be used in SQL statements and can perform computations, return results, and be used in SELECT, WHERE, or ORDER BY clauses.

Syntax

```
CREATE FUNCTION function_name (param1 type1, param2 type2, ...)
RETURNS return_type
LANGUAGE plpgsql
AS $$

DECLARE
    var1DATATYPE;
    var2 DATATYPE;
    …
    var_n DATATYPE;

BEGIN
    -- Function logic here
    RETURN return_value;
END;
$$;
```

Example

```
CREATE FUNCTION get_total_salary(department_name VARCHAR)
RETURNS DECIMAL
LANGUAGE plpgsql
AS $$
DECLARE
    total_salary DECIMAL;
BEGIN
    -- Calculate total salary for the department
    SELECT SUM(salary)
    INTO total_salary
    FROM employees
    WHERE department = department_name;

    -- Return the total salary
```

```
    RETURN total_salary;
END;
$$;

-- Calling the function
SELECT get_total_salary('Engineering');
```

Difference between Function and Procedure

1. **Return Value:**
   Function: Returns a single value or a set of values (e.g., scalar, table).
   Procedure: Does not return a value directly; can use `OUT` parameters to return values.

2. **Usage in SQL Statements:**
   Function: Can be used within SQL statements (e.g., `SELECT`, `WHERE`).
   Procedure: Cannot be used within SQL statements; called using `CALL`.

3. **Purpose:**
   Function: Typically used for computations and data retrieval.
   Procedure: Used for performing operations, including data modification and managing transactions.

4. **Transactional Control:**
   Function: Limited control over transactions (cannot commit or roll back transactions within the function).
   Procedure: Can include explicit transaction control (e.g., `COMMIT`, `ROLLBACK`).

5. **Side Effects:**
   Function: Ideally should not have side effects (e.g., modifying database state).
   Procedure: Often used to perform operations that change the database state.