1. **SQL Order of Execution:**

https://www.sisense.com/blog/sql-query-order-of-operations/

ORDER	CLAUSE	FUNCTION
1	from	Choose and join tables to get base data.
2	where	Filters the base data.
3	group by	Aggregates the base data.
4	having	Filters the aggregated data.
5	select	Returns the final data.
6	order by	Sorts the final data.
7	limit	Limits the returned data to a row count.

1) **FROM**
   - Fetch the relevant data tables
   - Include **JOIN**
   - In short, relevant table/tables (if JOIN used) are selected.

2) **WHERE**
   - After fetching tables, further filter the tables on columns.

3) **GROUP BY**
   - collapses fields of the result set into their distinct values.
   - **Tips:**
     "When using group by: Group by X means put all those with the same value for X in the same row. Group by X, Y put all those with the same values for both X and Y in the same row."
   - Then when you use an aggregation function (**COUNT, AVG, MAX),** you are performing the aggregated calculation over the entire "row" of grouped-by row mentioned above.
   - You can use GROUP BY 1, 2, 3 to refer to GROUP BY column 1, 2, 3

4) **HAVING**
   - Filter the aggregated data
   - Introduced because **WHERE** should take place before **GROUPBY** and thus cannot handle the aggregated functions

5) **SELECT**
   - Select relevant columns
   -

6) **ORDER BY**
   - Order by selected columns
   - Similar to GROUP BY, you can do ORDER BY 1, 2, 3
   - DESC/AESC (descending/ascending order)

7) **LIMIT**
   - Limit the number of rows presented

Operators:

1. **a <> b**: Non-equal (a does not equal b)
2. col_1 **LIKE "%sampleString%":** check whether the **sampleString** is in col_1
   - Can be used with an IF statement
3. **UNION ALL:** Appended datasets with same columns

CTE (Common Table Expressions):

1. **WITH (…) as a**: "with" statement, declared sub-tables ahead of time so that we can directly refer to the selected queries in the following codes.
2. **Recursive CTE:**
   https://stackoverflow.com/a/29758713/17758024
   https://stackoverflow.com/a/25010457/17758024

   Example:
   With Recursive CTE_NAME (col1, col2, …)
   AS (
   Anchor (Base Case of the Recursion)

   UNION ALL

   Non-recursive parts
)

The execution order of a recursive CTE is as follows:

- First, execute the anchor member to form the base result set (R0), use this result for the next iteration.
- Second, execute the recursive member with the input result set **from the previous iteration (Ri-1) and return a sub-result set (Ri)** until the termination condition is met.
- Third, combine all result sets R0, R1, … Rn using UNION ALL operator to produce the final result set.

Functions:

1. **Ceil/Floor**: Ceil and Floor functions
2. Extract Date Part: **DATE(*timestamp*)**
3. Extract Month Part: **MONTH(*timestamp*)**
4. **YEAR(*timestamp*)**
5. **DATEDIFF(Date1, Date2)**
6. **IF(condition, result_ifTrue, result_ifFalse):** Check condition for all rows and generate a new column accordingly

    - CASE Statement?

SQL Rank Functions:

https://www.sqlshack.com/overview-of-sql-rank-functions/

ROW_NUMBER() OVER(…):

- Create unique **sequential numbers** along rows
- If the order by value is the same, their appear order is arbitrary, and their row_number will be different

Rank() Over(…):

- Create unique **ranks** along rows
- If 2 values are the same, make them the same rank and skip the next rank number

Dense_Rank() Over(…):

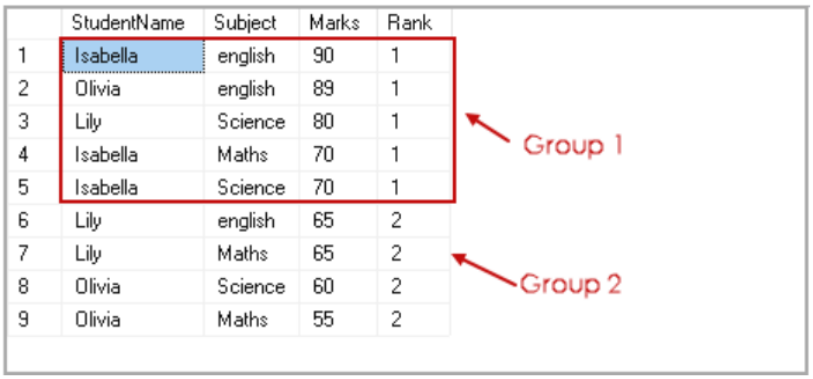
- Create unique **ranks** along rows

- If 2 values are the same, make them the same rank and **DO NOT** skip the next rank number

NTILE(N):

```
SELECT *,
       NTILE(2) OVER(
       ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY rank;
```

In the output, we can see two groups. Group 1 contains five rows, and Group 2 contains four rows.



	StudentName	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	1
3	Lily	Science	80	1
4	Isabella	Maths	70	1
5	Isabella	Science	70	1
6	Lily	english	65	2
7	Lily	Maths	65	2
8	Olivia	Science	60	2
9	Olivia	Maths	55	2

Any ORDER BY operations:

If you have 2 or more columns in ORDER BY (i.e. ORDER BY col_1, col_2), this means that sort by col_1's value first; if there is a tie, sort by col_2's value, and if there is still a tie, sort by col_3 and so on.

Window Functions:

https://mode.com/sql-tutorial/sql-window-functions/

AGG_FUNC() **OVER(PARTITION BY col_3, ORDER BY col_1, col_2)**

Inner Queries:

https://www.tutorialspoint.com/sql/sql-sub-queries.htm

Joins:

1. Inner join
   - Keep only shared rows with the matched ids (Intersection)
2. Outer join
   - Keep all rows that appeared in datasets (Union)
3. Left/Right join
   - Keep rows that appeared in the left/right dataset of the join

4. Cross Join:
    - T1: {‘col1’: [a, b, c]}, T2: {‘col2’: [1,2,3]}
    - (Select * from T1 cross join T2) T3
    - **Result:**

      T3: {‘col1’:[a,b,c,a,b,c,a,b,c], ‘col2’: [1,1,1,2,2,2,3,3,3]}

SQL Practice Question:

1. Users By Average Session Time

https://platform.stratascratch.com/coding/10352-users-by-avg-session-time

Answer:

```
SELECT user_id, AVG(TIMESTAMPDIFF(SECOND, load_time, exit_time)) as AVG_TIME
FROM
(
   SELECT user_id, DATE(timestamp),
   MAX(IF(action = 'page_load', timestamp, NULL)) as load_time,
   MIN(IF(action = 'page_exit', timestamp, NULL)) as exit_time
   FROM facebook_web_log
   group by user_id, DATE(timestamp)
) t
GROUP BY user_id
having AVG_TIME IS NOT NULL;
```

Approach:

Inner-to-Outer:

1. We want a table which, for each user in each day, contains the users' latest load time and earliest exit time.
   To do this, the inner query needs to first be grouped by **user_id** and the **date** part. Then, within each of these group of (user_id + date), we find the latest load time and earliest exit time with **MIN** and **MAX** function.

2. After inner query is constructed, grouping by **user_id** in the outer query to compute the average loading time for each user.
3. The **Having** Clause ensures that if the **TIMESTAMPDIFF** is NULL (time difference not interpretable because it is negative), we remove that row.

SQL Misc:

1. All kinds of joins (inner, outer, full outer, anti, even cross)

2. Dealing with dates, differences of dates, extracting year/day/month from dates
    - Extract Date Part: **DATE(*timestamp*)**
    - Extract Month Part: **MONTH(*timestamp*)**
    - Extract Year Part: **YEAR(*timestamp*)**
    - Date Difference: **DATEDIFF(Date1, Date2)**

3. Common aggregations and their uses, i.e. group by
4. Window aggregations for cumulative sums
5. Dealing with nulls, especially for outer joins (coalesce)
6. Finding orders using window aggregations (particularly row_number())
7. Finding winning streaks and number of repeated values using differences of window aggregations (row_number() over () - row_number() over (partition by X)) then grouping by this difference for determining inclusion to the sequence
8. Case when conditions

9. Summing over case when to get conditional sums/counts

10. Recursive CTEs to generate sequences, or sequences without gaps in the data, or dealing with problems where you have to iterate through everything in order based on exactly one previous result.
    https://learnsql.com/blog/sql-recursive-cte/

11. Regexp (very rare, haven't encountered in interview so far)
12. Prepared statements for making queries based on actual attribute values (pivoting a table without using built in function).