



Lab Number: 05

Course Code: CL2005

Course Title: Database Systems - Lab

Semester: Spring 2026

Agenda: DQL (Cont.)
Joining Tables, Set Operations,
Introduction to MySQL

Instructor: Muhammad Mehdi

Email: muhammad.mehdi@nu.edu.pk

Office: Rafaqat Lab



Table of Contents

1 SQL Data Query Language (DQL)	1
2 Joining Tables	1
2.1 Why Joining Tables is Necessary	1
2.2 Components of a Join	2
2.3 SQL JOIN Syntaxes	2
2.3.1 NATURAL JOIN	2
2.3.2 JOIN ... USING	3
2.3.3 JOIN ... ON	3
2.3.4 Implicit Join	5
2.4 Types of JOINS	5
2.4.1 INNER JOIN	6
2.4.2 LEFT JOIN (LEFT OUTER JOIN)	6
2.4.3 RIGHT JOIN (RIGHT OUTER JOIN)	7
2.4.4 FULL JOIN (FULL OUTER JOIN)	7
2.4.5 CROSS JOIN	8
2.4.6 SELF JOIN	9
3 SQL Set Operations	9
3.1 UNION	10
3.2 UNION ALL	10
3.3 INTERSECT	10
3.4 EXCEPT	11



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 Joining Tables

Real-world relational databases store related information across multiple tables to reduce redundancy and improve consistency.

To retrieve meaningful combined information, SQL provides two primary mechanisms:

- **JOIN operations:** combine rows from two or more tables **horizontally**, based on related columns
- **Set operations:** combine complete result sets **vertically** (e.g., **UNION**, **INTERSECT**)

2.1 Why Joining Tables is Necessary

In relational database design:

- Data is normalized.
- Redundant information is removed.
- Related data is stored in separate tables.

Consider the following two related tables:

students
roll number (PK)
name
department (FK)

departments
code (PK)
name

If we want to display student names along with their department names, the information must be combined from both tables.

This is achieved using a **JOIN**, which allows you to combine rows from two or more tables based on a related column.



2.2 Components of a Join

A **join** in SQL requires the following components:

1. **Two or more tables**: the tables whose data you want to combine
2. **A join type**: specifies how rows from the tables are matched (e.g., **INNER JOIN**, **LEFT JOIN**)
3. **A join condition**: defines how the rows relate to each other

The join condition typically involves comparing:

- A **foreign key** in one table
- The corresponding **primary key** in another table

General Syntax:

The *JOIN_TYPE* determines which rows are included in the result, and the *JOIN_CONDITION* ensures only related rows are combined.

```
SELECT columns  
FROM table1  
JOIN_TYPE table2  
JOIN_CONDITION;
```

2.3 SQL JOIN Syntaxes

SQL provides multiple syntactic approaches to joining tables. These are not separate commands but variations in query structure.

2.3.1 NATURAL JOIN

A **NATURAL JOIN** returns all rows where the tables have matching values in columns with the same name. It automatically eliminates duplicate columns from the result set. This type of join is used when two tables share one or more common columns that have identical names and compatible data types.

It performs the following steps:

- Identifies the common column(s) by finding columns that have the same name and compatible data types in both tables
- Returns only the rows where the values in those common column(s) match



- If no common columns exist, it returns the Cartesian product (relational product) of the two tables

General Syntax:

```
SELECT columns  
FROM table1  
NATURAL JOIN table2;
```

Example:

```
SELECT *  
FROM students  
NATURAL JOIN departments;
```

2.3.2 JOIN ... USING

A second way to write a join is by using the **USING** keyword. This form returns only the rows where the values in the specified column match in both tables. The column listed in the **USING** clause must exist in both tables and must have the same name.

General Syntax:

```
SELECT columns  
FROM table1  
JOIN table2 USING (common_column);
```

Example:

```
SELECT *  
FROM students  
JOIN departments USING (dept_id);
```

2.3.3 JOIN ... ON

The most recommended and widely used form to express a join, especially when the tables do not share common column names, is to use the **JOIN ... ON** clause. This form of join returns only the rows that satisfy the specified join condition.



The join condition typically includes an equality comparison between two columns. These columns may have different names, but they must have compatible data types to allow meaningful comparison.

General Syntax:

```
SELECT table1_columns, table2_columns, ...  
FROM table1  
JOIN table2,  
...  
ON table1_column = table2_column, ...;
```

Example 1: Columns with Different Names

If the referenced columns have unique names across the tables, table qualifiers are not strictly required:

```
SELECT name, code  
FROM students  
JOIN departments  
ON department = code;
```

Example 2: Columns with the Same Name

If the tables contain columns with identical names, each common column must be qualified using the format: *table_name.column_name*

```
SELECT students.name, departments.name  
FROM students  
JOIN departments  
ON students.department = departments.code;
```

Example 3: Using Table Aliases (Recommended)

A common and preferred approach is to assign aliases to each table and reference columns using the format: *alias.column_name*

```
SELECT s.name, d.name  
FROM students s  
JOIN departments d  
ON s.department = d.code;
```

2.3.4 Implicit Join

This is an older method of writing joins in SQL. Multiple tables are listed in the **FROM** clause, separated by commas, and the join condition is specified in the **WHERE** clause.

If the join condition is omitted, the query produces a **Cartesian product**, which returns every possible combination of rows from the listed tables.

Although this syntax is still valid in SQL, it is considered outdated. Modern SQL standards recommend using the explicit **JOIN ... ON** syntax for better readability, clarity, and maintainability.

General Syntax:

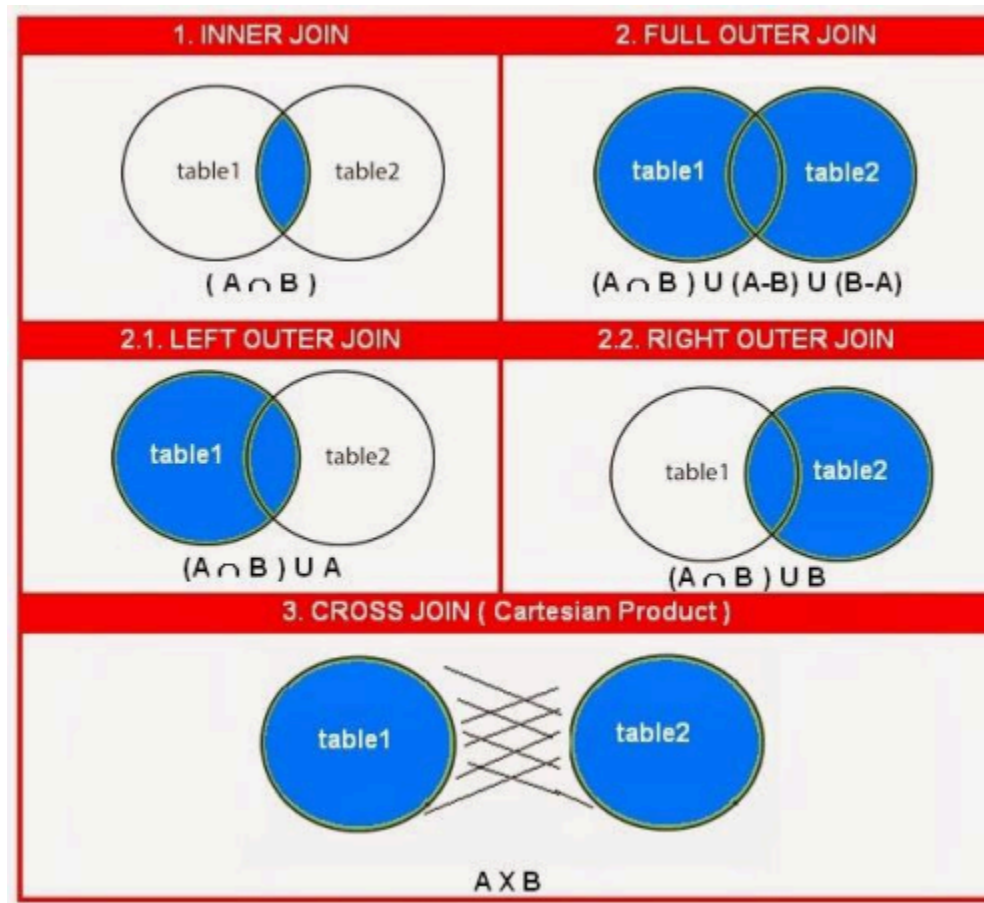
```
SELECT table1_columns, table2_columns, ...  
FROM table1, table2, ...  
WHERE table1_column = table2_column, ...;
```

Example:

```
SELECT name, code  
FROM students, departments  
WHERE department = code;
```

2.4 Types of JOINS

JOIN type determines which rows are returned.



2.4.1 INNER JOIN

Returns only matching rows in both tables. Only rows satisfying the join condition appear.

This is the default JOIN behavior.

Example:

```
SELECT s.name, d.dept_name
FROM students s
INNER JOIN departments d
ON s.department = d.code;
```

2.4.2 LEFT JOIN (LEFT OUTER JOIN)

Returns:

- All rows from left table



- Matching rows from right table
- NULL where no match exists

Used when left table data must always be preserved.

Example:

```
SELECT s.name, d.dept_name
FROM students s
LEFT JOIN departments d
ON s.department = d.code;
```

2.4.3 RIGHT JOIN (RIGHT OUTER JOIN)

Returns:

- All rows from right table
- Matching rows from left table

SQLite Note:

This type of JOIN is **not supported in SQLite**, but its behavior can be achieved by swapping table positions and using LEFT JOIN.

Examples:

```
-- Not supported in SQLite
SELECT s.name, d.name
FROM students s
RIGHT JOIN departments d
ON s.department = d.code;

-- Equivalent to RIGHT JOIN in SQLite
SELECT s.name, d.name
FROM departments d
LEFT JOIN students s
ON d.code = s.department;
```

2.4.4 FULL JOIN (FULL OUTER JOIN)

Returns:



- All rows from both tables
- Matching where possible
- NULL where no match

SQLite Note:

This type of JOIN is **not supported in SQLite**, but it can be simulated by performing a set union operation between LEFT JOIN of table 1 and table 2 with LEFT JOIN of table 2 and table 1.

Examples:

```
-- Not supported in SQLite
SELECT s.name, d.name
FROM students s
FULL JOIN departments d
ON s.department = d.code;

-- Equivalent to FULL JOIN in SQLite
SELECT s.name, d.name
FROM students s
LEFT JOIN departments d
ON s.department = d.code
UNION
SELECT s.name, d.name
FROM departments d
LEFT JOIN students s
ON d.code = s.department;
```

2.4.5 CROSS JOIN

Produces Cartesian product. Every row from the first table is paired with every row from the second table.

It can be useful for generating combinations, testing etc.

Example:

```
SELECT *
FROM students
CROSS JOIN departments;
```



2.4.6 SELF JOIN

Joining a table with itself.

Used when:

- hierarchical relationships exist
- an entity references another entity of same type

Common use cases:

- manager-employee relationships
- prerequisite chains
- category hierarchies

Examples:

```
-- A TA who is a student himself manages other students
SELECT s.name AS student, ta.name AS teaching_assistant
FROM students s
INNER JOIN students ta
ON s.ta_roll_number = ta.roll_number;

-- A manager who is an employee himself manages other employees
SELECT e.name AS employee, m.name AS manager
FROM employees e
LEFT JOIN employees m
ON e.manager_id = m.id;
```

3 SQL Set Operations

Set operations combine entire query results vertically. Set operations combine the results of **two SELECT queries** into one result set, like set theory:

Unlike JOIN:

- JOIN combines columns.
- Set operations combine rows.



Set operations require:

- Same number of columns
- Compatible data types
- Same column order
- Column names in the result set come from the first SELECT statement

3.1 UNION

Combines results and removes duplicates. Duplicates are automatically removed.

Example:

```
SELECT name FROM cs_students
UNION
SELECT name FROM se_students;
```

3.2 UNION ALL

Combines results but keeps duplicates. Faster than UNION because no duplicate elimination.

Example:

```
SELECT name FROM cs_students
UNION ALL
SELECT name FROM se_students;
```

3.3 INTERSECT

Returns only common rows between two result sets. Returns only common rows between two result sets.

Example:

```
SELECT name FROM cs_students
INTERSECT
SELECT name FROM scholarship_students;
```



3.4 EXCEPT

Returns rows from first query not present in second query. Removes matching rows found in second query.

Example:

```
SELECT name FROM students
EXCEPT
SELECT name FROM graduated_students;
```