Advanced SQL

You should also be familiar with certain advanced SQL concepts, such as window functions, query optimization, case statements, stored functions, and cursors, aside from the best practices for writing queries. These are useful for handling several use cases and will help you to solve complex problems easily.

RANK()

Rank of the current row within its partition, with gaps

```
RANK() OVER (
 PARTITION BY <expression>[{,<expression>...}]
 ORDER BY <expression> [ASC|DESC], [{,<expression>...}]
)
```

DENSE_RANK()

Rank of the current row within its partition, without gaps

```
DENSE_RANK() OVER (
 PARTITION BY <expression>[{,<expression>...}]
 ORDER BY <expression> [ASC|DESC], [{,<expression>...}]
)
```

PERCENT_RANK()

Percentage rank value, which always lies between 0 and 1

```
PERCENT_RANK() OVER (
 PARTITION BY <expression>[{,<expression>...}]
 ORDER BY <expression> [ASC|DESC], [{,<expression>...}]
)
```

ROW_NUMBER()

It returns unique values

```
ROW_NUMBER() OVER (
 PARTITION BY <expression>[{,
<expression>...}]
 ORDER BY <expression> [ASC|DESC], [{,
<expression>...}]
)
```

LAG()

It returns unique values

```
LAG(expr[, offset[, default]])
 OVER (Window_specification |
Window_name)
```

Name	Marks(out of 500)	Rank	Dense Rank	Row Number
Shubham Agarwal	495	1	1	1
Pariosh Sinha	495	1	1	2
Dilip Kumar	492	3	2	3

INTERVIEW QUESTIONS

How do you add ranking to rows using RANK()? 1

2 What is the difference between RANK() and DENSE_RANK()?

What is an auto-increment? 3

4 How do you use ROW_NUMBER()?

What Is a window function in SQL? 5

6 What Is the syntax of the OVER () Clause ?

Escribe the difference between window functions and aggregate functions. 7

8 What's the difference between window functions and the GROUP BY clause?

How do you define the window frame? 9

10 How does ORDER BY work with OVER? a union clause different from a join clause?

How would you find the second most purchased product? 11

12 When is the ranking field an aggregated value?

WINDOW function()

used to define multiple 'over' clauses.

```
LAG(expr[, offset[, default]])
  OVER (Window_specification | Window_name)
```

LEAD()

Percentage rank value, which always lies between 0 and 1

```
ROW_NUMBER() OVER (
  PARTITION BY <expression>[{,<expression>...}]
  ORDER BY <expression> [ASC|DESC], [{,
<expression>...}]
)
```

CASE Statement()

used to classify data values into different groups according to the given criteria

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2

  .
  WHEN conditionN THEN resultN
  ELSE result
END AS column_name;
```

CASE Expression

The CASE expression goes through conditions and returns a value when the first condition is met

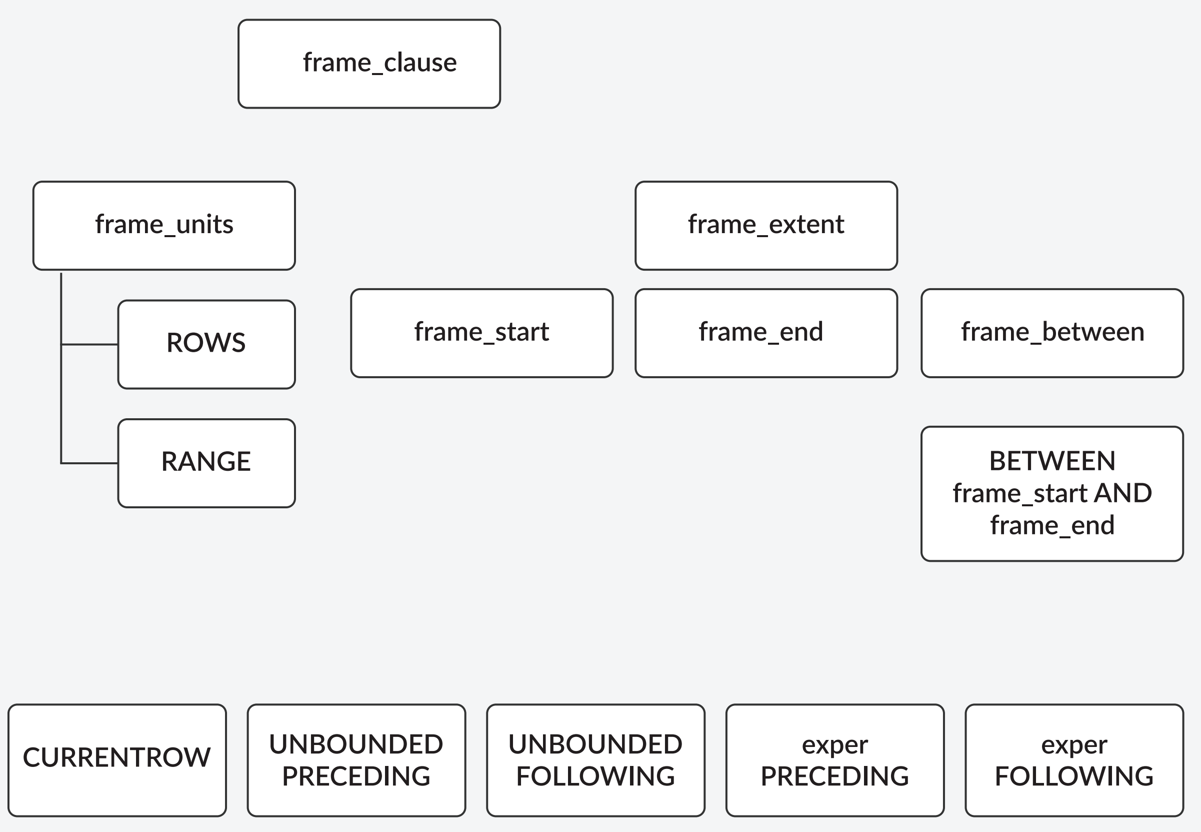
```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The  uantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

FRAME clause

used to subset a set of consecutive rows and calculate moving averages.

Keywords in the 'frame' clause: UNBOUNDED, PRECEDING, FOLLOWING, BETWEEN

```
SELECT
         time, subject, val,
         SUM(val) OVER (PARTITION BY subject ORDER BY time
                    ROWS UNBOUNDED PRECEDING)
           AS running_total,
         AVG(val) OVER (PARTITION BY subject ORDER BY time
                    ROWS BETWEEN 1 PRECEDING AND 1
FOLLOWING)
           AS running_average
      FROM observations;
```



frame_clause

frame_units

ROWS

RANGE

frame_extent

frame_start

frame_end

frame_between

BETWEEN frame_start AND frame_end

CURRENTROW

UNBOUNDED PRECEDING

UNBOUNDED FOLLOWING

exper PRECEDING

exper FOLLOWING

Indexing

An effective way to optimise query execution, as it selects the required data values instead of processing the entire table

Command for creating an index

```
CREATE INDEX index_name
ON table_name (column_1, column_2, ...);
```

Command for dropping an index

```
ALTER TABLE table_name
DROP INDEX index_name;
```

Command for adding an index

```
ALTER TABLE table_name
ADD INDEX index_name(column_1, column_2, ...)
```

Clustered Index	Non-Clustered Index
This is mostly the primary key of the table.	This is a combination of one or more columns of the table.
It is present within the table.	The unique list of keys is present outside the table.
It does not require a separate mapping.	The external table points to different sections of the main table.
It is relatively faster.	It is relatively slower.

Best practices

Some of the best practices that you should remember while writing an SQL query are as follows:

- Comment your code using a hyphen '-' for a single line and '/* ... */' for multiple lines of code.
- Always use table aliases when your query involves more than one source table.
- Assign simple and descriptive names to columns and tables.

Write SQL keywords in upper case and the names of columns, tables and variables in lower case.
Always use column names in the 'order by' clause instead of numbers.
Maintain the right indentation for different sections of a query.
Use new lines for different sections of a query.
Use a new line for each column name.
Use the SQL Formatter or the MySQL Workbench Beautification tool (Ctrl+B) to clean your code.

User Defined Functions(UDF)

The CREATE FUNCTION is also a DDL statement. The function body must contain one RETURN statement. Whenever you are inside a UDF, you need to define another delimiter and reset it to the default ';' (semicolon) after the function ends

```
DELIMITER $$

CREATE FUNCTION
function_name(func_parameter1,
func_parameter2, ...)
RETURN datatype [characteristics]
/*   func_body    */
BEGIN
<SQL Statements>
RETURN expression;
END ; $$

DELIMITER ;
CALL function_name;
```

```
DELIMITER $$

CREATE PROCEDURE
Procedure_name (<Paramter List>)
BEGIN
<SQL Statements>
END $$
DELIMITER ;
CALL Procedure_name;"
```

UDF	Stored Procedure
It supports only the input parameter, not the output.	It supports input, output and input-output parameters.
It cannot call a stored procedure	It can call a UDF.
It can be called using any SELECT statement.	It can be called using only a CALL statement.
It must return a value.	It need not return a value
Only the 'select' operation is allowed	All database operations are allowed.

employee_id	full_name	department	salary
100	full_name	SALES	1000.00
101	Sean Moldy	IT	1500.00
102	Peter Dugan	SALES	2000.00
103	Lilian Penn	SALES	1700.00
104	Milton Kowarsky	IT	1800.00
105	Milton Kowarsky	ACCOUNTS	1200.00
106	Airton Graue	ACCOUNTS	1100.00

Train_id	Station	Time
110	San Francisco	10.00.00
110	Redwood City	10:54:00
110	Palo Alto	11:02:00
110	San Jose	11:02:00
120	San Francisco	11:00:00
120	Redwood City	Non Stop
120	Palo Alto	12:49:00
120	San Jose	13:30:00

Query	Description

SELECT full_name, department,
RANK () OVER (ORDER BY salary) AS Rank_No
FROM employee;
	It ranks employees according to the salaries

SELECT full_name, department,
DENSE_RANK() OVER(ORDER BY department) AS department_rank
FROM employee;
	It ranks Department and show department number for each employee detail

SELECT
RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dept_ranking, department, employee_id, full_name, salary
FROM employee;
	It ranks employees according to their salaries within departments

SELECT
employee_id, full_name, department, salary, salary / MAX(salary)
OVER (PARTITION BY department ORDER BY salary DESC) AS salary_metric
FROM employee ORDER BY 5;
	Employees with the lowest salary (relative to their highest departmental salary) will be listed first

SELECT
train_id, station, time as "station_time", lead(time)
OVER (PARTITION BY train_id ORDER BY time) - time AS time_to_next_station
FROM train_schedule;
	It is used to find the time interval for next station from current station

SELECT
train_id, station, time as "station_time", lead(time)
OVER (PARTITION BY train_id ORDER BY time) - time AS time_to_next_station
FROM train_schedule;
	It shows the time elapsed between a train’s first stop and the current station