Database fundamentals summary (Part 3)

Agenda

1. **SQL Subqueries (Nested Queries)**
    - Basic Subquery
    - Using `ALL`
    - Using `ANY` / `SOME`
2. **SQL Aggregate Functions**
    - Common functions: `MAX`, `MIN`, `AVG`, `SUM`, `COUNT`
    - Practical examples
3. `GROUP BY` **&** `HAVING`
4. **SQL Views**
    - Definition & Use Cases
    - Creating a View
    - Querying Views
    - Updating / Modifying Views
    - Deleting Views
    - Advanced Options (`WITH CHECK OPTION`, Permissions)
    - Advantages of Views
5. **SQL Index**
    - What is an Index?
    - Advantages & Disadvantages
    - Best Practices
    - Syntax & Examples

- Key Idea: **Speed vs Overhead**

SQL Subqueries (Nested Queries)

1. Basic Subquery

- A query **inside another query**.
- **Used to compare against a single computed value.**

```
SELECT fname, salary
-- Outer query finds all employees earning more than that salary.
FROM Employee
WHERE salary > (
    SELECT salary -- Inner query finds Ahmed Ali's salary.
    FROM Employee WHERE fname = 'Ahmed' AND lname = 'Ali'
);
```

2. Using `ALL`

- Condition must be true **against ALL values** returned by subquery.
- If any value fails, row is excluded.

```
SELECT fname, salary FROM Employee
-- Employee salary must be higher than every salary in dept 10.
WHERE salary > ALL (
    SELECT salary FROM Employee WHERE dno = 10
);
```

3. Using `ANY` / `SOME`

- Condition must be true for **at least one value** from subquery.
- More flexible than `ALL`.

```
SELECT fname, salary FROM Employee
-- Employee salary is higher than at least one salary in dept 10.
WHERE salary > ANY (
    SELECT salary FROM Employee WHERE dno = 10
);
```

Summary

- **Basic Subquery** - >Query inside query (ex. Compare salary with Ahmed Ali)
- `ALL` - > Must satisfy condition vs **all values** (ex. Salary > all salaries in dept 10)
- `ANY` / `SOME` - > Must satisfy condition vs **at least one value** (ex. Salary > any salary in dept 10)

SQL Aggregate Functions

What Are Aggregate Functions?

- Perform a calculation on **a set of values** → return **1 value**.
- Ignore **NULLs** (except `COUNT(*)`).

Common aggregate functions:

- `MAX()` → highest value
- `MIN()` → lowest value
- `AVG()` → average value
- `SUM()` → total sum
- `COUNT()` → count of rows (ignores NULLs)

Tip: Always use aliases (`AS`) for clarity.

- - Highest & lowest salary
  SELECT MAX(salary) AS max_salary,
  MIN(salary) AS min_salary FROM Employee;
- - Average salary
  SELECT AVG(salary) AS avg_salary FROM Employee;
- - Total salary
  SELECT SUM(salary) AS total_salary FROM Employee;
- - Count employees with non-null salary
  SELECT COUNT(salary) AS employee_count FROM Employee;

- - Count all rows (even if salary is NULL)
  SELECT COUNT(*) AS total_employees FROM Employee;

GROUP BY & HAVING

- `GROUP BY` → groups rows by column(s).
- `HAVING` → filters **groups** after aggregation.
- `WHERE` → filters **rows** before grouping.

Example: Average Salary per Department

```
SELECT department_number, AVG(salary) AS avg_salary FROM Employee
GROUP BY department_number HAVING MAX(salary) > 1800;
-- Show only departments where the top salary > 1800
```

SQL Views

What is a View?

- A **View** = **logical table** (virtual table).
- **Does not store data** → only stores the query definition.
- Acts like a **window** to underlying tables.
- Uses:
  - Simplify complex queries
  - Restrict sensitive data
  - Present data in a customized way

Creating a View

```
CREATE VIEW EmployeeProject AS
SELECT e.fname,e.lname,p.pname,w.hours FROM Employee e
JOIN WorksOn w ON e.emp_id = w.emp_id
JOIN Project p ON w.project_id = p.project_id;
```

```
-- View Name → EmployeeProject
-- Combines data from multiple tables (Employee, WorksOn, Project)
```

Querying a View

```
SELECT * FROM EmployeeProject -- Query a View just like a regular table.
WHERE hours > 20;
```

Updating / Modifying Views

- **Simple View** (single table, no joins/aggregates) → DML (`INSERT`, `UPDATE`, `DELETE`) usually **allowed**.
- **Complex View** (joins, aggregates, subqueries) → DML often **restricted**.
- Modify View:

```
CREATE OR REPLACE VIEW EmployeeProject AS -- new definition here
```

Delete a view

```
DROP VIEW EmployeeProject;
```

Advanced Options

- WITH CHECK OPTION → Ensures any `INSERT/UPDATE` via the View satisfies the View condition.

```
CREATE VIEW HighSalaryEmployees AS
SELECT fname, lname, salary FROM Employee
WHERE salary > 3000 WITH CHECK OPTION;
```

- **Access control** → Grant permissions on the View instead of base table:

```
GRANT SELECT ON HighSalaryEmployees TO UserX;
```

Advantages of Views

1. **Simplifies queries** (hide joins/logic).
2. **Restricts access** (security & privacy).
3. **Consistency** (reuse same query definition).
4. **Abstraction** (users don’t need to know underlying schema).

SQL Index

What is an Index?

- **Index** = Database object that **speeds up data retrieval.**
- Works like a **phonebook** → find rows quickly **without scanning the whole table**.
- Built on **one or more columns**, stores **sorted values + row pointers.**

Advantages

- Much faster `SELECT` queries.
- Improves performance of `JOIN`, `WHERE`, and `ORDER BY`.

Disadvantages

- Slows down `INSERT` / `UPDATE` / `DELETE` (because index must be updated).
- Consumes extra **storage space**.

Best Practices

- Index **columns used often in** `WHERE`, `JOIN`, `ORDER BY`, `GROUP BY`.
- Avoid indexing columns that:
  - Change frequently
  - Have **low selectivity** (e.g., gender `M/F`)
- Syntax & Examples:

```
-- 1. Create index on last name
CREATE INDEX idx_employee_name ON Employee (lname);
-- 2. Composite index (last name + department)
CREATE INDEX idx_emp_lname_dept ON Employee (lname, department_id);
-- 3. Drop index
DROP INDEX idx_employee_name;
--Note: The DBMS decides automatically when to use an index in a query.
```

Key Idea

- **Indexes = Speed for SELECT, Cost for DML (Insert/Update/Delete).**
- Use wisely: balance **performance vs overhead**.