National University
Of Computer and Emerging Sciences

Foundation of Advancement
Of Science and Technology

Lab Number:	04
Course Code:	CL2005
Course Title:	Database Systems - Lab
Semester:	Spring 2026
Agenda:	DQL (Cont.) Grouping & Aggregation, Relational Design
Instructor:	Muhammad Mehdi
Email:	muhammad.mehdi@nu.edu.pk
Office:	Rafaqat Lab

Table of Contents

1 SQL Data Query Language (DQL)

DQL commands query and retrieve the different sets of data from the database. The `SELECT` keyword is used to retrieve data from one or more tables.

2 Grouping and Aggregation

In SQL, grouping and aggregation are used to summarize data by **organizing rows into groups** based on one or more columns and then applying aggregate functions to each group. This is useful for generating reports and meaningful insights from large datasets.

2.1 The GROUP BY Clause

The `GROUP BY` clause groups rows that have the same values in specified columns. It is commonly used with aggregate functions such as `COUNT()`, `SUM()`, `etc.` to perform calculations on each group.

Without `GROUP BY`, aggregate functions compute values over the entire table.

2.1.1 GROUP BY with Single Column

When `GROUP BY` is used with a single column, rows are grouped based on the unique values of that column. Each distinct value forms one group, and aggregate functions are applied to each group separately.

Use cases: Average CGPA per department, total salary per designation, number of students per class.

Rules:

- Every selected column must either be part of the `GROUP BY` clause or be used inside an aggregate function.
- Grouping is performed before aggregate functions are applied.

General Syntax:

```
SELECT group_column, AGGR_FUNC(S)(non-group_column)
FROM table_name
GROUP BY group_column;
```

Examples:

```
-- Without Grouping: Average CGPA of all students
SELECT AVG(cgpa)
FROM students;

-- With Grouping: Average CGPA per department
SELECT department, AVG(cgpa)
FROM students
GROUP BY department;
```

2.1.2 GROUP BY with Multiple Columns

When multiple columns are used in `GROUP BY`, rows are grouped based on the unique combination of values in those columns. This allows more detailed aggregation.

Use cases: Total sales per department per year, average marks per class per subject.

Rules:

- All non-aggregated columns in the `SELECT` clause must appear in the `GROUP BY` clause.
- The order of columns in `GROUP BY` affects how the grouping is logically formed.

General Syntax:

```
SELECT
    group_column1, group_column2, ...,
    AGGR_FUNC(S)(non-group_column)
FROM table_name
GROUP BY group_column1, group_column2, ...;
```

Example:

```
-- Average CGPA for each department and batch
SELECT department, batch, AVG(cgpa)
FROM students
GROUP BY department, batch;
```

2.2 The HAVING Clause

The `HAVING` clause is used to filter grouped results based on aggregate conditions. Unlike `WHERE`, which filters individual rows before grouping, `HAVING` filters groups after aggregation is performed.

Use cases: Departments with more than 10 students, classes with average CGPA above a threshold.

General Syntax:

```
SELECT group_column(s), AGGR_FUNC(S)(non-group_column)
FROM table_name
GROUP BY group_column(s)
HAVING aggregate_condition;
```

Examples:

```
-- Departments with average CGPA greater than 3.0
SELECT department, AVG(cgpa) AS avg_cgpa
FROM students
GROUP BY department
HAVING AVG(cgpa) > 3.0;

-- Departments with more than 10 students
SELECT department, COUNT(*) AS student_count
FROM students
GROUP BY department
HAVING COUNT(*) > 10;
```

2.2.1 WHERE vs HAVING

WHERE	HAVING
Filters rows before grouping	Filters rows (groups) after grouping
Applied before `GROUP BY`	Applied after `GROUP BY`
Cannot use aggregate functions	Used with aggregate functions
Filters raw data	Filters aggregated results

Example:

```
-- Departments of batch 2023 having average CGPA below 2.5
SELECT department, AVG(cgpa) AS avg_cgpa
FROM students
WHERE batch = 2023
GROUP BY department
HAVING AVG(cgpa) < 2.5;
```

3 Working with Multiple Tables

In real-world databases, data is not stored in a single table. Instead, it is distributed across multiple related tables to reduce duplication and maintain consistency.

For example, If we store student name, department name, and faculty name in one table, the department and faculty information would be repeated for every student. This leads to:

- **Data redundancy** (duplicate data)
- **Update anomalies** (changes must be made in many places)
- **Inconsistency** (conflicting values for the same data)
- **Inefficiency** (wasted storage and slower operations)

By separating independent entities (e.g., Students, Departments, Faculty) into different tables and linking them using keys, databases become easier to maintain, more consistent, and more scalable.

3.1 Primary Keys

A **primary key** uniquely identifies each row in a table. It is a column (or set of columns) in a table whose values are always unique and non-empty.

It has the following properties:

- Must be **unique**
- Must **not be NULL**
- Should be **stable** (its value should not change over time)
- There is **only one primary key per table** (it may consist of one or more columns)

3.1.1 Column-Level (Inline) Definition

This style is commonly used for **single-column primary keys**.

Examples:

```
CREATE TABLE students (
    id INTEGER PRIMARY KEY,    -- Alias for rowid in SQLite
    name TEXT NOT NULL
);

CREATE TABLE books (
    isbn TEXT PRIMARY KEY,     -- Not an alias for rowid
    author TEXT NOT NULL
);
```

SQLite Note:

- In SQLite, `INTEGER PRIMARY KEY` is an alias for `rowid`. It auto-increments by default.
- Whereas, `AUTOINCREMENT` prevents reuse of deleted rowids (rarely needed).

3.1.2 Table-Level Definition

Table-level definitions are used for:

- Single-column primary keys (optional style)
- **Composite primary keys** or multi-column keys (mandatory for this case)

Examples:

```
CREATE TABLE students (
    id INTEGER,
    name TEXT,
    PRIMARY KEY (id) -- Single-column primary key
);

CREATE TABLE orders (
    prod_id INTEGER,
    customer_id INTEGER,
    cost REAL,
    PRIMARY KEY (prod_id,customer_id) -- Multi-column primary key
);
```

3.2 Foreign Keys

A foreign key is a column (or set of columns) in one table that refers to the primary key of another table. It is used to establish and enforce relationships between tables.

Foreign keys enforce **referential integrity**. They are used to establish relationships between tables in a relational database.

3.2.1 Column-Level (Inline) Definition

This style is used for single-column foreign keys.

```
CREATE TABLE departments (
    dept_id INTEGER PRIMARY KEY,
    dept_name TEXT NOT NULL
);

CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    dept_id INTEGER REFERENCES departments(dept_id)
);
```

3.2.2 Table-Level Definition

This style is required for:

- Composite foreign keys (multi-column)
- Or when you want clearer separation of constraints

Examples:

```
CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    dept_id INTEGER,
    FOREIGN KEY (dept_id)  -- Single-column foreign key
        REFERENCES departments(dept_id)
);

CREATE TABLE order_logs (
```

```
    prod_id INTEGER,
    customer_id INTEGER,
    log_time NUMERIC,
    FOREIGN KEY (prod_id,customer_id) -- Multi-column foreign key
        REFERENCES orders(prod_id, customer_id)
);
```

3.3 Referential Integrity

Referential integrity ensures that relationships between tables remain valid and consistent. It prevents situations where a record in one table refers to a non-existent record in another table.

In SQL, referential integrity is enforced using **foreign key constraints**.

Rules:

- A foreign key must reference an existing parent (referenced) row.
- Parent rows cannot be deleted or updated if child rows depend on them (unless actions are defined).

SQLite Note:

Unlike other DBMSs, SQLite enforces foreign keys only when explicitly enabled with the following command:

```
PRAGMA foreign_keys = ON;
```

3.4 Foreign Key Actions (ON DELETE / ON UPDATE)

When a row in a **parent table** is deleted or its primary key is updated, related rows in the **child (referencing) table** may be affected.

SQL allows you to define referential / foreign key actions that specify what should happen in such cases.

3.4.1 Supported Referential Actions in SQLite

Action	**Meaning**
`NO ACTION` *(default)*	Rejects the operation if it violates referential integrity

`RESTRICT`	Same as `NO ACTION` in SQLite
`CASCADE`	Propagates the delete/update to the child table
`SET NULL`	Sets the foreign key value in the child table to `NULL`
`SET DEFAULT`	Sets the foreign key value to its default value

3.4.2 Examples

3.4.2.1 CASCADE

If a department is deleted or its ID is changed, related student records are automatically updated or removed.

```
CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    dept_id INTEGER,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

3.4.2.2 SET NULL

If a department is deleted, the student remains, but `dept_id` becomes `NULL`.

```
CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    dept_id INTEGER,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```