SQL Placement Preparation Notes

1. SQL Basics

- **SQL**: Structured Query Language, used for managing relational databases.
- **Types of SQL commands**:
  - **DDL**: CREATE, ALTER, DROP
  - **DML**: SELECT, INSERT, UPDATE, DELETE
  - **DCL**: GRANT, REVOKE
  - **TCL**: COMMIT, ROLLBACK, SAVEPOINT

2. Data Types

- **INT**, **VARCHAR(n)**, **CHAR(n)**, **DATE**, **FLOAT**, **BOOLEAN**

3. Constraints

- **NOT NULL**: No empty values
- **UNIQUE**: No duplicate values
- **PRIMARY KEY**: NOT NULL + UNIQUE
- **FOREIGN KEY**: References another table
- **CHECK**: Ensures value condition
- **DEFAULT**: Sets default value

4. Joins

- **INNER JOIN**: Common rows only
- **LEFT JOIN**: All from left + matched right
- **RIGHT JOIN**: All from right + matched left
- **FULL OUTER JOIN**: All rows from both
- **SELF JOIN**: Join table with itself

5. Subqueries & CTEs

- **Subquery**: Query inside another query
- **CTE (WITH)**: Temporary result set for reuse

6. GROUP BY & Aggregates

- Used with functions: COUNT, SUM, AVG, MAX, MIN
- **HAVING**: Filter after GROUP BY

7. Set Operations

- **UNION**: Combines, removes duplicates
- **UNION ALL**: Combines, keeps duplicates
- **INTERSECT**: Common rows
- **EXCEPT**: Rows in first not in second

8. Indexes

- Improve query speed
- **CREATE INDEX index_name ON table(column);**

9. Normalization

- Removes redundancy, increases integrity
- **1NF (First Normal Form)**: No repeating groups or arrays. All attributes must contain atomic (indivisible) values.
- **2NF (Second Normal Form)**: 1NF + every non-key attribute fully functionally dependent on the entire primary key (eliminates partial dependencies).
- **3NF (Third Normal Form)**: 2NF + no transitive dependency (non-key attribute should not depend on another non-key attribute).
- **BCNF (Boyce-Codd Normal Form)**: A stronger version of 3NF. Every determinant must be a candidate key.
- **4NF (Fourth Normal Form)**: BCNF + no multi-valued dependencies (an attribute should not have multiple independent values).

- **5NF (Fifth Normal Form)**: 4NF + no join dependency. Data should be reconstructable from smaller relations without redundancy.

10. Transactions

- **ACID** properties:
  - **Atomicity**: All or nothing
  - **Consistency**: Valid state only
  - **Isolation**: Transactions don't interfere
  - **Durability**: Changes persist after commit

Frequently Asked Interview Questions & Answers

1. **Difference between WHERE and HAVING?**
   - `WHERE` filters rows before aggregation.
   - `HAVING` filters after aggregation (used with GROUP BY).
2. **What is the difference between DELETE, TRUNCATE, and DROP?**
   - `DELETE` : Removes rows, can be rolled back.
   - `TRUNCATE` : Removes all rows, faster, cannot be rolled back.
   - `DROP` : Deletes table structure permanently.
3. **Explain different types of JOINs with examples.**
   - `INNER JOIN` : Matches rows from both tables.
   - `LEFT JOIN` : All left rows + matched right.
   - `RIGHT JOIN` : All right rows + matched left.
   - `FULL JOIN` : All rows from both tables.
   - Example: `SELECT * FROM A LEFT JOIN B ON A.id = B.id;`
4. **What is a PRIMARY KEY vs FOREIGN KEY?**
   - `PRIMARY KEY` : Uniquely identifies each row in a table.
   - `FOREIGN KEY` : Enforces link between two tables.
5. **How does indexing improve performance?**
   - Indexes help locate data quickly, avoiding full table scans.
   - Downside: Slower INSERT/UPDATE due to maintenance.
6. **Write a query to find the 2nd highest salary.**

```sql
SELECT MAX(salary) FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

7. **What are window functions?**
   - Functions like `ROW_NUMBER()` , `RANK()` , `DENSE_RANK()` that work over a window of rows.
   - Example:

```sql
SELECT name, salary, RANK() OVER (ORDER BY salary DESC) FROM employees;
```

8. **What is normalization and its types?**
   - Technique to reduce data redundancy and improve data integrity.
   - Types:
     - 1NF (First Normal Form): No repeating groups or arrays. All attributes must contain atomic (indivisible) values.
     - 2NF (Second Normal Form): 1NF + every non-key attribute fully functionally dependent on the entire primary key (eliminates partial dependencies).
     - 3NF (Third Normal Form): 2NF + no transitive dependency (non-key attribute should not depend on another non-key attribute).
     - BCNF (Boyce-Codd Normal Form): A stronger version of 3NF. Every determinant must be a candidate key.
     - 4NF (Fourth Normal Form): BCNF + no multi-valued dependencies (an attribute should not have multiple independent values).
     - 5NF (Fifth Normal Form): 4NF + no join dependency. Data should be reconstructable from smaller relations without redundancy.
9. **Difference between UNION and UNION ALL?**
   - `UNION` : Removes duplicates.
   - `UNION ALL` : Keeps all duplicates.
10. **Explain ACID properties with an example.**

- **Atomicity**: All operations succeed or fail.
- **Consistency**: Always valid state (e.g., no negative bank balance).
- **Isolation**: Transactions don't conflict (e.g., booking same seat).
- **Durability**: Changes persist even after power loss.

Practice Queries

```sql
-- 1. Second highest salary
SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM employees);

-- 2. Employees with duplicate names
SELECT name, COUNT(*) FROM employees GROUP BY name HAVING COUNT(*) > 1;

-- 3. Employees in multiple departments
SELECT emp_id FROM employee_dept GROUP BY emp_id HAVING COUNT(DISTINCT dept_id) > 1;

-- 4. Join employees and departments
SELECT e.name, d.dept_name FROM employees e JOIN departments d ON e.dept_id = d.id;
```

SQL Window Functions Guide

What Are Window Functions?

Window functions perform calculations across a set of rows that are related to the current row. Unlike aggregate functions, they don't collapse rows into a single result - each row retains its identity while gaining additional calculated columns.

Basic Syntax:

```
window_function() OVER (
    [PARTITION BY column1, column2, ...]
    [ORDER BY column1, column2, ...]
    [ROWS/RANGE frame_specification]
)
```

Key Components

- **PARTITION BY**: Divides rows into groups (like GROUP BY but doesn't collapse rows)
- **ORDER BY**: Defines the order for calculations within each partition
- **Frame**: Specifies which rows to include in the calculation (default is all rows in partition)

Sample Data

Let's use this employee table for all examples:

```sql
CREATE TABLE employees (
    id INT,
    name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10,2),
    hire_date DATE
);

INSERT INTO employees VALUES
(1, 'Alice', 'Engineering', 75000, '2020-01-15'),
(2, 'Bob', 'Engineering', 80000, '2019-03-20'),
(3, 'Charlie', 'Engineering', 85000, '2021-06-10'),
(4, 'Diana', 'Marketing', 65000, '2020-08-12'),
(5, 'Eve', 'Marketing', 70000, '2019-11-05'),
(6, 'Frank', 'Marketing', 72000, '2021-02-28'),
(7, 'Grace', 'Sales', 60000, '2020-04-18'),
(8, 'Henry', 'Sales', 62000, '2019-07-22'),
(9, 'Ivy', 'Sales', 68000, '2021-09-14');
```

1. ROW_NUMBER()

Assigns a unique sequential number to each row within a partition.

Example 1: Basic Row Numbering

```sql
SELECT
    name,
    department,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) as overall_rank
FROM employees;
```

Result:

```
name     | department  | salary | overall_rank
---------|-------------|--------|-------------
Charlie  | Engineering | 85000  | 1
Bob      | Engineering | 80000  | 2
Alice    | Engineering | 75000  | 3
Frank    | Marketing   | 72000  | 4
Eve      | Marketing   | 70000  | 5
Ivy      | Sales       | 68000  | 6
Diana    | Marketing   | 65000  | 7
Henry    | Sales       | 62000  | 8
Grace    | Sales       | 60000  | 9
```

Example 2: Row Number by Department

```sql
SELECT
    name,
    department,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as dept_rank
FROM employees;
```

Result:

```
name     | department  | salary | dept_rank
---------|-------------|--------|----------
Charlie  | Engineering | 85000  | 1
Bob      | Engineering | 80000  | 2
Alice    | Engineering | 75000  | 3
Frank    | Marketing   | 72000  | 1
Eve      | Marketing   | 70000  | 2
Diana    | Marketing   | 65000  | 3
Ivy      | Sales       | 68000  | 1
Henry    | Sales       | 62000  | 2
Grace    | Sales       | 60000  | 3
```

2. RANK()

Assigns ranks with gaps for tied values. If two rows have the same value, they get the same rank, and the next rank is skipped.

Example 1: Ranking with Ties

```sql
-- Let's modify data to show ties
UPDATE employees SET salary = 70000 WHERE name = 'Diana';

SELECT
    name,
    department,
    salary,
    RANK() OVER (ORDER BY salary DESC) as salary_rank
FROM employees;
```

Result:

```
name     | department  | salary | salary_rank
---------|-------------|--------|------------
Charlie  | Engineering | 85000  | 1
Bob      | Engineering | 80000  | 2
Alice    | Engineering | 75000  | 3
Frank    | Marketing   | 72000  | 4
Eve      | Marketing   | 70000  | 5
Diana    | Marketing   | 70000  | 5
Ivy      | Sales       | 68000  | 7
Henry    | Sales       | 62000  | 8
Grace    | Sales       | 60000  | 9
```

Example 2: Ranking Within Departments

```sql
SELECT
    name,
    department,
    salary,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dept_rank
FROM employees;
```

3. DENSE_RANK()

Similar to RANK() but doesn't skip ranks after ties. Consecutive ranks are assigned even when there are ties.

Example 1: Dense Ranking

```sql
SELECT
    name,
    department,
    salary,
    DENSE_RANK() OVER (ORDER BY salary DESC) as dense_rank
FROM employees;
```

Result:

```
name     | department  | salary | dense_rank
---------|-------------|--------|------------
Charlie  | Engineering | 85000  | 1
Bob      | Engineering | 80000  | 2
Alice    | Engineering | 75000  | 3
Frank    | Marketing   | 72000  | 4
Eve      | Marketing   | 70000  | 5
Diana    | Marketing   | 70000  | 5
Ivy      | Sales       | 68000  | 6
Henry    | Sales       | 62000  | 7
Grace    | Sales       | 60000  | 8
```

Comparison: ROW_NUMBER vs RANK vs DENSE_RANK

```sql
SELECT
    name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) as row_num,
    RANK() OVER (ORDER BY salary DESC) as rank_val,
    DENSE_RANK() OVER (ORDER BY salary DESC) as dense_rank_val
FROM employees;
```

Result:

```
name     | salary | row_num | rank_val | dense_rank_val
---------|--------|---------|----------|---------------
Charlie  | 85000  | 1       | 1        | 1
Bob      | 80000  | 2       | 2        | 2
Alice    | 75000  | 3       | 3        | 3
Frank    | 72000  | 4       | 4        | 4
Eve      | 70000  | 5       | 5        | 5
Diana    | 70000  | 6       | 5        | 5
Ivy      | 68000  | 7       | 7        | 6
Henry    | 62000  | 8       | 8        | 7
Grace    | 60000  | 9       | 9        | 8
```

4. LAG() and LEAD()

Access values from previous or next rows within the same result set.

Example 1: Compare with Previous Row

```sql
SELECT
    name,
    department,
    salary,
    LAG(salary, 1) OVER (ORDER BY salary DESC) as prev_salary,
    salary - LAG(salary, 1) OVER (ORDER BY salary DESC) as salary_diff
FROM employees;
```

Result:

```
name     | department  | salary | prev_salary | salary_diff
---------|-------------|--------|-------------|------------
Charlie  | Engineering | 85000  | NULL        | NULL
Bob      | Engineering | 80000  | 85000       | -5000
Alice    | Engineering | 75000  | 80000       | -5000
Frank    | Marketing   | 72000  | 75000       | -3000
Eve      | Marketing   | 70000  | 72000       | -2000
Diana    | Marketing   | 70000  | 70000       | 0
Ivy      | Sales       | 68000  | 70000       | -2000
Henry    | Sales       | 62000  | 68000       | -6000
Grace    | Sales       | 60000  | 62000       | -2000
```

Example 2: Department-wise Comparison

```
SELECT
    name,
    department,
    salary,
    LAG(salary, 1) OVER (PARTITION BY department ORDER BY salary DESC) as prev_in_dept,
    LEAD(salary, 1) OVER (PARTITION BY department ORDER BY salary DESC) as next_in_dept
FROM employees;
```

5. FIRST_VALUE() and LAST_VALUE()

Get the first or last value in a window frame.

Example 1: Highest and Lowest in Department

```
SELECT
    name,
    department,
    salary,
    FIRST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC) as highest_in_dept,
    LAST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC
                            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as lowest_
FROM employees;
```

Result:

```
name     | department  | salary | highest_in_dept | lowest_in_dept
---------|-------------|--------|-----------------|---------------
Charlie  | Engineering | 85000  | 85000          | 75000
Bob      | Engineering | 80000  | 85000          | 75000
Alice    | Engineering | 75000  | 85000          | 75000
Frank    | Marketing   | 72000  | 72000          | 65000
Eve      | Marketing   | 70000  | 72000          | 65000
Diana    | Marketing   | 65000  | 72000          | 65000
Ivy      | Sales       | 68000  | 68000          | 60000
Henry    | Sales       | 62000  | 68000          | 60000
Grace    | Sales       | 60000  | 68000          | 60000
```

Example 2: First and Last Hire by Department

```sql
SELECT
    name,
    department,
    hire_date,
    FIRST_VALUE(name) OVER (PARTITION BY department ORDER BY hire_date) as first_hire,
    LAST_VALUE(name) OVER (PARTITION BY department ORDER BY hire_date
                          ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as last_hir
FROM employees;
```

6. NTILE()

Divides rows into a specified number of approximately equal groups.

Example 1: Salary Quartiles

```sql
SELECT
    name,
    department,
    salary,
    NTILE(4) OVER (ORDER BY salary DESC) as salary_quartile
FROM employees;
```

Result:

```
name     | department  | salary | salary_quartile
---------|-------------|--------|----------------
Charlie  | Engineering | 85000  | 1
Bob      | Engineering | 80000  | 1
Alice    | Engineering | 75000  | 1
Frank    | Marketing   | 72000  | 2
Eve      | Marketing   | 70000  | 2
Diana    | Marketing   | 70000  | 2
Ivy      | Sales       | 68000  | 3
Henry    | Sales       | 62000  | 3
Grace    | Sales       | 60000  | 4
```

Example 2: Department Performance Tiers

```
SELECT
    name,
    department,
    salary,
    NTILE(3) OVER (PARTITION BY department ORDER BY salary DESC) as dept_tier
FROM employees;
```

7. PERCENT_RANK()

Calculates the relative rank as a percentage (0 to 1).

Example 1: Salary Percentile

```
SELECT
    name,
    department,
    salary,
    PERCENT_RANK() OVER (ORDER BY salary DESC) as salary_percentile,
    ROUND(PERCENT_RANK() OVER (ORDER BY salary DESC) * 100, 2) as percentile_pct
FROM employees;
```

Result:

```
name     | department  | salary | salary_percentile | percentile_pct
---------|-------------|--------|-------------------|---------------
Charlie  | Engineering | 85000  | 0.0               | 0.00
Bob      | Engineering | 80000  | 0.125             | 12.50
Alice    | Engineering | 75000  | 0.25              | 25.00
Frank    | Marketing   | 72000  | 0.375             | 37.50
Eve      | Marketing   | 70000  | 0.5               | 50.00
Diana    | Marketing   | 70000  | 0.5               | 50.00
Ivy      | Sales       | 68000  | 0.75              | 75.00
Henry    | Sales       | 62000  | 0.875             | 87.50
Grace    | Sales       | 60000  | 1.0               | 100.00
```

8. CUME_DIST()

Calculates the cumulative distribution (percentage of rows with values less than or equal to current row).

Example 1: Cumulative Distribution

```
SELECT
    name,
    department,
    salary,
    CUME_DIST() OVER (ORDER BY salary DESC) as cumulative_dist,
    ROUND(CUME_DIST() OVER (ORDER BY salary DESC) * 100, 2) as cumulative_pct
FROM employees;
```

Result:

```
name     | department  | salary | cumulative_dist | cumulative_pct
---------|-------------|--------|-----------------|---------------
Charlie  | Engineering | 85000  | 0.11            | 11.11
Bob      | Engineering | 80000  | 0.22            | 22.22
Alice    | Engineering | 75000  | 0.33            | 33.33
Frank    | Marketing   | 72000  | 0.44            | 44.44
Eve      | Marketing   | 70000  | 0.67            | 66.67
Diana    | Marketing   | 70000  | 0.67            | 66.67
Ivy      | Sales       | 68000  | 0.78            | 77.78
Henry    | Sales       | 62000  | 0.89            | 88.89
Grace    | Sales       | 60000  | 1.0             | 100.00
```

9. Aggregate Window Functions

Use SUM(), COUNT(), AVG(), MIN(), MAX() with OVER clause for running calculations.

Example 1: Running Totals

```
SELECT
    name,
    department,
    salary,
    SUM(salary) OVER (ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as ru
    AVG(salary) OVER (ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as ru
FROM employees
ORDER BY hire_date;
```

Example 2: Moving Averages

```
SELECT
    name,
    department,
    salary,
    hire_date,
    AVG(salary) OVER (ORDER BY hire_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) as moving_avg
    COUNT(*) OVER (ORDER BY hire_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) as window_size
FROM employees
ORDER BY hire_date;
```

10. NTH_VALUE()

Returns the nth value in the window frame.

Example 1: Second Highest Salary

```
SELECT
    name,
    department,
    salary,
    NTH_VALUE(salary, 2) OVER (PARTITION BY department ORDER BY salary DESC
                              ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as secon
FROM employees;
```

Example 2: Middle Value

```
SELECT
    name,
    department,
    salary,
    NTH_VALUE(salary, 2) OVER (PARTITION BY department ORDER BY salary
                              ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as midd
FROM employees;
```

Practical Use Cases

1. Top N per Group

Find top 2 earners in each department:

```
SELECT name, department, salary
FROM (
    SELECT
        name,
        department,
        salary,
        ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as rn
    FROM employees
) ranked
WHERE rn <= 2;
```

2. Percentage Calculations

Calculate salary as percentage of department total:

```
SELECT
    name,
    department,
    salary,
    ROUND(salary * 100.0 / SUM(salary) OVER (PARTITION BY department), 2) as pct_of_dept_total
FROM employees;
```

3. Running Totals and Cumulative Analysis

Calculate cumulative salary by department:

```
SELECT
    name,
    department,
    salary,
    SUM(salary) OVER (PARTITION BY department ORDER BY hire_date
                      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as running_total
FROM employees;
```

4. Year-over-Year Growth Analysis

Track salary changes over time:

```
SELECT
    name,
    department,
    salary,
    hire_date,
    LAG(salary, 1) OVER (PARTITION BY name ORDER BY hire_date) as prev_salary,
    salary - LAG(salary, 1) OVER (PARTITION BY name ORDER BY hire_date) as salary_increase,
    ROUND(((salary - LAG(salary, 1) OVER (PARTITION BY name ORDER BY hire_date)) /
           LAG(salary, 1) OVER (PARTITION BY name ORDER BY hire_date)) * 100, 2) as growth_rate
FROM employee_history;
```

5. Moving Averages for Trend Analysis

Calculate 3-month moving average of sales:

```
SELECT
    month,
    sales_amount,
    AVG(sales_amount) OVER (ORDER BY month ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) as moving_
    sales_amount - AVG(sales_amount) OVER (ORDER BY month ROWS BETWEEN 2 PRECEDING AND CURRENT 
FROM monthly_sales;
```

6. Gap Analysis - Finding Missing Values

Identify gaps in sequential data:

```sql
SELECT
    id,
    LAG(id) OVER (ORDER BY id) as prev_id,
    id - LAG(id) OVER (ORDER BY id) as gap
FROM orders
WHERE id - LAG(id) OVER (ORDER BY id) > 1;
```

7. Quartile Analysis for Performance Review

Categorize employees into performance quartiles:

```sql
SELECT
    name,
    department,
    performance_score,
    NTILE(4) OVER (ORDER BY performance_score DESC) as performance_quartile,
    CASE
        WHEN NTILE(4) OVER (ORDER BY performance_score DESC) = 1 THEN 'Top Performer'
        WHEN NTILE(4) OVER (ORDER BY performance_score DESC) = 2 THEN 'High Performer'
        WHEN NTILE(4) OVER (ORDER BY performance_score DESC) = 3 THEN 'Average Performer'
        ELSE 'Needs Improvement'
    END as performance_category
FROM employees;
```

8. First and Last Transaction Analysis

Track customer behavior patterns:

```sql
SELECT
    customer_id,
    transaction_date,
    amount,
    FIRST_VALUE(amount) OVER (PARTITION BY customer_id ORDER BY transaction_date) as first_purcl
    LAST_VALUE(amount) OVER (PARTITION BY customer_id ORDER BY transaction_date
                            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as last_pu
    ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY transaction_date) as transaction_sequer
FROM transactions;
```

9. Duplicate Detection and Ranking

Find and rank duplicate records:

```sql
SELECT
    email,
    name,
    registration_date,
    ROW_NUMBER() OVER (PARTITION BY email ORDER BY registration_date) as duplicate_rank
FROM users
WHERE ROW_NUMBER() OVER (PARTITION BY email ORDER BY registration_date) > 1;
```

10. Median Calculation

Calculate median salary by department:

```sql
SELECT DISTINCT
    department,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary) OVER (PARTITION BY department) as mediar
FROM employees;

-- Alternative using NTILE for approximate median
SELECT
    department,
    salary,
    NTILE(2) OVER (PARTITION BY department ORDER BY salary) as half,
    CASE
        WHEN NTILE(2) OVER (PARTITION BY department ORDER BY salary) = 1
        THEN 'Lower Half'
        ELSE 'Upper Half'
    END as salary_group
FROM employees;
```

11. Cohort Analysis

Analyze user retention by signup month:

```sql
SELECT
    signup_month,
    activity_month,
    COUNT(DISTINCT user_id) as active_users,
    FIRST_VALUE(COUNT(DISTINCT user_id)) OVER (PARTITION BY signup_month ORDER BY activity_montl
    ROUND(COUNT(DISTINCT user_id) * 100.0 /
          FIRST_VALUE(COUNT(DISTINCT user_id)) OVER (PARTITION BY signup_month ORDER BY activity
FROM user_activity
GROUP BY signup_month, activity_month;
```

12. ABC Analysis (Pareto Analysis)

Classify products by revenue contribution:

```sql
SELECT
    product_id,
    revenue,
    SUM(revenue) OVER (ORDER BY revenue DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) a
    SUM(revenue) OVER () as total_revenue,
    ROUND(SUM(revenue) OVER (ORDER BY revenue DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
          SUM(revenue) OVER (), 2) as cumulative_percentage,
    CASE
        WHEN ROUND(SUM(revenue) OVER (ORDER BY revenue DESC ROWS BETWEEN UNBOUNDED PRECEDING AND
             SUM(revenue) OVER (), 2) <= 80 THEN 'A'
        WHEN ROUND(SUM(revenue) OVER (ORDER BY revenue DESC ROWS BETWEEN UNBOUNDED PRECEDING AND
             SUM(revenue) OVER (), 2) <= 95 THEN 'B'
        ELSE 'C'
    END as abc_category
FROM product_sales;
```

13. Time Series Analysis - Peak Detection

Identify local maxima in time series data:

```sql
SELECT
    date,
    value,
    LAG(value) OVER (ORDER BY date) as prev_value,
    LEAD(value) OVER (ORDER BY date) as next_value,
    CASE
        WHEN value > LAG(value) OVER (ORDER BY date) AND
             value > LEAD(value) OVER (ORDER BY date) THEN 'Peak'
        WHEN value < LAG(value) OVER (ORDER BY date) AND
             value < LEAD(value) OVER (ORDER BY date) THEN 'Valley'
        ELSE 'Normal'
    END as trend_point
FROM time_series_data;
```

14. Customer Lifetime Value Calculation

Calculate CLV using window functions:

```sql
SELECT
    customer_id,
    order_date,
    order_value,
    SUM(order_value) OVER (PARTITION BY customer_id ORDER BY order_date
                          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as lifetime_value,
    COUNT(*) OVER (PARTITION BY customer_id ORDER BY order_date
                  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as order_count,
    AVG(order_value) OVER (PARTITION BY customer_id ORDER BY order_date
                          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as avg_order_value
FROM orders;
```

15. Session Analysis

Analyze user session patterns:

```sql
SELECT
    user_id,
    page_view_time,
    LAG(page_view_time) OVER (PARTITION BY user_id ORDER BY page_view_time) as prev_page_time,
    page_view_time - LAG(page_view_time) OVER (PARTITION BY user_id ORDER BY page_view_time) as
    CASE
        WHEN page_view_time - LAG(page_view_time) OVER (PARTITION BY user_id ORDER BY page_view_
        THEN 'New Session'
        ELSE 'Same Session'
    END as session_indicator
FROM page_views;
```

Key Differences Summary

Function	Behavior with Ties	Use Case	Example Result
ROW_NUMBER()	Always unique numbers	Pagination, unique identifiers	1, 2, 3, 4, 5, 6
RANK()	Same rank, skips next	Traditional ranking	1, 2, 2, 4, 5, 6
DENSE_RANK()	Same rank, no gaps	Continuous ranking	1, 2, 2, 3, 4, 5

Function	Behavior with Ties	Use Case	Example Result
PERCENT_RANK()	Percentage rank (0-1)	Percentile analysis	0.0, 0.2, 0.4, 0.6, 0.8, 1.0
CUME_DIST()	Cumulative distribution	Distribution analysis	0.17, 0.33, 0.5, 0.67, 0.83, 1.0
NTILE(n)	Divides into n groups	Quartiles/deciles	1, 1, 2, 2, 3, 3
LAG()/LEAD()	Previous/next values	Trend analysis	Previous or next row value
FIRST_VALUE()	First value in window	Baseline comparison	First value in partition
LAST_VALUE()	Last value in window	Final comparison	Last value in partition
NTH_VALUE()	Nth value in window	Specific position	Value at nth position

Window Frame Specifications

Frame Types

- **ROWS**: Physical number of rows
- **RANGE**: Logical range based on values

Common Frame Patterns

```sql
-- Current row only (default for ranking functions)
ROWS BETWEEN CURRENT ROW AND CURRENT ROW

-- All preceding rows and current row
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

-- All rows in partition
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

-- Moving window (3 rows)
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

-- Centered window
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
```

Frame Examples

```sql
SELECT
    name,
    salary,
    -- Running total
    SUM(salary) OVER (ORDER BY salary ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as runn
    -- 3-row moving average
    AVG(salary) OVER (ORDER BY salary ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) as moving_avg_3
    -- Centered average
    AVG(salary) OVER (ORDER BY salary ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) as centered_avg
FROM employees;
```

Best Practices

1. **Choose the right function for your use case**:
    - Use **ROW_NUMBER()** when you need unique identifiers or for pagination
    - Use **RANK()** for traditional ranking where gaps after ties are acceptable
    - Use **DENSE_RANK()** when you want continuous ranking without gaps
    - Use **NTILE()** for creating equal-sized groups or percentiles
    - Use **LAG()/LEAD()** for comparing with adjacent rows
    - Use **FIRST_VALUE()/LAST_VALUE()** for baseline comparisons

2. **Always consider the ORDER BY clause** - it determines the ranking logic and is crucial for meaningful results
3. **Use PARTITION BY strategically** to create separate ranking groups and avoid unexpected results
4. **Be explicit with frame specifications** when using aggregate window functions to ensure predictable results
5. **Handle NULL values appropriately**:

```sql
-- Use COALESCE or NULLIF when needed
LAG(salary, 1, 0) OVER (ORDER BY hire_date) as prev_salary_or_zero
```

6. **Combine window functions for complex analysis**:

```sql
SELECT
    name,
    salary,
    DENSE_RANK() OVER (ORDER BY salary DESC) as salary_rank,
    NTILE(4) OVER (ORDER BY salary DESC) as salary_quartile,
    PERCENT_RANK() OVER (ORDER BY salary DESC) as salary_percentile
FROM employees;
```

7. **Use CTEs for complex window function queries** to improve readability:

```sql
WITH ranked_employees AS (
    SELECT
        name,
        department,
        salary,
        ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as dept_rank
    FROM employees
)
SELECT * FROM ranked_employees WHERE dept_rank <= 3;
```

8. **Consider performance implications**:
    - Window functions can be resource-intensive on large datasets
    - Consider using appropriate indexes on PARTITION BY and ORDER BY columns
    - Use LIMIT with caution as it's applied after window functions

Performance Tips

1. **Index strategy**: Create indexes on columns used in PARTITION BY and ORDER BY clauses

```sql
CREATE INDEX idx_emp_dept_salary ON employees(department, salary DESC);
```

2. **Limit result sets early**: Use WHERE clauses before window functions when possible

```sql
SELECT
    name,
    ROW_NUMBER() OVER (ORDER BY salary DESC) as rank
FROM employees
WHERE department = 'Engineering'  -- Filter first
AND salary > 50000;
```

3. **Use appropriate frame specifications**: Avoid UNBOUNDED FOLLOWING when not necessary

```sql
-- More efficient
SUM(salary) OVER (ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

-- Less efficient if you don't need the full window
SUM(salary) OVER (ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

4. **Consider materialized views** for frequently used window function results
5. **Use EXPLAIN ANALYZE** to understand query execution plans and optimize accordingly

Common Pitfalls to Avoid

1. **Forgetting frame specifications with aggregate functions**:

```sql
-- This might not give expected results
LAST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC)

-- Better approach
LAST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC
                        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

2. **Misunderstanding NULL handling**:

```sql
-- NULLs are typically ordered last in ascending order
ROW_NUMBER() OVER (ORDER BY salary NULLS FIRST) -- Be explicit
```

3. **Using wrong function for the task**:

```sql
-- Don't use RANK() if you need unique identifiers
-- Use ROW_NUMBER() instead
```

4. **Not considering ties in ranking**:

```
-- Be aware that RANK() and DENSE_RANK() handle ties differently
-- Choose based on your business requirements
```

5. **Inefficient frame specifications**:

```
-- Avoid this pattern for large datasets
AVG(salary) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

-- Consider if you really need the full window
AVG(salary) OVER (PARTITION BY department) -- This might be sufficient
```