**Advanced SQL Concepts – Study Guide**

**1. Window Functions**

**Overview**

Window functions allow you to perform calculations across rows related to the current row, without collapsing rows like GROUP BY.

**Syntax**

sql

FUNCTION\_NAME(...) OVER (

PARTITION BY column1

ORDER BY column2

)

**Common Window Functions**

| **Function** | **Purpose** |
| --- | --- |
| ROW\_NUMBER() | Unique row number per partition |
| RANK() / DENSE\_RANK() | Ranking with/without gaps |
| LAG() / LEAD() | Access prior/next row value |
| SUM(), AVG() OVER | Cumulative or grouped aggregates |

**Example**

sql

SELECT emp\_id, department, salary,

ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rank,

LAG(salary) OVER (PARTITION BY department ORDER BY salary) AS prev\_salary,

SUM(salary) OVER (PARTITION BY department) AS total\_dept\_salary

FROM employees;

**2. CTE vs Derived Table**

**CTE (Common Table Expression)**

* Temporary named result set
* Improves readability
* Can be recursive

sql

WITH dept\_avg AS (

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

)

SELECT \* FROM dept\_avg WHERE avg\_salary > 55000;

**Derived Table**

* Inline subquery in the FROM clause
* No name reuse possible

sql

SELECT \* FROM (

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

) AS avg\_dept

WHERE avg\_salary > 55000;

**Comparison**

| **Use Case** | **CTE** | **Derived Table** |
| --- | --- | --- |
| Readability | Yes | No |
| Reuse subquery | Yes | No |
| One-off logic | No | Yes |

**3. UNION and EXCEPT**

**UNION**

* Combines results from two queries
* Removes duplicates by default

sql

SELECT name FROM customers

UNION

SELECT name FROM suppliers;

**EXCEPT**

* Returns rows in the first query but not in the second

sql

SELECT name FROM customers

EXCEPT

SELECT name FROM suppliers;

**Notes**

* Use UNION ALL to include duplicates
* Not all RDBMS support EXCEPT (e.g., MySQL uses NOT IN or LEFT JOIN)

**4. Materialized View**

**Definition**

* A physically stored result of a query
* Improves performance for complex aggregations
* Needs to be refreshed

**Syntax (PostgreSQL)**

sql

CREATE MATERIALIZED VIEW sales\_summary AS

SELECT region, SUM(amount) AS total\_sales

FROM orders

GROUP BY region;

REFRESH MATERIALIZED VIEW sales\_summary;

**Use Cases**

* Dashboards
* BI reports
* Heavy aggregations

**5. MERGE / UPSERT**

**Purpose**

Handle INSERT and UPDATE in one SQL block based on matching condition.

**Syntax (SQL Server, Oracle, MySQL 8+)**

sql

MERGE INTO target\_table AS target

USING source\_table AS source

ON target.id = source.id

WHEN MATCHED THEN

UPDATE SET target.col = source.col

WHEN NOT MATCHED THEN

INSERT (id, col) VALUES (source.id, source.col);

**Use Cases**

* Sync from staging to main tables
* Upsert employee or customer records

**6. Isolation Levels and Deadlocks**

**Isolation Levels**

Control how concurrent transactions interact.

| **Isolation Level** | **Prevents Dirty Read** | **Prevents Non-Repeatable Read** | **Prevents Phantom** |
| --- | --- | --- | --- |
| READ UNCOMMITTED | No | No | No |
| READ COMMITTED | Yes | No | No |
| REPEATABLE READ | Yes | Yes | No |
| SERIALIZABLE | Yes | Yes | Yes |

**Deadlock Example**

Scenario:

1. Transaction A locks row 1 and waits for row 2
2. Transaction B locks row 2 and waits for row 1

Fixes:

* Always access resources in the same order
* Keep transactions short
* Use timeouts

**7. Stored Procedures**

**Definition**

A stored block of SQL code that can be reused and called with parameters.

**Syntax**

sql

CREATE PROCEDURE adjust\_salary(IN dept VARCHAR(50), IN pct INT)

BEGIN

UPDATE employees

SET salary = salary \* (1 + pct / 100)

WHERE department = dept;

END;

**Benefits**

* Encapsulation
* Reusability
* Performance (precompiled)

**8. Triggers**

**Definition**

Automatic actions invoked by events like INSERT, UPDATE, or DELETE.

**Example: Logging salary updates**

sql

CREATE TRIGGER log\_salary\_change

AFTER UPDATE ON employees

FOR EACH ROW

BEGIN

INSERT INTO audit\_log(emp\_id, old\_salary, new\_salary)

VALUES (OLD.emp\_id, OLD.salary, NEW.salary);

END;

**Use Cases**

* Audit logging
* Enforcing business rules
* Preventing invalid data changes

**9. Build Analytic View + Trigger for Integrity**

**Analytic View**

sql

CREATE VIEW employee\_analysis AS

SELECT emp\_id, department, salary,

RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS salary\_rank,

SUM(salary) OVER (PARTITION BY department) AS total\_dept\_salary

FROM employees;

**Integrity Enforcement with Trigger**

sql

CREATE TRIGGER prevent\_negative\_balance

BEFORE UPDATE ON accounts

FOR EACH ROW

BEGIN

IF NEW.balance < 0 THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient balance!';

END IF;

END;

**Summary Table**

| **Topic** | **Use Case** |
| --- | --- |
| Window Functions | Analytics, ranking, comparisons |
| CTE vs Derived Table | Readability vs temporary calculation |
| UNION / EXCEPT | Set operations across tables |
| Materialized View | Speeding up expensive query results |
| MERGE UPSERT | Single-step sync logic |
| Isolation Levels | Control concurrency and data integrity |
| Deadlocks | Understand and avoid blocking transactions |
| Stored Procedures | Reuse and simplify logic |
| Triggers | Automate responses to data changes |
| Analytics + Integrity | Combine reporting and data correctness |