Database fundamentals summary (Part 2)

Agenda

1. SQL Overview (DDL, DCL, DML)

2. SQL DDL – Data Definition Language

- `CREATE` , `ALTER` , `DROP` , `TRUNCATE`

3. SQL DCL – Data Control Language

- `GRANT` , `WITH GRANT OPTION` , `REVOKE`

4. SQL DML – Data Manipulation Language

- `INSERT` , `UPDATE` , `DELETE`
- Difference between `DELETE` vs `TRUNCATE`

5. Filtering & Searching `WHERE` & `LIKE`

- Operators: `AND` , `OR` , `BETWEEN` , `IN`
- **Aliases (`AS`)** → Temporary names for columns

6. Sorting & Removing Duplicates

- `ORDER BY` , `DISTINCT`

7. SQL Joins

- `INNER JOIN` , `LEFT JOIN` , `RIGHT JOIN` , `FULL OUTER JOIN` , `SELF JOIN`

8. Aggregate Functions

- `MAX()` , `MIN()` , `AVG()` , `SUM()` , `COUNT()`

9. Grouping & Filtering Groups

- `GROUP BY` & `HAVING`

10. `SELECT` Statement & Execution Order

- `FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY`

SQL – DDL (Data Definition Language)

- Defines & manages **structure** of DB (not data).
- Commands
  1. **CREATE** → new table (Defines columns, data types, and constraints (like primary key)).
  2. **ALTER** → modify table structure (ex. add a new column, change a column's type).
  3. **DROP** → delete table & data (Cannot be undone easily)
  4. **TRUNCATE** → clear rows, keep structure , keeps the table structure (columns remain).

SQL – DCL (Data Control Language)

- Controls **permissions,** It deals with **permissions and access control** for database objects (like tables, views, etc.).
- **GRANT** → give privileges (e.g., `SELECT` , `INSERT`, `UPDATE`)
  - **WITH GRANT OPTION** → allows the user to pass on these privileges to other users.
- **WITH GRANT OPTION** → allow user to grant others
- **REVOKE** → remove privileges
  - You can revoke a single permission (like `SELECT`) or all privileges from a user.
  - Ensures that only authorized users retain access to database objects.

SQL – DML (Data Manipulation Language)

INSERT

- Adds new rows into a table.
- Ways: full insert, partial insert, or using `SELECT`.
- **Use-Case:** Adding new records or migrating data.

```
-- Insert with all columns
INSERT INTO Employee (id, name, salary) VALUES (1, 'Ali', 5000);

-- Insert without column names (must follow table order)
INSERT INTO Employee VALUES (2, 'Sara', 6000);

-- Insert with some columns only (others = NULL/default)
INSERT INTO Employee (name) VALUES ('Omar');

-- Insert from another table
INSERT INTO Employee_Backup
SELECT * FROM Employee WHERE salary > 5000;
```

UPDATE

- Used to **modify existing data** in a table.
- Syntax: `UPDATE table_name SET column = value WHERE condition;`
- Always use `WHERE` (otherwise updates all rows!).

```
-- Update salary for employee with id = 3
UPDATE Employee
SET salary = 6000
WHERE id = 3;

-- Update both name and salary
UPDATE Employee
SET name = 'Ali', salary = 6500
WHERE id = 4;
```

DELETE

- Removes rows from a table.

- Syntax: `DELETE FROM table_name WHERE condition;`
- Safer with `WHERE`, or all rows will be deleted.
- `DELETE` **vs.** `TRUNCATE`:
  - `DELETE` → DML, can delete specific rows with `WHERE`, can rollback.
  - `TRUNCATE` → DDL, deletes all rows quickly, no `WHERE` allowed, cannot rollback.

```
-- Delete one employee by id
DELETE FROM Employee
WHERE id = 3;

-- Delete all employees in a department
DELETE FROM Employee
WHERE dept_id = 5;

-- Delete all rows from table
DELETE FROM Employee;  -- (like TRUNCATE but slower)

-- Remove all employees, reset table
TRUNCATE TABLE Employee;
```

Filtering Data with `WHERE`

- **Key idea:** Aliases make query results **clearer and readable**.
- Used to **filter rows** in a `SELECT` query.
- Common operators:
  - `AND` → both must be true
  - `OR` → at least one true
  - `BETWEEN` → in a range
  - `IN` → matches any value in list

```
SELECT
   first_name || ' ' || last_name AS full_name,    -- combine names
```

```
    salary,                          -- monthly salary
    salary * 0.1 AS bonus,           -- 10% bonus
    salary * 12 AS yearly_salary     -- yearly salary
FROM Employee
WHERE salary * 12 > 10000;           -- filter by yearly salary
```

Pattern Matching with `LIKE`

- Use when you **don't know the exact value** but know the **pattern**.
- Wildcards:
  - `_` → one single character
  - `%` → zero or more characters

```
-- Names starting with "A"
SELECT name
FROM Employee
WHERE name LIKE 'A%';   -- Ali, Ahmed, Amira

-- Second letter is "o"
SELECT name
FROM Employee
WHERE name LIKE '_o%';  -- John, Mona

-- Matches Ahmad or Ahmed
SELECT name
FROM Employee
WHERE name LIKE 'Ahm_d';
```

Aliases in SQL (`AS`)

- Use `AS` to give a **temporary name** (alias) to a column or expression.
- Useful for:
  - **Calculations** (e.g., bonus, yearly salary)
  - **Combining columns** (full name)
  - **Filtering expressions**

```
SELECT
   first_name || ' ' || last_name AS full_name,   -- combine names
   salary,                                        -- show monthly salary
   salary * 0.1 AS bonus,                         -- 10% bonus
   salary * 12 AS yearly_salary                   -- yearly salary
FROM Employee
WHERE salary * 12 > 10000;                        -- filter by yearly salary
```

ORDER BY

- Used to **sort query results**.
- `ASC` → ascending (default).
- `DESC` → descending.
- Can sort by **multiple columns**.

```
SELECT first_name, ssn, department_number, salary
FROM Employee
ORDER BY
   department_number ASC, -- First, sort by department number (smallest → largest)
   salary DESC;                -- Then, within each department, sort by salary (highest → lowest)
```

DISTINCT

- Removes **duplicate rows** from results.
- Works on **single column** (unique values) or **multiple columns** (unique combinations).

```
-- Unique departments
SELECT DISTINCT department_number
FROM Employee;

-- Unique (department, supervisor) pairs
SELECT DISTINCT department_number, supervisor_ssn
FROM Employee;
```

SQL Joins

1. INNER JOIN

- Show only rows that match in **both tables**.

```
SELECT e.fname, d.dname
FROM Employee e
INNER JOIN Department d ON e.dno = d.dnumber;
-- Employees WITH a department
```

2. LEFT JOIN (LEFT OUTER JOIN)

- Show **all rows from left table**, + matches from right.

```
SELECT e.fname, d.dname -- If no match → NULL.
FROM Employee e
LEFT JOIN Department d ON e.dno = d.dnumber;
-- All employees, even if NO department
```

3. RIGHT JOIN (RIGHT OUTER JOIN)

- Show **all rows from right table**, + matches from left.

```
SELECT e.fname, d.dname -- If no match → NULL.
FROM Employee e
RIGHT JOIN Department d ON e.dno = d.dnumber;
-- All departments, even if NO employee
```

4. FULL OUTER JOIN

- Show **all rows from both tables**.

```
SELECT e.fname, d.dname -- If no match → NULL.
FROM Employee e
FULL OUTER JOIN Department d ON e.dno = d.dnumber;
-- All employees + all departments
```

5. SELF JOIN

- A table joins **itself** (hierarchy).

```
SELECT E.fname AS EmployeeName, S.fname AS SupervisorName
FROM Employee E
JOIN Employee S ON E.supervisor_ssn = S.ssn;
-- Employee with their supervisor
```

Summary

- INNER JOIN (Only matches) → Only rows with matches in both tables
- LEFT JOIN (All left) → All rows from left + matches from right
- RIGHT JOIN (All right) → All rows from right + matches from left
- FULL OUTER JOIN (Everything) →All rows from both tables
- SELF JOIN (Same table) →Join table with itself (e.g. employee & supervisor)

`SELECT` & Execution Order

Purpose

- `FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY`
- `SELECT` → choose which columns to display
- `FROM` → specify source table(s)
- `WHERE` → filter rows (before grouping)
- `GROUP BY` → group rows by column(s)
- `HAVING` → filter groups (after aggregation)
- `ORDER BY` → sort the final result

```
SELECT
   department_number,              -- Show department number
   MAX(salary) AS max_salary       -- Highest salary in each department
FROM Employees
GROUP BY department_number         -- Group employees by department
HAVING AVG(salary) > 1200          -- Keep only groups with avg salary > 1200
ORDER BY max_salary DESC;          -- Sort: highest salary first
```

Execution Order (inside DBMS)

1. `FROM` → load table(s)
2. `WHERE` → filter rows
3. `GROUP BY` → group rows
4. **Aggregate Functions** → compute `MAX()`, `AVG()`, etc.
5. `HAVING` → filter groups
6. `SELECT` → pick final columns
7. `ORDER BY` → sort results

Rules to Remember

- Any **non-aggregated column** in `SELECT` **must appear in** `GROUP BY`.
- `ORDER BY` can use columns from:
  - `SELECT` list
  - `GROUP BY` list
  - Aliases (`AS`)