SQL

What is SQL and Why SQL

- ❖ Data is where everything begins. Eventually, every organization has a lot of data that needs to be stored, processed and inspected.
- ❖ For storing and cleaning purposes, databases exist. There are various types of databases for a variety of data available.
- ❖ RDBMS or Relational Database Management Systems, are data stored in the form of relations or tabular format. There are other non relational databases for image, graph or other data.
- ❖ In RDBMS, we need to access, modify or procure some data. For this we mainly use a query language, sequel. This in short is SQL.
- ❖ SQL has a certain syntax which needs to be followed, although it will be quite the same for different types of DBMS chosen.

Commands

There are mainly 5 types of commands:

1. DDL - Data Definition Language - Using it we can create or destroy tables. Main keywords used are:
    a. CREATE - generates the table according to columns defined.
    b. ALTER - to modify schema, add or remove constraints.
    c. DROP - removes the table and its schema and object
    d. TRUNCATE - keeps the schema but empties out the table
2. DML - Data Manipulation Language - used to load, modify and remove data from a database. Main keywords used are:
    a. INSERT - loads single or multiple data
    b. UPDATE - modifies the column values according to any constraints through the 'where' keyword

        c. DELETE - deletes all records from table, schema intact
3. DCL - Data Control Language -
4. TCL - Transaction Control Language -
5. DQL - Data Query Language - Used to display data.
    a. SELECT - helps find the relevant data.
    b. FROM - specifies the table
    c. WHERE - helps define the constraint
    d. JOIN - helps in joining more than one connected tables

Data Types

Mainly, the data types used in SQL are in the create query.

1. VARCHAR - alphabets, numbers, alphanumeric, special characters are all allowed
2. INT - whole numbers or integer values allowed
3. DATE - date values in any format like dd-mm-yyyy or mm-dd-yy allowed
4. FLOAT - decimal values allowed
5. BOOLEAN - either 'TRUE' or 'FALSE' values allowed.

Constraints

The constraints are also applied inside the create query, while defining the relational schema. This helps avoid integrity issues and maintain the tables in well defined format. These when defined in queries may have some values next to it like VARCHAR(20). Thai specifies the limit of characters allotted to it, or it cannot be more than 20 characters in this case.

1. CHECK - controls values being inserted into table
2. NOT NULL - never has null or empty values in that column
3. UNIQUE - values never repeat
4. PRIMARY KEY - it is a merge of unique and not null
5. FOREIGN KEY - helps in a parent child relationship, this key allows us to access child tables into parent ones

---

Example (DML and DCL):

```
CREATE TABLE STAFF (
      STAFF_ID INT,
      STAFF_TYPE VARCHAR(20),
      FIRST_NAME VARCHAR(20) NOT NULL,
      LAST_NAME VARCHAR(20) NOT NULL,
      GENDER VARCHAR(10) CHECK (GENDER IN ('M', 'F')),
       JOIN_DATE DATE,
       DEPT VARCHAR(10),
       DEPARTMENT_ID INT,
       CONSTRAINT PK_STAFF PRIMARY KEY(STAFF_ID),
       CONSTRAINT FK_DEPT FOREIGN KEY(DEPARTMETNT_ID)
);

INSERT INTO STAFF (STAFF_ID, STAFF_TYPE, FIRST_NAME, LAST_NAME, GENDER, JOIN_DATE,
DEPARTMENT_ID) VALUES (123, 'Data Scientist', 'Alice', 'Brown', 'F', TO_DATE('01/01/25',
'DD/MM/YYYY'), 533);

ALTER TABLE STAFF DROP COLUMN GENDER;
ALTER TABLE STAFF ALTER COLUMN JOIN_DATE TYPE VARCHAR(10);
ALTER TABLE STAFF RENAME COLUMN DEPARTMENT_ID TO DEPT_ID;
ALTER TABLE STAFF ADD CONSTRAINT UNQ_DEPT UNIQUE (DEPT_ID);

UPDATE STAFF SET STAFF_TYPE = Business Analyst',  DEPT = 'Marketing' WHERE DEPT_ID = 533;

TRUNCATE TABLE STAFF;
DROP TABLE STAFF;
DELETE FROM STAFF;
```

Operators and Aggregators

Aggregators and operators are helpful even in writing subqueries.
There are many operators that can be used to fetch the required results:

1. Comparison: >, <, >=, <=, =, (!= or <>, both are same)
2. Arithmetic: +, -, /, *, ^, %
3. Clauses: GROUP BY, HAVING, ORDER BY, DISTINCT, LIMIT, CASE, ALIAS, ASC, DESC
4. Logical: AND, OR, NOT, IN, BETWEEN, LIKE, ALL, ANY, UNION, UNION ALL

Aggregate functions help organize the data.:

1. SUM - adds up all values
2. AVG - gives average of all values
3. COUNT - gives frequency or count of those values
4. MIN - gives minimum of all values
5. MAX - gives maximum of all values

Example (DQL):

```
SELECT * FROM STAFF;
SELECT * FROM STAFF WHERE DEPT = 'Marketing';
SELECT * FROM STAFF WHERE DEPT <> 'Marketing';
SELECT  s.nameA, s.nameB FROM STAFF AS s JOIN HR_TEAM AS h ON s.STAFF_ID = h.EMP_ID;
SELECT * FROM HR_TEAM WHERE AGE > 30 AND GENDER = 'F';
SELECT * STAFF WHERE FIRST_NAME LIKE 'S%';
SELECT * FROM STAFF ORDER BY AGE DESC;
SELECT * FROM STAFF ORDER BY JOIN_DATE ASC;
SELECT * FROM STAFF WHERE STAFF_TYPE IN ('BA', 'MA', 'CA');
SELECT DISTINCT FIRST_NAME FROM STAFF LIMIT 5;
SELECT * FROM STAFF WHERE AGE BETWEEN 20 AND 30 ORDER BY AGE;
SELECT
CASE WHEN AGE <= 20 THEN 'Young Brains'
WHEN AGE BETWEEN 20 AND 40 THEN 'Mature Adults'
WHEN AGE > 40 THEN 'Seniors'
END AS JOB_DESCRP;
SELECT DISTINCT (FIRST_NAME||' '||LAST_NAME) AS FULL_NAME FROM STAFF;
UNION ALL
SELECT DISTINCT JOIN_DATE FROM STAFF;
SELECT AVG(SALARY) FROM STAFF GROUP BY DEPT_ID;
SELECT DEPT_ID, SALARY FROM STAFF GROUP BY DEPT_ID HAVING AVG(SALARY) > 5000;
SELECT FIRST_NAME, LAST_NAME FROM STAFF WHERE STAFF_ID IN (SELECT EMP_ID FROM VACATION WHERE
VAC_DAYS < 10);
SELECT FIRST_NAME FROM STAFF WHERE SALARY > ALL(SELECT SALARY FROM STAFF WHERE DEPT_ID = 533);
SELECT FIRST_NAME FROM STAFF WHERE SALARY > ANY(SELECT SALARY FROM STAFF WHERE DEPT_ID = 533);
```

Joins

The joins help that related tables to work together. There are different types of joins:

1. INNER JOIN / JOIN - this simply gives the output as the common tuples in both the tables, according to the constraint specified using 'ON' keyword. If nothing is specified, it checks the same tuples throughout all the common columns.
2. LEFT JOIN / LEFT OUTER JOIN - this gives common tuples, along with all records of the left table, even though they do not match with the common constraint values.
3. RIGHT JOIN / RIGHT OUTER JOIN - this gives common tuples, along with all records of the right table, even though they do not match with the common constraint values.
4. FULL JOIN / FULL OUTER JOIN - this gives common tuples, along with all records of both tables, even though they do not match with the common constraint values.

5. CROSS JOIN - return the cartesian product of tuples from tables. There is no need for any constraint or common value definition. It just matches records of the left table with right, and prints all the combinations. So, total tuples (A x B) = m*n if table A has m and B has n tuples.
6. NATURAL JOIN - it is similar to inner join, but not the same as it. Here, there is no need to specify the common columns using ON. SQL decides this factor itself. It depends on the same name of the columns. If the columns of the same name exist, they all become the inner join condition cases.
7. SELF JOIN - it is a join with the table itself based on defined conditions. No specific keyword is used for this. Simple 'join' is used with the same tables on both left and right sides.

Example (joins): tables used -
EMPLOYEE (e_id, salary, e_name, dept_id, m_id)
DEPARTMENT (dept_id, dept_name)
MANAGER (m_id, m_name, dept_id)
PROJECTS (p_id, p_name, tm_id)
COMPANY (c_id, c_name, c_loc)
FAMILY (member_id, name, age, guardian_id)

Question: Select employees and their respective department names.
SELECT E.e_name, D.dept_name FROM EMPLOYEE E INNER JOIN DEPARTMENT D ON E.dept_id = D.dept_id; *fetches only common tuples*
Question: Fetch all employees with their department names.
SELECT E.e_name, D.dept_name FROM EMPLOYEE E LEFT JOIN DEPARTMENT D ON E.dept_id = D.dept_id; *fetches all columns of left table along with common column of right table, unknown values filled with null.*
Question: Fetch all departments with names of employees from them.
SELECT E.e_name, D.dept_name FROM EMPLOYEE E RIGHT JOIN DEPARTMENT D ON E.dept_id = D.dept_id; *fetches all columns of right table along with common column of left table, unknown values filled with null.*
Question: Fetch all employee names, with their dept, manager and project names as well.

```
SELECT E.e_name, D.dept_name, M.m_name, P.p_name
FROM EMPLOYEE E
        LEFT JOIN DEPARTMENT D ON D.dept_id = E.dept_id
        LEFT JOIN MANAGER M ON M.m_id = E.m_id
        LEFT JOIN PROJECTS P ON P.tm_id = E.e_id;
```

Question: Fetch all employees and their respective department names, make sure all department ids are fetched as well.

SELECT E.e_name, D.dept_name FROM EMPLOYEE E FULL JOIN DEPARTMENT D ON D.dept_id = E.dept_id; *fetches all columns of both tables with required columns, filling unknown with null.*
Question: Fetch all employee names with their company name and location.
SELECT E.e_name, C.c_name, C.c_loc FROM EMPLOYEE E CROSS JOIN COMPANY C;
Question: Fetch the names of guardians of members from the family table.
SELECT child.name, parent.name
FROM FAMILY AS child JOIN FAMILY AS parent ON child.guardian_id = parent.member_id;

Subqueries

This is helpful when the given query can be broken into more than one query.
Example: fetch employee details whose salary is greater than the average salary of all employees. The query would be

```
SELECT * FROM EMPLOYEES WHERE SALARY >= (SELECT AVG(SALARY) FROM EMPLOYEES);
```

The inner query is also called subquery and outer one is called main query using inner one to get results. Sometimes inner may rely on outer query, sometimes it may not. There are mainly three types of subqueries:

1. Scalar subqueries: always returns output of one row and one column, like the one in example above.
2. Multiple row/col subqueries: there are two types in this
    a. Multiple column - one row, many columns. This involves use of 'IN' for matching the filter condition with all the results.
       Example: Employees earning the highest salary in each department.

```
SELECT * FROM EMPLOYEES WHERE (dept,salary) IN (
      SELECT dept, MAX(salary) FROM EMPLOYEE GROUP BY DEPT
);
```

    b. Multiple row - one column, many rows
       Example: Departments with no employees.

```
SELECT * FROM DEPT WHERE dept NOT IN (
      SELECT DISTINCT dept IN EMPLOYEES
);
```

3. Correlated subqueries: this involves those subqueries that are related to the outer or the main query.
   Example 1: Find employees in each department who earn more than the average salary of their respective department.

```
SELECT * FROM EMPLOYEE e1 WHERE SALARY >= (
      SELECT AVG(SALARY) FROM EMPLOYEE e2 WHERE e1.dept = e2.dept
);
```

Main difference in this type of subquery is that it is executed each time for each tuple in the outer query. Other subqueries execute only once first then use it to find the data. But in correlated ones, it requires information from the main query too, hence executes multiple times.

Example 2: Find the departments who do not have any employees:

```
SELECT * FROM DEPT d WHERE NOT EXISTS (
SELECT 1 FROM EMPLOYEE e WHERE d.dept_name = e.dept_name
);
```

4. Nested subqueries: It is like a subquery inside a subquery, and so on.
   Example: fetch stores whose sales are more than average sales across all stores. Given that SALES (store_id, store_name, product, quantity, price).

```
SELECT * FROM (
      SELECT store_name, sum(price) as total_sales FROM SALES GROUP BY store_name
) sales1 JOIN (
      SELECT AVG(total_sales) FROM (
             SELECT store_name, sum(price) as total_sales FROM SALES
             GROUP BY Store_name;
      )
) sales2 ON sales1.total_sales > sales2.sales;
```

This is not feasible as the same subquery is getting repeated. This creates a use for the ***WITH clause***. Same query modified:

```
WITH sales AS (
      SELECT store_name, sum(price) as total_sales FROM SALES  GROUP BY store_name
)
SELECT * FROM sales JOIN (
      SELECT AVG(total_sales) AS avg_sales FROM sales
) avgsalestable
ON avgsalestable.avg_sales = sales.total_sales;
```

Different clauses in SQL where subqueries are allowed: SELECT, FROM, WHERE, HAVING.

Example: Fetch all employees and add a remark for those earning more than average.

```
SELECT *, (
      CASE WHEN salary > AVG(salary) THEN 'higher than avg'
      ELSE NULL
      END
) AS remarks FROM EMPLOYEE;
```

But this is inefficient in terms of query running time. Hence, we can have different and optimal ways always with joins.

Example 1: Fetch stores which have sold more units than average units of all stores. Given that SALES (store_id, store_name, product, quantity, price).

```
SELECT store_name, SUM(quantity)
FROM SALES
GROUP BY store_name
HAVING SUM(quantity) > (SELECT AVG(quantity) FROM SALES);
```

Other commands where we can use subqueries: INSERT, UPDATE, DELETE, QUERY.
Example 2: Insert distinct employee id and dept name into emp_history table.

```
INSERT INTO EMP_HISTORY
SELECT E.emp_id, D.dept_name
FROM EMPLOYEE E JOIN DEPARTMENT D ON E.dept_id = D.dept_id
WHERE NOT EXISTS(
      SELECT 1 FROM EMP_HISTORY EH WHERE EH.emp_id = E.emp_id
);
```

Example 3: Give 10% increment to employees of Bangalore based on maximum salary earned by employees in each department. Use employees from emp_history table only.

```
UPDATE EMPLOYEE E
SET salary = (
      SELECT MAX(salary) * 1.1 FROM EMP_HISTORY EH WHERE EH.emp_id = E.emp_id)
WHERE E.dept_name IN (SELECT dept_name FROM DEPARTMENT
                      WHERE dept_loc = 'Bangalore')
AND E.emp_id IN (SELECT emp_id FROM EMP_HISTORY);
```

Example 4: Delete all departments who do not have any employees.

```
DELETE FROM DEPARTMENT
```

```
WHERE dept_name IN (
      SELECT dept_name FROM DEPARTMENT D WHERE NOT EXISTS (
            SELECT 1 FROM EMPLOYEE E WHERE E.dept_name = D.dept_name
      )
);
```

WITH Clause

It is not necessary that a query function should start only with a SELECT keyword. It can also have WITH in the beginning.
Example: Fetch employees with salaries more than the average salary of all employees.

```
WITH avg_salary (insert_custom_cols_or_can_be_excluded) AS
      (SELECT AVG(salary) FROM EMPLOYEE)
SELECT * FROM EMPLOYEE WHERE salary > avg_salary;
```

The main advantage of using this clause is that it makes the query readable, easy for maintaining and debugging. Further, it reduces the query time complexity.

Window Function

The syntax of queries using window function is a bit different from the normal ones. The main functions used here are rank, dense_rank, lead, lag, row_number.
Example: Fetch the employee details along with their maximum department salary.

```
SELECT E.* ,
MAX(salary) OVER() AS max_salary
FROM EMPLOYEE E;
```

The 'OVER' clause will not treat max(salary) as an aggregate function, but as a window function over that case. Here, no case is mentioned so it takes the MAX(salary) of all employees by default, one window for all records. To go by departments:

```
SELECT E.* ,
MAX(salary) OVER( PARTITION BY dept_name ) AS max_salary
FROM EMPLOYEE E;
```

Here, the window is now created for every distinct value in the ‘dept_name’ column. Results will have a column max_salary with max salary of that respective department. Other aggregate functions such as MAX, MIN, COUNT, SUM or AVG can also be used.

Some unique window functions are as below:

1. ROW_NUMBER: assigns a unique value to each row of the table.

   Example: Assign a simple row number to index the table.

```
SELECT E.*,
ROW_NUMBER() OVER() AS rn
FROM EMPLOYEE E;
```

   Example: Assign row numbers according to the department.

```
SELECT E.*,
ROW_NUMBER() OVER(PARTITION BY dept_name) AS rn
FROM EMPLOYEE E;
```

   Example: Fetch first two employees who joined in each of the departments, assume that order assigned by employee_id is same as joining_date order.

```
SELECT * FROM (
      SELECT E.*,
      ROW_NUMBER() OVER(PARTITION BY dept_name ORDER BY emp_id) AS rn
      FROM EMPLOYEE E
) x
WHERE x.rn < 3;
```

2. RANK: useful when you need to assign a rank to rows within a dataset based on a specific criteria. The main difference between rank and row number is that, row number is like an index. It is unique by row. But rank may repeat given the same condition appears again. After a duplicate rank is encountered, the next rank is skipped. For example if scores are 100, 50, 50, 30 then their respective ranks are 1,2,2,4.

   Example: Fetch top 3 employees in each department earning max(salary).

```
SELECT * FROM (
      SELECT E.*,
      RANK() OVER( PARTITION BY dept_name ORDER BY salary DESC) AS rnk
      FROM EMPLOYEE E
) x
```

```
WHERE x.rnk < 4;
```

3. DENSE_RANK: There is only a small difference between rank and dense_rank, which is that, even after encountering duplicate values, it won't skip any values. So, for the same scores 100, 50, 50, 30 we would have the respective dense_ranks as 1,2,2,3. Structure will be the same as usual.
   Example: Fetch top 3 employees in each department earning max(salary), use dense ranks.

```
SELECT * FROM (
       SELECT E.*,
       DENSE_RANK() OVER( PARTITION BY dept_name ORDER BY salary DESC) AS dns_rnk
       FROM EMPLOYEE E
) x
WHERE x.dns_rnk < 4;
```

4. LAG: it helps show the records of the previous row. There are arguments to it as follows: LAG(col_name, previous_nth_row, default_val).
   Example: Query to display if the salary of an employee is higher, lower or equal to the previous employee in their respective departments.
   Below query fetches the records of the previous one by lagging two steps behind the usual and gets attached as a column to the present one. If value doesn't exist, it adds 0 as mentioned in the argument. If not explicitly mentioned, it inserts [null].

```
SELECT E.*,
LAG(salary, 2, 0) OVER( PARTITION BY dept_nameORDER BY emp_id ) AS prev_emp_sal
FROM EMPLOYEE E;
Using this as a subquery we compare as below:
SELECT E.*,
LAG(salary, 1, 0) OVER( PARTITION BY dept_nameORDER BY emp_id ) AS prev_emp_sal
CASE  WHEN E.salary > E.prev_emp_sal THEN 'Higher than prev sal'
      WHEN E.salary < E.prev_emp_sal THEN 'Lower than prev sal'
      WHEN E.salary = E.prev_emp_sal THEN 'Equal to prev sal'
END salary_range
FROM EMPLOYEE E;
```

5. LEAD: this is pretty similar to LAG. Instead of previous values, we will be using the next value. Same syntax and format goes for its arguments as well.
   Example: Query to display if the salary of an employee is higher, lower or equal to the next employee in their respective departments.

```sql
SELECT E.*,
LEAD(salary, 1, 0) OVER( PARTITION BY dept_nameORDER BY emp_id ) AS next_emp_sal
CASE  WHEN E.salary > E.next_emp_sal THEN 'Higher than next sal'
      WHEN E.salary < E.next_emp_sal THEN 'Lower than next sal'
      WHEN E.salary = E.next_emp_sal THEN 'Equal to next sal'
END salary_range
FROM EMPLOYEE E;
```

6. FIRST_VALUE: fetches the first value in the defined partition.
   Example: Fetch the most expensive product under each category. Given the table:
   PRODUCT (product_category, brand, product_name, price).

```sql
SELECT *,
FIRST_VALUE(product_name) OVER ( PARTITION BY product_category ORDER BY price DESC)
AS most_expensive
FROM PRODUCT;
```

7. LAST_VALUE: similar to first value, instead of first, it fetches last value from the defined partition.
   Example: Fetch the most expensive product under each category. Given the table:
   PRODUCT (product_category, brand, product_name, price).

```sql
SELECT *,
LAST_VALUE(product_name) OVER ( PARTITION BY product_category ORDER BY price) AS
most_expensive
FROM PRODUCT;
```

   This is the same query, yet we use the last value, as we now order by price in ascending manner. But this might not work well, due to something called predefined frames.

8. FRAMES: the frame clause is used inside the OVER clause and after you define all the required constraints. Suppose we want the least expensive product now.

```sql
SELECT *,
LAST_VALUE(product_name) OVER (
      PARTITION BY product_category
      ORDER BY price DESC
      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS least_expensive
FROM PRODUCT;
```

This red bold statement is the default frame, used by SQL even if we do not mention it. It means that it considers the rows between the unbounded preceding (the very first record) and current one. This will not be a problem in first_value, since the first row is first for (1,2) rows, and (1,2,3) and (1,2,3,4) rows as well. So to remove this problem we use 'UNBOUNDED FOLLOWING' instead of 'CURRENT ROW'.

```
SELECT *,
LAST_VALUE(product_name) OVER (
      PARTITION BY product_category
      ORDER BY price DESC
      RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
) AS least_expensive
FROM PRODUCT;
```

This frame definition can also be written as '**ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**'. This will show a difference when there are duplicate records for the last value benign fetched. If there are 3 products with the same min price, then the 'current row' in frame with 'row' will give each time a different last_value. But with 'range', we get the same for all of them. It considers the duplicates.
To sum up, the 'row' looks at the last row value, while 'range' looks at the last same value, even including duplicates.

9. ANOTHER WAY TO DEFINE WINDOW FUNCTIONS:
   Say we have a query as follows:

```
SELECT *,
FIRST_VALUE(product_name) OVER (
      PARTITION BY product_category ORDER BY price DESC
      RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
) AS most_expensive
LAST_VALUE(product_name) OVER (
      PARTITION BY product_category ORDER BY price DESC
      RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
) AS least_expensive
FROM PRODUCT;
```

We rewrite it as:

```sql
SELECT *,
FIRST_VALUE(product_name) OVER w1 AS most_expensive
LAST_VALUE (product_name) OVER w1 AS least_expensive
FROM PRODUCT
WINDOW w1 AS (
       PARTITION BY product_category ORDER BY price DESC
       RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
);
```

10. NTH_VALUE: this will fetch value from position specified. Similar to first and last, it will have another argument, rest remains the same. If that nth value doesn't exist then it returns [null]. Specifying a proper frame is important here, else results will vary with different frames.
    Example: Extract second most expensive product under each category.

```sql
SELECT *,
NTH_VALUE(product_name, 2) OVER w1 AS second_most_expensive
FROM PRODUCT
WINDOW w1 AS (
       PARTITION BY product_category ORDER BY price DESC
       RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
);
```

11. NTILE: used to group together a set of data within the defined partition, then place it into certain buckets. The buckets created will be equal-frequency or will have an equal number of records. The precedence in case of unequal buckets will be first, then second and so on.
    Example: Write a query to segregate expensive phones, mid range, and cheap phones.

```sql
SELECT product_name,
CASE  WHEN x.buckets = 1 THEN 'Expensive'
      WHEN x.buckets = 1 THEN 'Mid-Range'
      WHEN x.buckets = 1 THEN 'Cheap'
END bucket_name
FROM (
     SELECT *,
     NTILE(3) OVER (ORDER BY price DESC) AS buckets
     FROM PRODUCTS WHERE product_category = 'Phone'
) x;
```

12. CUME_DIST: stands for cumulative distribution. It identifies the distribution percentage of each record with all rows within a result set or partition. The values will be in range of (0,1]. The formula is → (current_row_num / total_num_of_rows). Duplicate records will have the same distribution with current_row_num as row_num of last duplicate row. Say, rows (1,2,3,4,5,6,7) have values (10,20,30,40,50,50,70). Then for 50, cum_dist = 6/7 and for 70, cum_dist = 7/7 = 1.
    Example: Fetch all the products that are constituting 30% of data in the products table based on price.

```
SELECT product_name, ( cum_dist||'%' ) AS cum_dist FROM (
      SELECT *,
      CUME_DIST() OVER(ORDER BY price DESC) AS cum_dist
      FROM PRODUCTS
) x
WHERE x.cum_dist <= 0.3;
```

    This fetches a value into the column cum_dist. If we convert it into a percentage format, it gives distribution of that record.

```
SELECT *,
CUME_DIST() OVER(ORDER BY price DESC) AS cum_dist
ROUND(CUME_DIST() OVER(ORDER BY price DESC)::NUMERIC * 100, 2) AS cum_dist_percentage
FROM PRODUCTS;
```

13. PERCENT_RANK: it is another way of representing ranks as percentages. It is similar to cume_dist() and also lies in the range of [0,1].
    The formula is → ((current_row_num - 1) / (total_num_of_rows - 1))
    Example: Identify how much percentage more expensive is product 'xyz' when compared to all products.

```
SELECT product_name, perc_rnk FROM (
      SELECT *,
      PERCENT_RANK() OVER( ORDER BY price ) AS perc_rnk
      FROM PRODUCT
) x
WHERE x.product_name = 'xyz';
```

Views

Views are database objects created over sql queries. They don't store any data and can be treated as a virtual table. When we are sharing things with external parties, we would not like them to know the internal and confidential table schemas and structure for privacy purposes and also due to less expertise. Hence we use the views.

1. They help reduce complexity of SQL queries
2. They help in maintaining privacy and confidentiality of organization.

Example-data: Use the following tables:
CUSTOMER (cid, cname, phone, email, address)
PRODUCT (pid, pname, brand, price)
ORDER (oid, pid, quantity, cid, discount, date)
Fetch the order summary to be given to the clients/vendors.
Different commands used are as follows:

1. CREATE VIEW: Generates the view. Remember that after creating the view, altering the tables will not change anything in the view. It will remain captured as it was when the view was created.

```
CREATE VIEW order_summary AS
SELECT O.oid, O.date, P.pname, C.cname, (P.price * O.quantity) - ((P.price *
O.quantity)*discount::float/100) AS cost
FROM CUSTOMER C
JOIN ORDER ON O.cid = C.cid
JOIN PRODUCT ON P.pid = O.pid;

SELECT * FROM order_summary;
```

2. CREATE ROLE and GRANT: For the clients to view, we create a user and grant access to that view (only readable).
   CREATE ROLE Alice LOGIN PASSWORD 'Alice';
   GRANT SELECT ON order_summary TO Alice;

3. CREATE OR REPLACE VIEW: this will rewrite the underlying query of the existing view, or creates a new one, if it doesn't exist. You cannot change the order of columns, column data types and column names. But, a new column can be added at the end of all columns. It will work with no issues and will replace the old one.
   Further this helps in refreshing the view as well. When tables are altered or new values are inserted into it, this command helps update the views' schemas.

```
CREATE OR REPLACE VIEW order_summary AS
SELECT O.oid, O.date, P.pname, C.cname, (P.price * O.quantity) - ((P.price *
O.quantity)*discount::float/100) AS cost
FROM CUSTOMER C
```

```
JOIN ORDER ON O.cid = C.cid
JOIN PRODUCT ON P.pid = O.pid;
```

4. ALTER VIEW: it can rename the columns.

```
ALTER VIEW order_summary RENAME COLUMN oid AS order_id;
```

5. DROP VIEW: deletes that view from schema.

```
DROP VIEW order_summary;
```

6. UPDATE: In order to create updatable views:
    - Views should be created from a single table/view only.
    - Cannot have a DISTINCT clause in it. Views can be created with a distinct clause, but cannot be updated in such cases.
    - Cannot have GROUP BY clause in it. Views can be created with a group by clause, but cannot be updated in such cases.
    - Cannot have WITH clause or WINDOW FUNCTIONS.

```
CREATE OR REPLACE VIEW expensive AS
SELECT * FROM PRODUCT WHERE price > 100;

UPDATE expensive SET pname = 'ABC', brand = 'XYZ' WHERE pid = 100;
```

7. WITH CHECK: it allows to constrain the insertion into the views. Suppose a view exists for products of brand A. Then it shouldn't insert values with a brand as B or C. This can be achieved through checks in our create view. This checks the where conditions, then only allows the insertion.

```
CREATE OR REPLACE VIEW a_products AS
SELECT * FROM PRODUCT WHERE brand = 'A'
WITH CHECK OPTION;
```

Materialized View

If you have a query that takes a lot of time to execute, we create a materialized view on SQL query. It is usually used in tables with a humongous amount of data. This concept is general across all RDBMS systems. It is a database object created over SQL query. Unlike the normal views, the materialized view:

1. Stores the SQL query used to create that view
2. Stores data returned from the SQL query

So, in short, it doesn't go and execute the query each and every time it is called. It stores the data for once and all the moment it is created. Normal views are not like that and execute the queries each time they are called.
How to create a materialized view and show it:

```
CREATE MATERIALIZED VIEW mv_name AS
SELECT id, AVG(salary), COUNT(*) FROM large_table_name GROUP BY some_col;

SELECT * FROM mv_name;
```

One point to note is that it is not an auto update, if we alter the table, the changes won't reflect in the materialized view. In order to update it, we need to manually do a refresh. This is useful when the refresh rate is large, like a monthly or yearly thing.
REFRESH MATERIALIZED VIEW mv_name;

Recursive Queries

Recursive queries are helpful in executing queries without built-in functions. Sometimes we might need to use UNION ALL for some databases, or UNION works well. Similarly, the RECURSIVE keyword is also an option in some cases.
Example: Display numbers 1 to 10 without using built in functions.

```
WITH RECURSIVE numbers AS (
      SELECT 1 AS n
      UNION
      SELECT n+1 FROM numbers WHERE n < 10;
)
SELECT * FROM numbers;
```

Example: Find hierarchy of employees working under a manager 'Alice'. Given table EMP (emp_id, name, manager_id, salary, designation). Add a level attribute in order to distinguish the hierarchy.

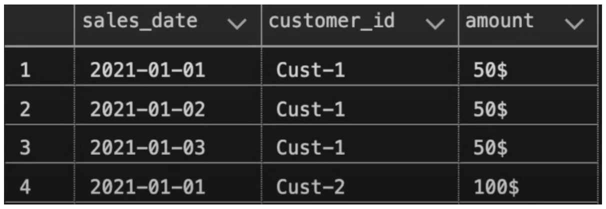
```
WITH RECURSIVE emp_hierarchy AS (
      SELECT *, 1 AS level FROM EMP WHERE name = 'Alice'
      UNION
      SELECT *, H.level+1 FROM emp_hierarchy H JOIN EMP E ON E.manager_id = H.emp_id;
)
SELECT * FROM emp_hierarchy;
```

Example: Find hierarchy of managers for a given employee, say 'Bob'.

```
WITH RECURSIVE emp_hierarchy AS (
      SELECT *, 1 AS level FROM EMP WHERE name = 'Bob'
      UNION
      SELECT *, H.level+1 FROM emp_hierarchy H JOIN EMP E ON H.manager_id = E.emp_id;
)
SELECT * FROM emp_hierarchy ORDER BY level DESC;
```

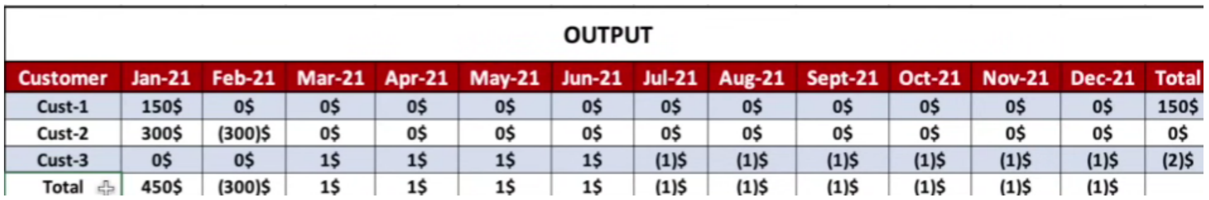
Pivots and Crosstabs

Generally pivots and crosstabs are useful when we want to convert the rows to columns and vice-versa. Only a syntax difference is noted between them. Pivots only are supported in Oracle and SQL server, Crosstabs only in PostgreSQL, and we have to use CASE if we are having a MySQL database for this purpose.
To explain this concept we have used table SALES (date, customer_id, amount). What we want to do is to rotate it and make columns as different months of the year from Jan to June. Say, only three unique customer_ids are existing in the database spanning over 6 months, so rows will be three, one for each id, and cells will contain the amount summed up according to each month.



	sales_date	customer_id	amount
1	2021-01-01	Cust-1	50$
2	2021-01-02	Cust-1	50$
3	2021-01-03	Cust-1	50$
4	2021-01-01	Cust-2	100$

⬇️



OUTPUT

Customer	Jan-21	Feb-21	Mar-21	Apr-21	May-21	Jun-21	Jul-21	Aug-21	Sept-21	Oct-21	Nov-21	Dec-21	Total
Cust-1	150$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$	150$
Cust-2	300$	(300)$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$
Cust-3	0$	0$	1$	1$	1$	1$	(1)$	(1)$	(1)$	(1)$	(1)$	(1)$	(2)$
Total	450$	(300)$	1$	1$	1$	1$	(1)$	(1)$	(1)$	(1)$	(1)$	(1)$	

Sample format looks like this:

```
SELECT * FROM (
      BASE-QUERY-HERE
) AS alias-name PIVOT (
      AGGREGATE FUNCTION
      FOR COLUMN-VALUE IN ( ' ', ' ' , ' ' )
) AS alias-name;
```

The base query should be used for transforming the date and modifying the amount by removing the $ sign. We need to keep in mind that usually base queries have at least 3 columns, first one as a unique identifier(customer_id), second one as a list of categories of new columns (months), then last one as value to be loaded in the cells (amount).

```
SELECT * FROM (
      SELECT customer_id AS customer,
      CAST(FORMAT(date, 'MMM-yy') AS INT) AS date,
      REPLACE(amount, '$', '') AS amount
      FROM SALES
) AS sales_data PIVOT (
      SUM(amount)
      FOR date IN ( [Jan-25], [Feb-25] , [Mar-25], [Apr-25], [May-25], [Jun-25] )
) AS pivot_table;
```

Suppose now we want a total column along with three customers, giving a total of each month. Then we use the same pivot format with another pivot query and union it. In the second, we use a common unique identifier to sum over all customers.

```
SELECT * FROM (
      SELECT customer_id AS customer,
      CAST(FORMAT(date, 'MMM-yy') AS INT) AS date,
      REPLACE(amount, '$', '') AS amount
      FROM SALES
) AS sales_data PIVOT (
      SUM(amount)
      FOR date IN ( [Jan-25], [Feb-25] , [Mar-25], [Apr-25], [May-25], [Jun-25] )
) AS pivot_table
UNION
SELECT * FROM (
      SELECT 'Total' AS customer,
      CAST(FORMAT(date, 'MMM-yy') AS INT) AS date,
      REPLACE(amount, '$', '') AS amount
      FROM SALES
) AS sales_data PIVOT (
      SUM(amount)
      FOR date IN ( [Jan-25], [Feb-25] , [Mar-25], [Apr-25], [May-25], [Jun-25] )
) AS pivot_table;
```

Next, we want total across each customer. Last one was a total row. Now we want a total column. For this, we need to first replace NULLs with zeros. For that, we use coalesce function and finally we put this big query into a with clause and use it easily.

```
WITH pivot_data AS (
SELECT * FROM (
       SELECT customer_id AS customer,
       CAST(FORMAT(date, 'MMM-yy') AS INT) AS date,
       REPLACE(amount, '$', '') AS amount
       FROM SALES
) AS sales_data PIVOT (
       SUM(amount)
       FOR date IN ( [Jan-25], [Feb-25] , [Mar-25], [Apr-25], [May-25], [Jun-25] )
) AS pivot_table
UNION
SELECT * FROM (
       SELECT 'Total' AS customer,
       CAST(FORMAT(date, 'MMM-yy') AS INT) AS date,
       REPLACE(amount, '$', '') AS amount
       FROM SALES
) AS sales_data PIVOT (
       SUM(amount)
       FOR date IN ( [Jan-25], [Feb-25] , [Mar-25], [Apr-25], [May-25], [Jun-25] )
) AS pivot_table
)
final_data AS (
       SELECT customer,
       COALESCE([Jan-25], 0) AS Jan_25,
       COALESCE([Feb-25], 0) AS Feb_25,
       COALESCE([Mar-25], 0) AS Mar_25,
       COALESCE([Apr-25], 0) AS Apr_25,
       COALESCE([May-25], 0) AS May_25,
       COALESCE([Jun-25], 0) AS Jun_25,
       FROM pivot_data
)
SELECT customer,
CASE WHEN Jan_25 < 0 THEN CONCAT( '(' , Jan_25, ')$' ) ELSE CONCAT( Jan_25, '$' ) END AS Jan_25,
CASE WHEN Feb_25 < 0 THEN CONCAT( '(' , Feb_25, ')$' ) ELSE CONCAT( Feb_25, '$' ) END AS Feb_25,
CASE WHEN Mar_25 < 0 THEN CONCAT( '(' , Mar_25, ')$' ) ELSE CONCAT( Mar_25, '$' ) END AS Mar_25,
CASE WHEN Apr_25 < 0 THEN CONCAT( '(' , Apr_25, ')$' ) ELSE CONCAT( Apr_25, '$' ) END AS Apr_25,
CASE WHEN May_25 < 0 THEN CONCAT( '(' , May_25, ')$' ) ELSE CONCAT( May_25, '$' ) END AS May_25,
CASE WHEN Jun_25 < 0 THEN CONCAT( '(' , Jun_25, ')$' ) ELSE CONCAT( Jun_25, '$' ) END AS Jun_25,
CASE WHEN customer = 'Total' THEN '' ELSE
      CASE WHEN (Jan_25+Feb_25+Mar_25+Apr_25+May_25+Jun_25) < 0 THEN
      CONCAT('(', Jan_25+Feb_25+Mar_25+Apr_25+May_25+Jun_25, ')$') ELSE
       CONCAT(Jan_25+Feb_25+Mar_25+Apr_25+May_25+Jun_25, '$') END
       END AS Total
FROM final_table;
```

To use the same query in Oracle, we would want to change the format() to to_char(), use ||
instead of concat for more than two arguments, remove alias names for pivot and base query,
remove square bracket representations and use single quotes with aliases, further replace
coalesce with NVL().

Now we will execute the same query using CROSSTABS. Crosstab accepts the base query as
its first argument in the form of a string. So wherever there are quotes inside, we need to add
another quote to escape it. Next argument will contain the column names. Again the base
query has the same rules mentioned above about having three columns and their purpose. In
base queries now we need to use a group by clause due to the presence of duplicates of
customer orders each month.
After defining the crosstab, we need to define the column names to be visible along with their
data types below.

```
SELECT * FROM CROSSTAB (
      'SELECT customer_id AS customer, TO_CHAR(date, ''Mon-YY'') AS INT) AS date,
SUM(CAST(REPLACE(amount, ''$'', '''') AS INT)) AS amount FROM SALES GROUP BY customer_id, date
ORDER BY 1' ,
'VALUES (''Jan-25''), (''Feb-25''), (''Mar-25''), (''Apr-25''), (''May-25''), (''Jun-25'') '

) AS ( customer VARCHAR, Jan-25 BIGINT, Feb-25 AS BIGINT, Mar-25 AS BIGINT, Apr-25 AS BIGINT,
May-25 AS BIGINT, Jun-25 AS BIGINT )
UNION
SELECT * FROM CROSSTAB (
      'SELECT ''Total'' AS customer, TO_CHAR(date, ''Mon-YY'') AS INT) AS date,
SUM(CAST(REPLACE(amount, ''$'', '''') AS INT)) AS amount FROM SALES GROUP BY date ORDER BY 1' ,
'VALUES (''Jan-25''), (''Feb-25''), (''Mar-25''), (''Apr-25''), (''May-25''), (''Jun-25'') '

) AS ( customer VARCHAR, Jan-25 BIGINT, Feb-25 AS BIGINT, Mar-25 AS BIGINT, Apr-25 AS BIGINT,
May-25 AS BIGINT, Jun-25 AS BIGINT )
ORDER BY 1;
```

The tasks left are to convert the null to zeros and another total column. We can
proceed the same way by putting the whole query in a WITH clause and using that
alias to concat $, replace NULL with 0, and add a 'Total' column by selecting from the
WITH clause by using proper CASE statements as done above.

In MySql, this cute little freak won't work. You have to use only with and case
statements.

```
WITH base_query AS (
      SELECT customer_id,
      DATE_FORMAT(date, '%b-%y') AS date,
      REPLACE (amount, '$', '') AS amount
      FROM SALES
)
SELECT customer,
SUM(CASE WHEN date = 'Jan-25' THEN amount ELSE 0 END) AS Jan_25,
SUM(CASE WHEN date = 'Feb-25' THEN amount ELSE 0 END) AS Feb_25,
SUM(CASE WHEN date = 'Mar-25' THEN amount ELSE 0 END) AS Mar_25,
SUM(CASE WHEN date = 'Apr-25' THEN amount ELSE 0 END) AS Apr_25,
SUM(CASE WHEN date = 'May-25' THEN amount ELSE 0 END) AS May_25,
SUM(CASE WHEN date = 'Jun-25' THEN amount ELSE 0 END) AS Jun_25,
SUM(amount) AS Total
FROM base_query
GROUP BY customer
UNION
SELECT 'Total' AS customer,
SUM(CASE WHEN date = 'Jan-25' THEN amount ELSE 0 END) AS Jan_25,
SUM(CASE WHEN date = 'Feb-25' THEN amount ELSE 0 END) AS Feb_25,
SUM(CASE WHEN date = 'Mar-25' THEN amount ELSE 0 END) AS Mar_25,
SUM(CASE WHEN date = 'Apr-25' THEN amount ELSE 0 END) AS Apr_25,
SUM(CASE WHEN date = 'May-25' THEN amount ELSE 0 END) AS May_25,
SUM(CASE WHEN date = 'Jun-25' THEN amount ELSE 0 END) AS Jun_25,
'' AS Total
FROM base_query;
```

This will not have a $ sign in it. Don’t freak out. You just have to move this cutie as a second query inside the WITH as final_data. Then we can select it from final_data and concat with $ sign. So the final badass is as follows:

```
WITH base_query AS (
      SELECT customer_id,
      DATE_FORMAT(date, '%b-%y') AS date,
      REPLACE (amount, '$', '') AS amount
      FROM SALES
)
final_data AS (
     SELECT customer,
     SUM(CASE WHEN date = 'Jan-25' THEN amount ELSE 0 END) AS Jan_25,
     SUM(CASE WHEN date = 'Feb-25' THEN amount ELSE 0 END) AS Feb_25,
     SUM(CASE WHEN date = 'Mar-25' THEN amount ELSE 0 END) AS Mar_25,
     SUM(CASE WHEN date = 'Apr-25' THEN amount ELSE 0 END) AS Apr_25,
     SUM(CASE WHEN date = 'May-25' THEN amount ELSE 0 END) AS May_25,
     SUM(CASE WHEN date = 'Jun-25' THEN amount ELSE 0 END) AS Jun_25,
     SUM(amount) AS Total
     FROM base_query
```

```
      GROUP BY customer
      UNION
      SELECT 'Total' AS customer,
      SUM(CASE WHEN date = 'Jan-25' THEN amount ELSE 0 END) AS Jan_25,
      SUM(CASE WHEN date = 'Feb-25' THEN amount ELSE 0 END) AS Feb_25,
      SUM(CASE WHEN date = 'Mar-25' THEN amount ELSE 0 END) AS Mar_25,
      SUM(CASE WHEN date = 'Apr-25' THEN amount ELSE 0 END) AS Apr_25,
      SUM(CASE WHEN date = 'May-25' THEN amount ELSE 0 END) AS May_25,
      SUM(CASE WHEN date = 'Jun-25' THEN amount ELSE 0 END) AS Jun_25,
      '' AS Total
      FROM base_query
)
SELECT customer,
CASE WHEN Jan_25 < 0 THEN CONCAT( '(' , Jan_25, ')$' ) ELSE CONCAT( Jan_25, '$' ) END AS Jan_25,
CASE WHEN Feb_25 < 0 THEN CONCAT( '(' , Feb_25, ')$' ) ELSE CONCAT( Feb_25, '$' ) END AS Feb_25,
CASE WHEN Mar_25 < 0 THEN CONCAT( '(' , Mar_25, ')$' ) ELSE CONCAT( Mar_25, '$' ) END AS Mar_25,
CASE WHEN Apr_25 < 0 THEN CONCAT( '(' , Apr_25, ')$' ) ELSE CONCAT( Apr_25, '$' ) END AS Apr_25,
CASE WHEN May_25 < 0 THEN CONCAT( '(' , May_25, ')$' ) ELSE CONCAT( May_25, '$' ) END AS May_25,
CASE WHEN Jun_25 < 0 THEN CONCAT( '(' , Jun_25, ')$' ) ELSE CONCAT( Jun_25, '$' ) END AS Jun_25,
CASE WHEN Total < 0 THEN ''
       ELSE  (CASE WHEN Total < 0 THEN CONCAT( '(' , Total, ')$' ) ELSE CONCAT( Total, '$' ) END)
AS Total
FROM final_data;
```

Identity Column

An identity column is a special type of column in SQL Server that automatically generates sequential values for each row in a table. This can be useful for creating unique identifiers, such as primary keys. Syntax is IDENTITY(seed_value, increment_value). An easy example is as below:

```
CREATE TABLE Customer (
     id int NOT NULL IDENTITY(1,1),
     name varchar(255),
     email varchar(255),
     modified datetime DEFAULT (getDate()),
     CONSTRAINT customer_pkey PRIMARY KEY(id)
)
```

Normalization

Normalization is important to remove data redundancies.

First normal form:

1. It should have atomic values or unique values

2. Each row should have single valued columns, no necessity of primary key here.

Second Normal Form:

1. Must be in 1NF
2. No partial dependencies must be present (proper subset of any ck → non prime attributes) or (prime attributes → non prime attributes) must not be present.
3. Table must have a primary key and relation to other tables is to be done through foreign keys.

Third Normal Form:

1. Must be in 2NF
2. No transitive dependencies must be present (non prime attributes → non prime attributes) or A→B and B→C must not be present.

BCNF or Boyce Codd Normal Form:

1. Must be in 3NF
2. For each trivial functional dependency X→Y, X must be a super key.

Built In Functions

These are basically some functions to keep in mind for query formulation:

1. **String based:**
    a. UPPER(first_name)
    b. LOWER(last_name)
    c. LENGTH(string_name)
    d. TRIM(string_name) or TRIM('x' FROM 'xxhelxxloxxx')
    e. INSTR(string_name, substring_part)
    f. SUBSTR(main_string, 1, 10) – indexing starts from 1, not 0.
    g. CONCAT(string_a, string_b)
    h. ASCII('A') = 65
    i. CHAR(65) = 'A'
    j. REPLACE(string_name, old_part, new_part)
2. **Numeric based:**
    a. ABS(-40) = 40
    b. SQRT(144) = 12
    c. MOD(10,3) = 1
    d. POWER(2,3) = 8
    e. TRUNCATE(7.765432, 3) = 7.765
    f. TRUNCATE(7.765432, -1) = 8
    g. FLOOR(40.657) = 40
    h. ROUND(24.3) = 24
    i. CEILING(22.3) = 23

    j. PI = 3.14
    k. SQUARE(7) = 49
    l. SUM()
    m. AVG()
    n. LOG(num, base)
    o. EXP(num)
    p. RAND() → returns a random number
3. **<u>Date based:</u>**
    a. CURTIME() or CURRENT_TIME()
    b. NOW()
    c. SYSDATE()
    d. MONTH('11-10-24') = 10
    e. YEAR('11-10-24') = 24
    f. DAY('11-10-24') = 11
    g. FORMAT('11-10-24', 'DD-MM-YY')
4. **<u>Others:</u>**
    a. CAST('123', INT)
    b. ISNULL(x)
    c. ISNUMERIC(x)
    d. SYSTEM USER
    e. USER NAME

Order of Execution

The order of execution for SQL queries is:

1. FROM: or JOIN clause
2. WHERE: clause
3. GROUP BY: clause
4. HAVING: clause
5. SELECT: clause
6. DISTINCT: clause
7. ORDER BY: clause
8. LIMIT: and/or OFFSET clause

This order ensures that the results are filtered, grouped, selected, sorted, and limited appropriately.