Joins

Contents

[1. Introduction 2](#_Toc186451076)

[SQL Joins 3](#_Toc186451077)

[Understanding Joins in Databases 5](#_Toc186451078)

[Understanding ON Clause vs WHERE Clause in SQL Joins 8](#_Toc186451079)

[2. Inner Join 10](#_Toc186451080)

[Simplifying Queries with Aliases 11](#_Toc186451081)

[Self Join 13](#_Toc186451082)

[Joining multiple tables 18](#_Toc186451083)

[Compound Join 22](#_Toc186451084)

[3. Outer Joins 23](#_Toc186451085)

[Left Outer Join 23](#_Toc186451086)

[Right Outer Join 25](#_Toc186451087)

[Full Outer Join 27](#_Toc186451088)

[USING Clause 29](#_Toc186451089)

[Natural Join 32](#_Toc186451090)

[Cross Join 33](#_Toc186451091)

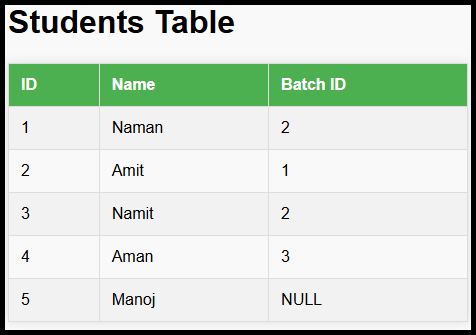
[Implicit Join and its Drawbacks 40](#_Toc186451092)

# Introduction

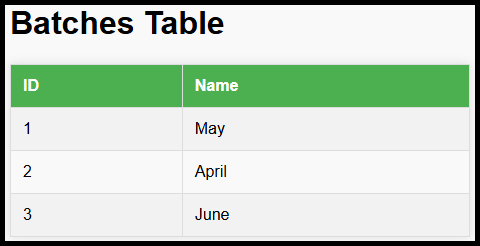
### Problem Setup

You have two tables:

1. **Students Table**
   * Columns: ID, Name, BatchID



1. **Batches Table**
   * Columns: ID, Name



### Goal

For every student, retrieve the **student name** and the **batch name**.

### Programming Analogy

Before SQL joins, the problem was approached programmatically:

#### **Problem Statement**

You need to get a combined list with **student names** and their **batch names** from two separate tables (or lists):

* students (contains student details like ID, name, and batchID)
* batches (contains batch details like ID and name).

|  |
| --- |
| initialize an empty list called `answer`  for each `student` in `students`:      for each `batch` in `batches`:          if `student.batchID` == `batch.ID`:              add a tuple (`student.name`, `batch.name`) to `answer` |

**Time Complexity**: O(N×M), where N is the number of students, and M is the number of batches.

## SQL Joins

* SQL simplifies this process using **joins**, which:
  + Combine rows from two tables into a single "virtual table."
  + Match rows based on a specified condition (e.g., Student.BatchID = Batch.ID).

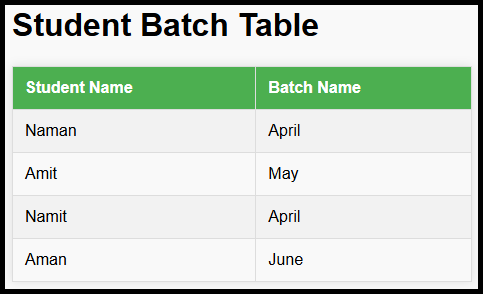
To solve the problem:

|  |
| --- |
| SELECT Students.Name, Batches.Name  FROM Students      JOIN Batches      ON Students.BatchID = Batches.ID; |

Explanation:

* FROM Students: Begin with the Students table.
* JOIN Batches: Combine rows from the Batches table.
* ON Students.BatchID = Batches.ID: Specify the condition for matching rows.

Result:

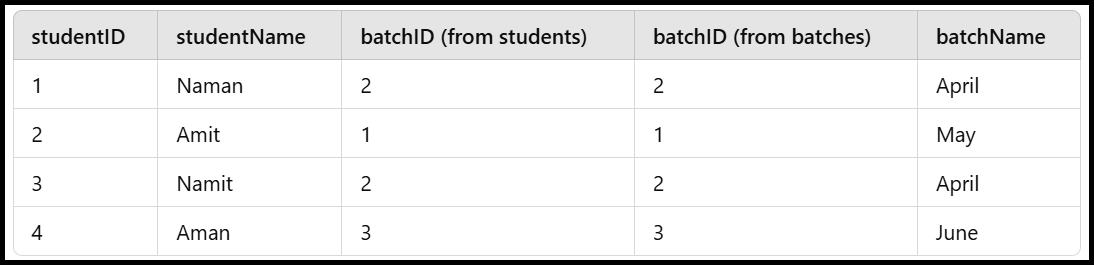


### NULL Handling in Joins

* The row with BatchID = NULL (e.g., Manoj) is excluded in **JOIN** because NULL cannot be matched to any value.

### Virtual Table Concept

* When the JOIN is executed, a **virtual table** is created internally with all columns from both tables.:
  + Combined columns from both tables.
  + Matched rows based on the ON condition.
* For example:



## **Understanding Joins in Databases**

#### **1. Basics of Joins**

* Joins combine rows from two tables based on a condition.
* If **no condition** is applied:
  + Every row from the first table will match every row from the second table.
  + Maximum rows in the result = N×M, where N = rows in Table A, M = rows in Table B.

#### **2. Column Name Conflicts**

* If both tables have columns with the same name:
  + Prefix the column with the table name to avoid ambiguity (e.g., students.name, batches.name).
  + This ensures clarity in specifying which column to reference.

#### **3. Temporary Memory Usage**

* Joins create a **virtual table** internally:
  + Takes temporary memory space during query execution.
  + The memory is released after the query output is returned.
  + No permanent storage space is consumed.

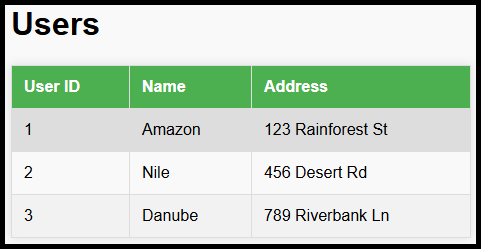
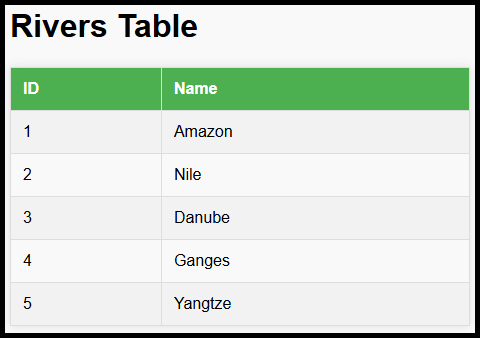
#### **4. Foreign Keys Are Not Mandatory**

* A join does not require foreign keys or pre-defined relationships between tables.
* The only requirement is that the **join condition** must be satisfied.

### **Example Problem: Joining Unrelated Tables**

#### Problem Statement:

* There are two tables:
  1. users (columns: ID, name, address).
  2. rivers (columns: ID, name).
* **Goal**: Find the address of users whose name is the same as the name of a river.

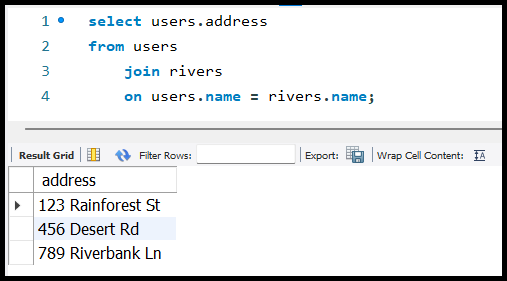


#### Approach:

1. **Understand the Logic**:
   * Combine the users and rivers tables based on the condition: users.name = rivers.name.
   * For the resulting rows, print the users.address.
2. **SQL Query**:

|  |
| --- |
| SELECT users.address  FROM users      JOIN rivers      ON users.name = rivers.name; |

1. **Explanation**:
   * JOIN rivers combines the users and rivers tables.
   * ON users.name = rivers.name specifies the join condition.
   * SELECT users.address fetches the required column (address) from the resulting rows.



### **Key Takeaways**

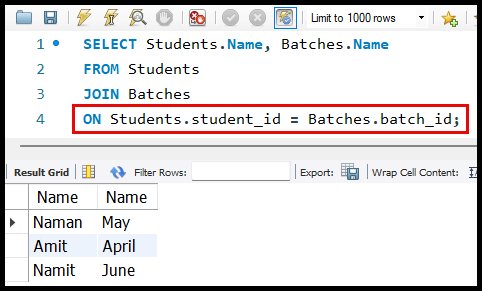
1. Joins depend only on the **condition** specified, not on pre-defined relationships like foreign keys.
2. When using joins:
   * Clarify ambiguous column names by prefixing with table names.
   * Understand that joins create temporary virtual tables, using memory during execution.
3. In cases of unrelated tables, define the required condition to combine rows meaningfully.

## Understanding ON Clause vs WHERE Clause in SQL Joins

1. **Key Concept: Using** ON **vs** WHERE **in Joins**
   * ON **Clause**:
     + Used to specify the condition for combining rows from different tables during the join process.
     + Filters rows before creating the virtual table, reducing its size and making the query more efficient.
   * WHERE **Clause**:
     + Used to filter rows after the virtual table is created.
     + If the condition for the join is moved to the WHERE clause, all possible combinations of rows (Cartesian product) are created first, resulting in a large and inefficient virtual table.
2. **Example Query Using** ON **Clause**:

|  |
| --- |
| SELECT Students.Name, Batches.Name  FROM Students      JOIN Batches      ON Students.student\_id = Batches.batch\_id; |

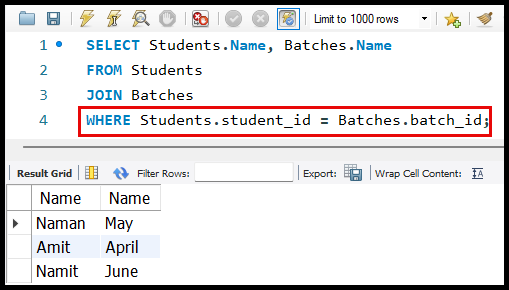
* + The join condition (Students.BatchID = Batches.ID) is applied while combining rows from the two tables.
  + This reduces the number of rows in the virtual table early, improving query performance.



1. **Example Query Using** WHERE **Clause**:

|  |
| --- |
| SELECT Students.Name, Batches.Name  FROM Students      JOIN Batches  WHERE Students.student\_id = Batches.batch\_id; |

* + Here, all possible combinations of rows (Cartesian product) are created first, and only then are the rows filtered based on the condition (Students.BatchID = Batches.ID).
  + This increases the size of the virtual table, making the query slower.



1. **Difference in Query Execution**:
   * **With** ON **Clause**:
     + Combines rows based on the condition directly during the join process.
     + Resulting virtual table contains fewer rows.
     + Query is faster and more optimized.
   * **With** WHERE **Clause**:
     + Creates a virtual table with all possible row combinations (Cartesian product).
     + Filters rows after the join, leading to unnecessary computation and slower query execution.
2. **Performance Impact**:
   * Using the ON clause ensures the virtual table is smaller in size, reducing memory usage and improving the speed of the query.
   * Using the WHERE clause for join conditions can lead to slower queries, especially for tables with a large number of rows.
3. **Illustration of Virtual Table Creation**:
   * If the **Students** table has N rows and the **Batches** table has M rows:
     + Without a condition (ON or WHERE), the virtual table will have N × M rows (Cartesian product).
     + With the ON clause, only rows satisfying the join condition are included in the virtual table.
4. **General Recommendation**:
   * Always use the ON clause for join conditions to avoid creating unnecessarily large virtual tables and ensure better query performance.

# Inner Join

**Definition**:  
An INNER JOIN returns rows from two tables where the specified join condition is met.

* **Default Behaviour**:
  + The JOIN keyword, by default, implies an INNER JOIN.
  + Writing INNER JOIN explicitly is optional, as JOIN alone functions the same way.
* **Example Queries**:

|  |  |
| --- | --- |
| SELECT Students.Name  FROM Students      JOIN Batches      ON Students.BatchID = Batches.ID; | SELECT Students.Name  FROM Students      INNER JOIN Batches      ON Students.BatchID = Batches.ID; |

* + Both queries are equivalent.

## Simplifying Queries with Aliases

* **Purpose of Aliases**:
  + Shorten table names to improve readability and reduce repetition in queries.
  + Minimize the chances of typos or bugs caused by repeated usage of full table names.
* **Syntax for Aliases**:
  + Assign an alias after the table name, either using the AS keyword (optional) or directly with a space.
  + Aliases do not require the AS keyword, though it can be included for clarity.
* **Example with Aliases**:

|  |
| --- |
| SELECT S.Name  FROM Students S      JOIN Batches B      ON S.BatchID = B.ID; |

* + Here, S represents the Students table, and B represents the Batches table.
  + This query is easier to read and less prone to errors.

**Benefits of Using Aliases in Joins**:

* **Improved Readability**:
  + Queries are less cluttered and more concise.
* **Simplified Maintenance**:
  + Reduces the effort required to modify or debug queries.
* **Lower Risk of Errors**:
  + Minimizes typing errors when referencing table names repeatedly.

**Why Use** INNERJOIN**?**

* The INNER JOIN only includes rows that satisfy the join condition, making it efficient and relevant for most scenarios where related data is required from two tables.
* Explanation of why it is called "INNER JOIN" will be provided later in subsequent lessons.

**General Best Practices**:

* Use INNER JOIN for clarity when teaching or documenting, even though it is optional.
* Always use table aliases in complex queries, especially when joining multiple tables.

## Self Join

#### **What is a Self Join?**

* A **Self Join** is a type of inner join where a table is joined with itself.
* It is used when rows from the same table are related to each other.
* A self join is essentially a regular join, but it uses the same table twice with different aliases.

#### **Example Scenario:**

* **Students Table:**
  + Each student has a BuddyID which references the ID of another **student** in the same table.
  + The goal is to print the **Student Name** and their **Buddy Name**.

#### **How to Solve the Problem with a Separate Buddies Table?**

* Assume there are two separate tables:
  + **Students**: Contains student data.
  + **Buddies**: Contains buddy data.
* The query would look like

|  |
| --- |
| SELECT S.Name AS StudentName, B.Name AS BuddyName  FROM Students S      JOIN Buddies B ON S.BuddyID = B.ID; |

#### **How to Solve the Problem Using Self Join?**

1. Since there is no separate Buddies table, we join the Students table with itself.
2. Each student is their buddy's reference, so we use a self join to match students and their buddies.

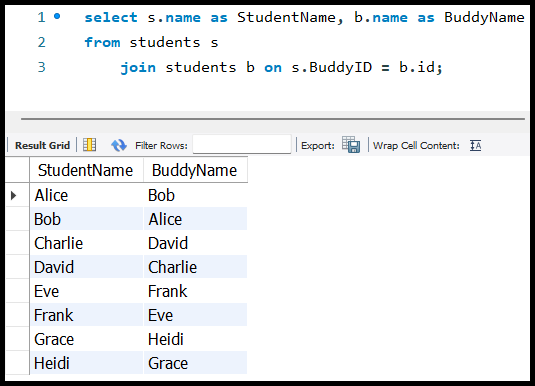
**Steps to Implement Self Join:**

1. Assign **aliases** to differentiate the two instances of the same table:
   * First instance: S (represents the student).
   * Second instance: B (represents the buddy).
2. Join the table on the condition:
   * S.BuddyID = B.ID (the BuddyID in the first instance matches the ID in the second instance).
3. Select the required columns:
   * S.Name (Student Name).
   * B.Name (Buddy Name).

Pseudo Code:

|  |
| --- |
| # Iterate over each student in the table  for student in students:      # Find the corresponding buddy by matching BuddyID with ID      for buddy in students:          if student.BuddyID == buddy.ID:              # Add the pair of student name and buddy name to results              results.append({                  "StudentName": student.Name, "BuddyName": buddy.Name}) |

|  |
| --- |
| SELECT S.Name AS StudentName, B.Name AS BuddyName  FROM Students S      JOIN Students B ON S.BuddyID = B.ID; |



#### **Explanation of Query:**

* **Aliases:**
  + S: Represents the student (outer table).
  + B: Represents the buddy (inner table).
* **Join Condition:**
  + S.BuddyID = B.ID: Matches the BuddyID of a student with the ID of another student.
* **Columns Selected:**
  + S.Name: Prints the name of the student.
  + B.Name: Prints the name of the buddy.

### **Self Join with Employee-Manager Example**

#### **Example Scenario**

* **Table Used**: employees from the SQL HR database.
* Each employee has a column called reports\_to, which contains the employee\_id of their manager.
* Goal: For every employee, print their name and their manager's name.

#### **Steps to Solve**

1. **Alias the Table**:
   * Use aliases to distinguish the two instances of the employees table:
     + E (for employee)
     + Mgr (for manager)
2. **Join Condition**:
   * The reports\_to column of the employee (E.reports\_to) should match the employee\_id column of the manager (Mgr.employee\_id).
3. **Select Columns**:
   * Select the first\_name of the employee (E.first\_name) and the first\_name of the manager (Mgr.first\_name).
4. **Query Syntax**:

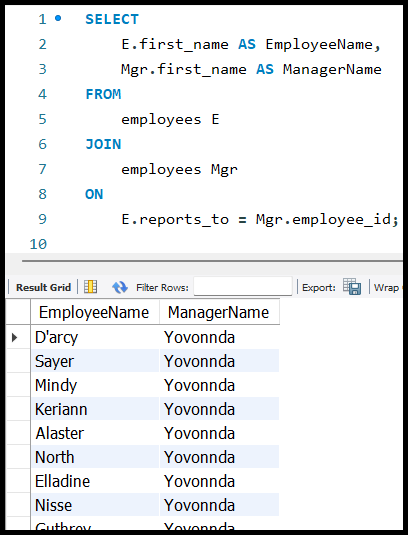
|  |
| --- |
| SELECT      E.first\_name AS EmployeeName,      Mgr.first\_name AS ManagerName  FROM employees E JOIN employees Mgr      ON      E.reports\_to = Mgr.employee\_id; |

#### **Output Explanation**

* Each row in the result displays:
  + The name of the employee.
  + The name of the manager they report to.

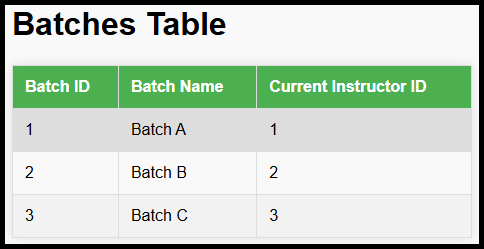
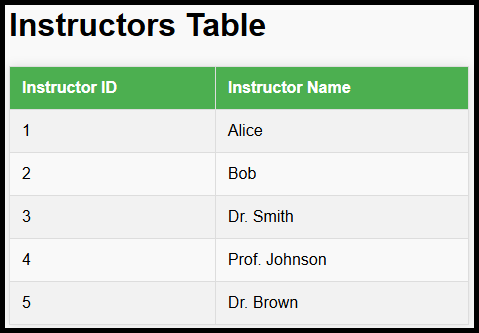
#### **Handling** NULL **Values**

* **Why doesn't the CEO appear in the output?**
  + The CEO has no manager, so their reports\_to value is NULL.
  + NULL cannot be matched with any value, even another NULL.



## Joining multiple tables

#### **Schema Overview**

1. Students **Table**:
   * StudentID: Unique ID for each student.
   * StudentName: Name of the student.
   * BatchID: ID of the batch the student is part of.
2. Batches **Table**:
   * BatchID: Unique ID for each batch.
   * BatchName: Name of the batch.
   * CurrentInstructorID: ID of the instructor currently teaching the batch.
3. Instructors **Table**:
   * InstructorID: Unique ID for each instructor.
   * InstructorName: Name of the instructor.

### **Question**

For every student, print the name of the instructor teaching that student.

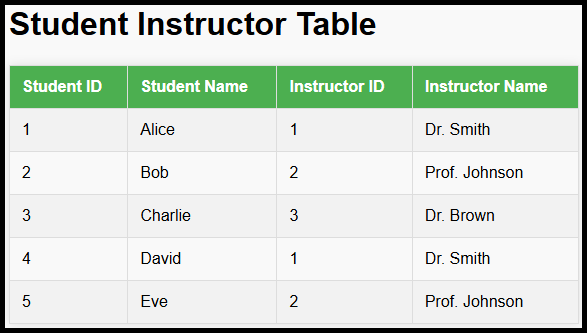
### **Key Insight**

There is no direct link between a Student and an Instructor. To find the instructor teaching a specific student, we need to use an intermediary table:

* **Path**: Students → Batches → Instructors.
* **Link**:
  + Students.BatchID → Batches.BatchID.
  + Batches.CurrentInstructorID → Instructors.InstructorID.

### **Approach**

1. **Visualize the Ideal Table**:
   * Ideally, we need a table with:



* To create this table, we must use joins to combine information from the Students, Batches, and Instructors tables.

### **Steps to Solve**:

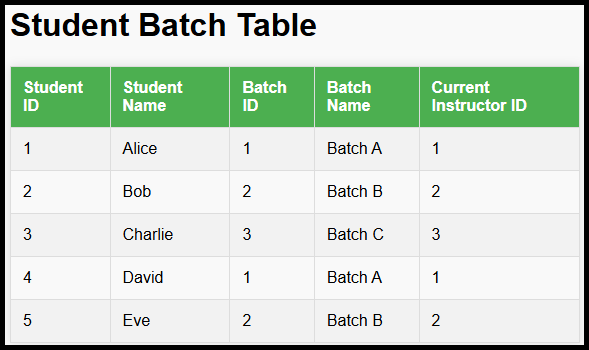
* First, join the Students table with the Batches table on BatchID.
* Then, join the resulting table with the Instructors table on CurrentInstructorID.

#### **SQL Query**

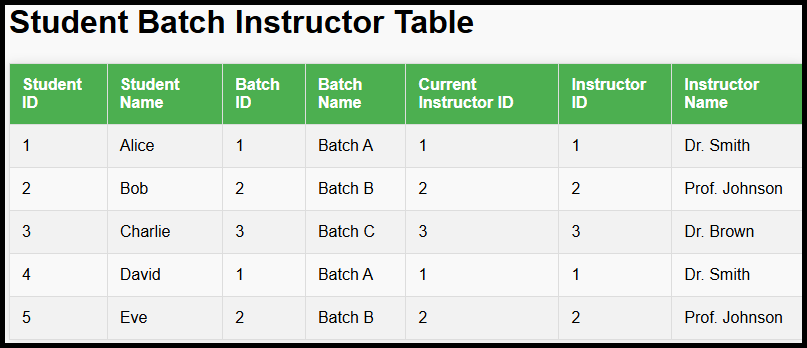
|  |
| --- |
| SELECT      S.StudentName AS Student\_Name,      I.InstructorName AS Instructor\_Name  FROM      Students S  JOIN      Batches B ON S.BatchID = B.BatchID  JOIN      Instructors I ON B.CurrentInstructorID = I.InstructorID; |

#### **Virtual Table Generated Internally**

* After the first join (Students and Batches):



* After the second join (Batches and Instructors):



* If no direct link exists between two tables, look for an intermediary table that can bridge them.
* Use joins to combine data across multiple tables and create a virtual table as needed.
* The sequence of joins matters, especially when there are multiple intermediaries.

### Pseudocode for Mapping Students to Their Instructors

* The task involves iterating through the Students, Batches, and Instructors data and applying conditions to map each student to their corresponding instructor.

|  |
| --- |
| # Input: Lists of students, batches, and instructors  students = [      {"StudentID": 1, "StudentName": "Alice", "BatchID": 101},      {"StudentID": 2, "StudentName": "Bob", "BatchID": 102},      # Add more students as needed  ]  batches = [      {"BatchID": 101, "BatchName": "Math", "CurrentInstructorID": 201},      {"BatchID": 102, "BatchName": "Science", "CurrentInstructorID": 202},      # Add more batches as needed  ]  instructors = [      {"InstructorID": 201, "InstructorName": "Dr. Smith"},      {"InstructorID": 202, "InstructorName": "Dr. Jones"},      # Add more instructors as needed  ]  # Output: List of students with their instructors  result = []  # Nested loops to map students to instructors  for student in students:      for batch in batches:          # Check if the student's BatchID matches the BatchID in the batches table          if student["BatchID"] == batch["BatchID"]:              for instructor in instructors:                  # Check if the batch's CurrentInstructorID matches the InstructorID                  if batch["CurrentInstructorID"] == instructor["InstructorID"]:                      # Add the student-instructor mapping to the result                      result.append({                          "StudentName": student["StudentName"],                          "InstructorName": instructor["InstructorName"]                      })  # Print the result  for entry in result:      print(f"Student: {entry['StudentName']}, Instructor: {entry['InstructorName']}") |

## **Compound Join**

1. **Definition**:
   * A **compound join** allows joining tables based on multiple conditions.
   * These conditions can be combined using logical operators such as **AND** and **OR**.
2. **Key Points**:
   * Compound joins are an extension of basic joins but involve additional flexibility with conditions.
   * You can specify multiple conditions between the tables being joined.
3. **Syntax**:

|  |
| --- |
| SELECT      S.StudentID,      S.StudentName,      B.BatchName,      I.InstructorName  FROM      students S      JOIN batches B ON S.BatchID = B.BatchID      JOIN instructors I ON B.CurrentInstructorID = I.InstructorID      AND B.BatchName = 'Batch A'; |

1. **Usage**:
   * Combine conditions with logical operators:
     + **AND**: Both conditions must be true.
     + **OR**: At least one condition must be true.

# **Outer Joins**

##### **Inner Join Recap**

* An **inner join** only includes rows that meet the specified condition between two tables.
* **Key Points:**
  + Not all rows of the left table or right table are included in the final output.
  + Rows with no matches or rows containing NULL values in the join condition are excluded.

##### **Outer Joins Introduction**

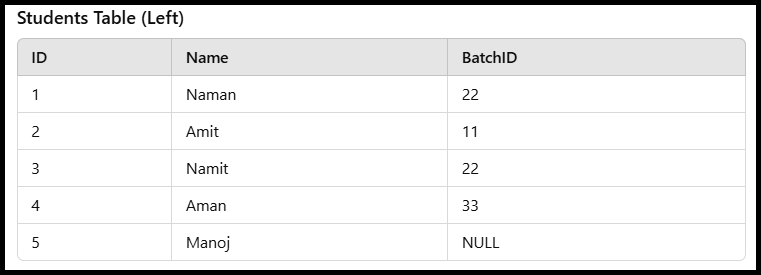
* Outer joins address the limitation of inner joins by ensuring that all rows from one or both tables are included in the output, even if there is no match in the other table.
* **Types of Outer Joins:**
  1. **Left Outer Join**
  2. **Right Outer Join**
  3. **Full Outer Join**

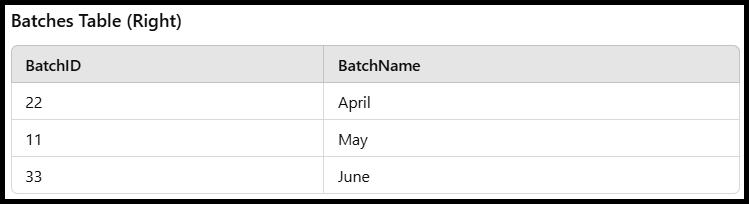
## **Left Outer Join**

* **Definition:**
  + Begins as an inner join but ensures all rows from the left table are included in the output.
  + If a row from the left table has no match in the right table, it is added to the result with NULL values for the right table columns.
* **Syntax:**

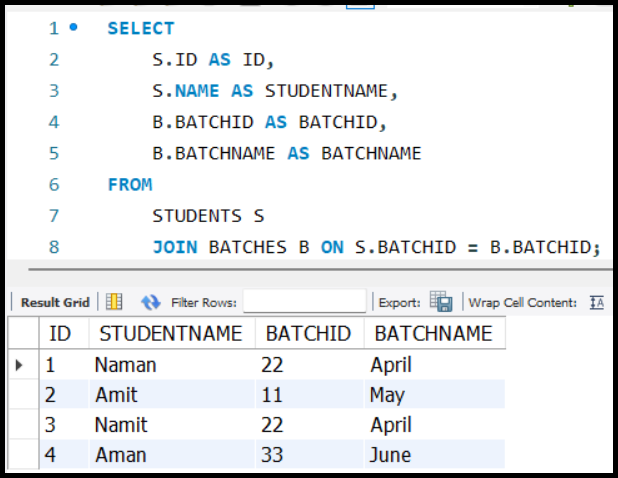
|  |
| --- |
| SELECT      columns  FROM      TableA      LEFT JOIN TableB ON TableA.Column = TableB.Column; |

* + The keyword OUTER is optional: LEFT JOIN and LEFT OUTER JOIN are equivalent.
* **Example Workflow:**
  + **Given Tables:**

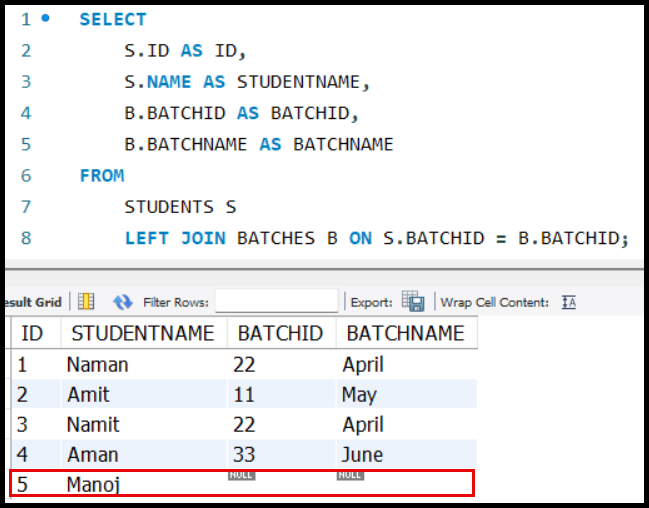




Inner Join



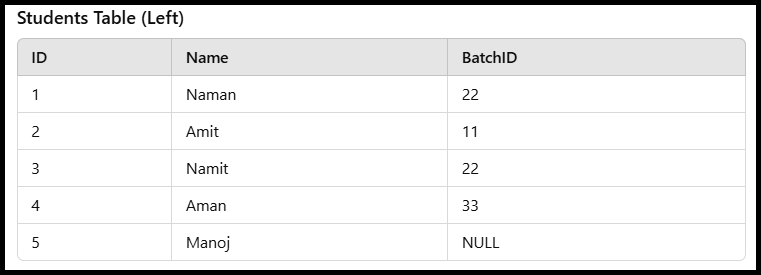
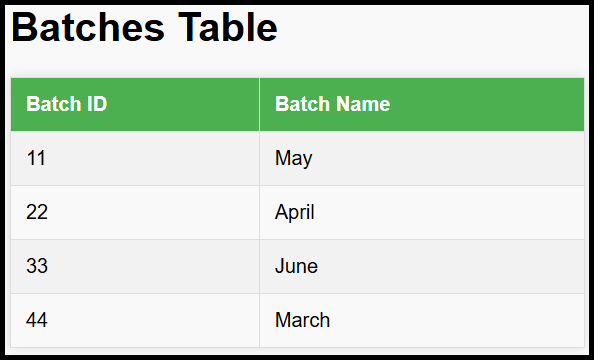
Left Outer Join

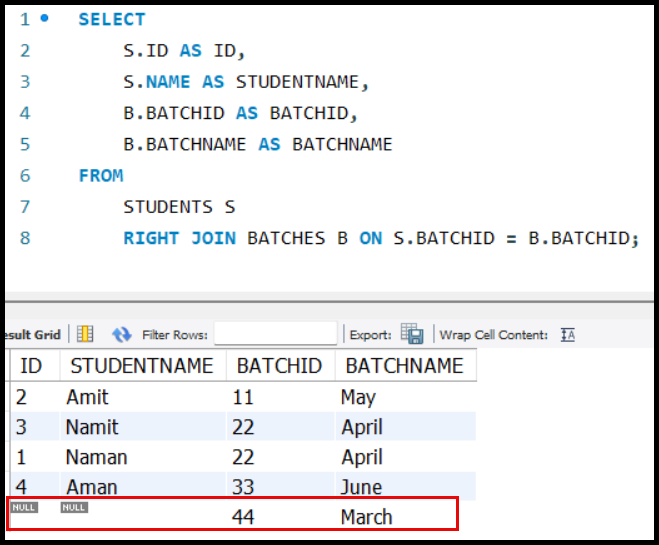


## **Right Outer Join**

* **Definition:**
  + Ensures all rows from the right table are included in the output.
  + If a row from the right table has no match in the left table, it is added with NULL values for the left table columns.

|  |
| --- |
| SELECT      columns  FROM      TableA  RIGHT JOIN      TableB  ON      TableA.Column = TableB.Column; |

Tables Used:

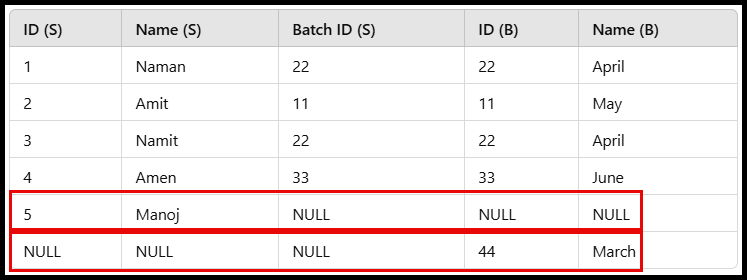


|  |
| --- |
| SELECT      S.ID AS ID,      S.NAME AS STUDENTNAME,      B.BATCHID AS BATCHID,      B.BATCHNAME AS BATCHNAME  FROM      STUDENTS S      RIGHT JOIN BATCHES B ON S.BATCHID = B.BATCHID; |

## **Full Outer Join**

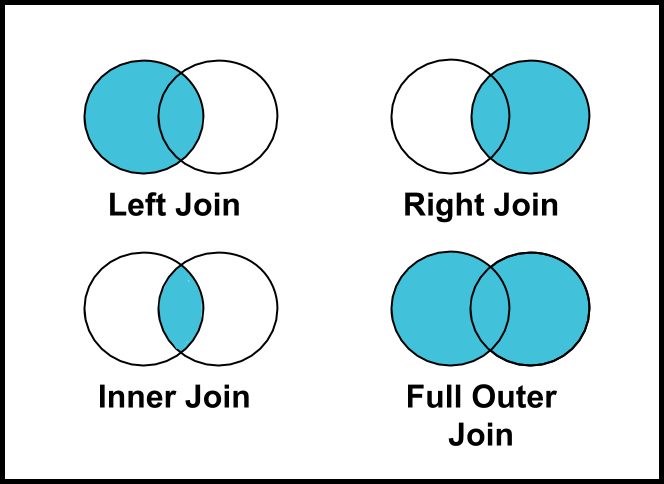
* **Definition:**
  + Ensures all rows from both tables are included in the output.
  + Rows with no match in either table are filled with NULL values for the missing columns.
* **Syntax:**

|  |
| --- |
| SELECT      columns  FROM      TableA  FULL OUTER JOIN      TableB  ON      TableA.Column = TableB.Column; |



* MySQL does not support FULL OUTER JOIN directly. However, you can achieve the same result using a combination of LEFT JOIN and RIGHT JOIN with UNION.

|  |
| --- |
| SELECT      S.ID AS ID,      S.NAME AS STUDENTNAME,      B.BATCHID AS BATCHID,      B.BATCHNAME AS BATCHNAME  FROM      STUDENTS S      LEFT JOIN BATCHES B ON S.BATCHID = B.BATCHID  UNION  SELECT      S.ID AS ID,      S.NAME AS STUDENTNAME,      B.BATCHID AS BATCHID,      B.BATCHNAME AS BATCHNAME  FROM      STUDENTS S      RIGHT JOIN BATCHES B ON S.BATCHID = B.BATCHID; |



## USING Clause

* Joins are typically written with an **ON** clause specifying the condition for joining two tables.

|  |
| --- |
| SELECT \*  FROM STUDENTS  JOIN INSTRUCTORS  ON STUDENTS.BATCH\_ID = INSTRUCTORS.BATCH\_ID; |

* The ON clause defines the relationship between columns from two tables.

#### Motivation for the USING Clause:

* Often, tables have columns with the same name (e.g., batch\_id in students and instructors).
* In such cases, the join condition is usually based on these columns.

#### What is the USING Clause?

* A syntactic sugar introduced in SQL to simplify writing joins for columns with the same name.
* Reduces the verbosity of join queries by removing the need to explicitly specify the table names.
* Example with USING clause:

|  |
| --- |
| SELECT S.ID AS STUDENT\_ID,      S.NAME AS STUDENT\_NAME,      S.BATCHID AS BATCH\_ID,      I.ID AS INSTRUCTOR\_ID,      I.NAME AS INSTRUCTOR\_NAME  FROM STUDENTS S      JOIN INSTRUCTORS I  USING (BATCHID); |

* Equivalent to:

|  |
| --- |
| SELECT S.ID AS STUDENT\_ID,      S.NAME AS STUDENT\_NAME,      S.BATCHID AS BATCH\_ID,      I.ID AS INSTRUCTOR\_ID,      I.NAME AS INSTRUCTOR\_NAME  FROM STUDENTS S      JOIN INSTRUCTORS I      ON S.BATCHID = I.BATCHID; |

#### Key Points:

* USING(COLUMN\_NAME) is a shorthand for:

TABLE1.COLUMN\_NAME = TABLE2.COLUMN\_NAME

* **No Performance Improvement:** The USING clause is purely for readability and code simplicity.
* **Limitations:**
  + Both tables must have a column with the **same name** to use USING.
  + If the column names differ, the USING clause cannot be applied.
  + Example:

SELECT \*

FROM TABLE1

    JOIN TABLE2

USING (COMMON\_COLUMN\_NAME);

* + If column names are different:

SELECT \*

FROM TABLE1

    JOIN TABLE2

    ON TABLE1.COLUMN1 = TABLE2.COLUMN2;

#### Practical Example:

1. **Scenario:**
   * students table: Contains student details with a batch\_id.
   * instructors table: Contains instructor details with a batch\_id.
   * Query: Find the instructor of each student.
2. **Solution Using** USING**:**

SELECT S.ID AS STUDENT\_ID,

    S.NAME AS STUDENT\_NAME,

    S.BATCHID AS BATCH\_ID,

    I.ID AS INSTRUCTOR\_ID,

    I.NAME AS INSTRUCTOR\_NAME

FROM STUDENTS S

    JOIN INSTRUCTORS I

USING (BATCHID);

#### Analogy with Java:

* The USING clause is like Java's enhanced for-each loop for simplicity:
  + Traditional loop:

for (int i = 0; i < list.size(); i++) {

    System.out.println(list.get(i));

}

* + Simplified loop:

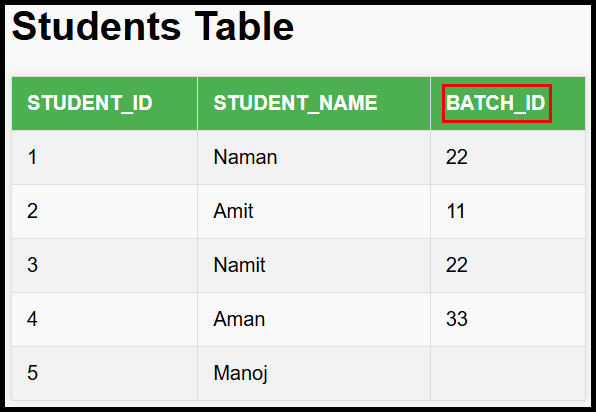
for (String item : list) {

    System.out.println(item);

}

* Similarly, the USING clause reduces repetitive code when the join condition is based on common column names.

## **Natural Join**

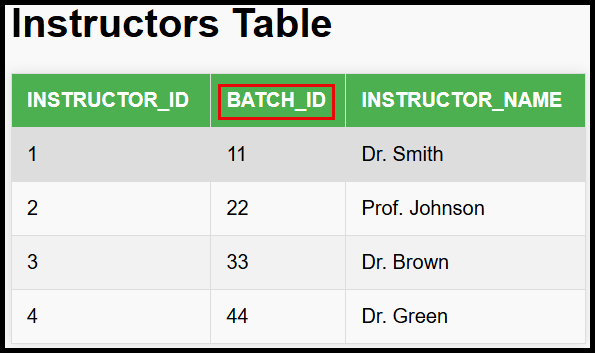
* **Purpose**: Another syntactic sugar for JOIN operations when joining two tables based on all columns with the same name.
* **Definition**:
  + A NATURAL JOIN automatically matches all columns with the same name and joins them on equality of their values.
* **Example**:

SELECT \*

FROM INSTRUCTORS

NATURAL JOIN STUDENTS;

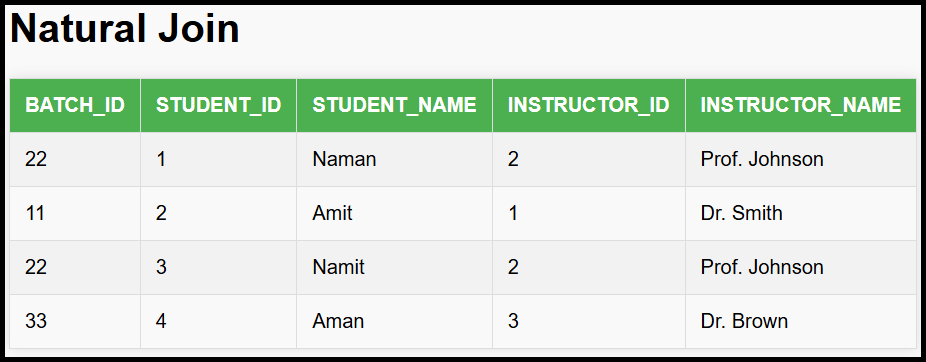
* + Equivalent to:

SELECT \*

FROM INSTRUCTORS

    JOIN STUDENTS

    ON INSTRUCTORS.BATCH\_ID = STUDENTS.BATCH\_ID



* **Caution**:
  + A NATURAL JOIN considers **all columns** with the same name, which might not always be the desired behavior.
  + Example Problem:
    - If BATCH\_ID exist in both tables, the join condition will include all three columns, which might not be correct.
* **Best Practice**:
  + Ensure column names are distinct for unrelated data. For example:
    - Use INSTRUCTOR\_ID and STUDENT\_ID instead of just ID.
    - Use INSTRUCTOR\_NAME and STUDENT\_NAME instead of just NAME.
  + This avoids unintended join conditions in a NATURAL JOIN.

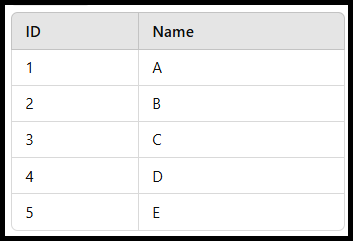
## Cross **Join**

#### **What is a Cross Join?**

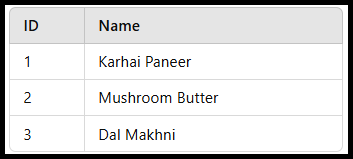
* **Definition**: A cross join pairs every row from one table with every row from another table.
* **No Condition**: Cross join does not require any condition to join the tables.
* **Output Rows**: If table A has M rows and table B has N rows, the resulting table will have M×N rows.

#### **Example of Cross Join**

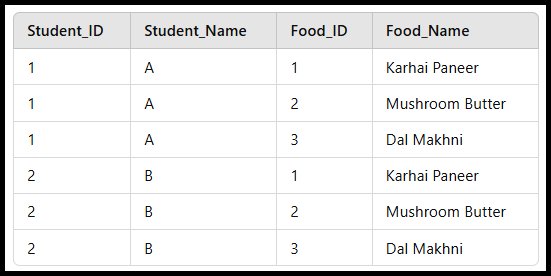
* **Tables**:
  + students table:



* + food\_items table:



* + **Result of Cross Join** (students CROSS JOIN food\_items):

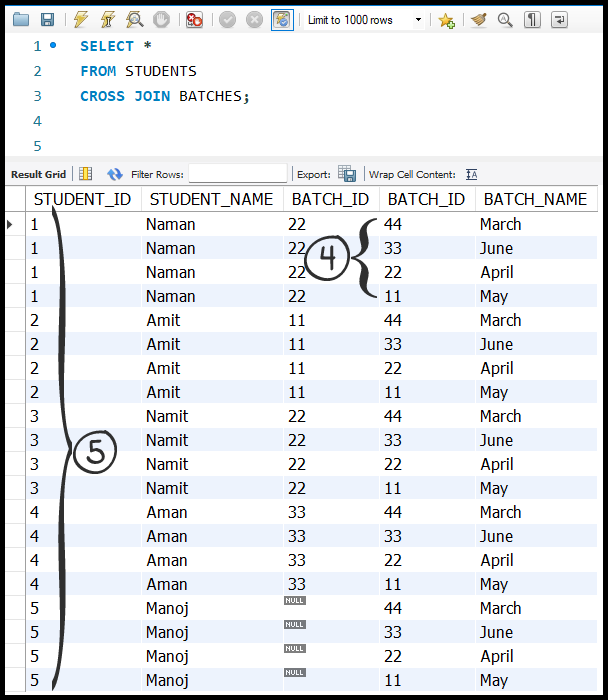


* + Total Rows: 5×3=15.
* Use Case:
  + To create combinations or pairs of every item from two sets.
  + Example: Assigning every person in a party to bring specific food items.
* Example

SELECT \*

FROM STUDENTS

CROSS JOIN BATCHES;



* + Total Rows: 5×4=20.

**Cross Join vs. Full Join**

* **Cross Join**:
  + Produces M×NM \times NM×N rows (all possible combinations).
  + No condition required.
  + Example use: When all combinations are needed for analysis.
* **Full Join**:
  + Combines rows from two tables based on a condition and includes:
    - Rows that match the condition.
    - Rows that do not match from both tables.
  + Produces fewer rows than a cross join.

**Key Differences:**

| **Feature** | **Cross Join** | **Full Join** |
| --- | --- | --- |
| Condition | No condition required | Requires a matching condition. |
| Rows Produced | M×N (all pairs) | Rows that match condition + unmatched rows. |
| Use Case | All combinations needed | Combine rows while preserving unmatched rows. |

#### **Join Without Condition**

* + A join without a condition behaves like a **cross join**.

SELECT \*

FROM STUDENTS

JOIN FOOD\_ITEMS;

* + This results in M×N rows as no condition is specified.

#### **Efficiency Considerations**

* When a condition is applied **on the join**, fewer rows are produced in the virtual table, improving efficiency:

SELECT \*

FROM STUDENTS S

JOIN BATCHES B

ON S.BATCH\_ID = B.BATCH\_ID;

* Using WHERE after a cross join creates an intermediate table with **M×N rows**, which is less efficient:

SELECT \*

FROM STUDENTS S

JOIN BATCHES B

WHERE S.BATCH\_ID = B.BATCH\_ID;

### **Full Join and Comparison with Cross Join**

#### **What Happens in a Full Join?**

* **Definition**: A full join combines rows from two tables:
  + Rows that satisfy the join condition.
  + Rows that do not satisfy the condition, with unmatched rows filled with NULL.
* **Behaviour with a False Condition**:
  + If the join condition is always false, no rows will match.
  + The output will include:
    - All rows from the first table, with NULL for the columns of the second table.
    - All rows from the second table, with NULL for the columns of the first table.

#### **Example: Full Join with a False Condition**

| students | |
| --- | --- |
| **ID** | **Name** |
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| 5 | E |

| food\_items | |
| --- | --- |
| **ID** | **Name** |
| 1 | Karhai Paneer |
| 2 | Mushroom Butter |
| 3 | Dal Makhni |

* **Tables**:
  + students table:
  + food\_items table:
* **Full Join with Always False Condition**:

SELECT \*

FROM STUDENTS

FULL JOIN FOOD\_ITEMS

ON 1 = 0; -- ALWAYS FALSE CONDITION

* **Result**:

| **Student\_ID** | **Student\_Name** | **Food\_ID** | **Food\_Name** |
| --- | --- | --- | --- |
| 1 | A | NULL | NULL |
| 2 | B | NULL | NULL |
| 3 | C | NULL | NULL |
| 4 | D | NULL | NULL |
| 5 | E | NULL | NULL |
| NULL | NULL | 1 | Karhai Paneer |
| NULL | NULL | 2 | Mushroom Butter |
| NULL | NULL | 3 | Dal Makhni |

#### **Difference Between Cross Join and Full Join**

* **Cross Join**:
  + Produces M×N rows (all combinations of rows from both tables).
  + No condition is required.
  + Example: Pair every student with every food item.
* **Full Join**:
  + Produces M+N rows when the condition is always false:
    - M: Rows from the first table with NULL for the second table's columns.
    - N: Rows from the second table with NULL for the first table's columns.
  + Includes matching rows, if any, based on the condition.
  + Example: Combine unmatched rows from two tables into a single result.

#### **Key Differences in Row Count**

| **Feature** | **Cross Join** | **Full Join** |
| --- | --- | --- |
| **Rows Produced** | M×N | M+N (when no match). |
| **Condition** | No condition required. | Requires a condition. |

#### **Conclusion**

* **Cross Join** is used for creating combinations of all rows.
* **Full Join** is used for merging data while preserving all unmatched rows.
* **Efficiency**:
  + Cross join creates a large virtual table (M×N).
  + Full join creates a smaller virtual table (M+N) when rows do not match.

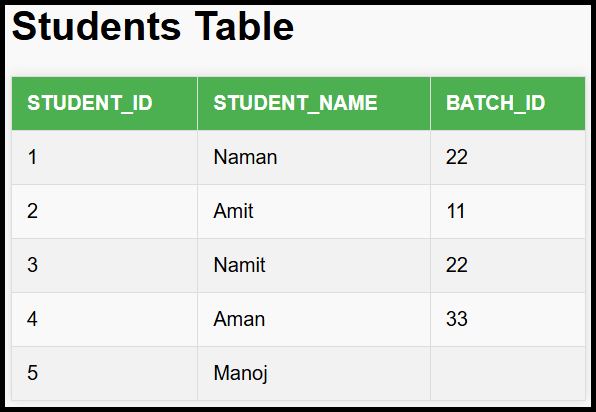
## **Implicit Join and its Drawbacks**

#### **What is an Implicit Join?**

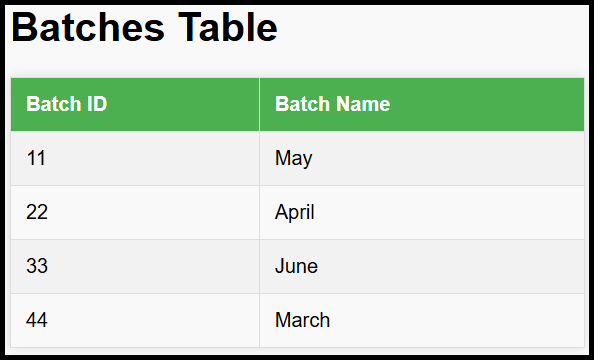
* **Definition**: An implicit join occurs when:
  + Tables are combined without explicitly specifying a JOIN type.
  + The query uses a comma-separated list of tables in the FROM clause.
  + A condition is applied in the WHERE clause to match rows.

#### **Example of an Implicit Join**

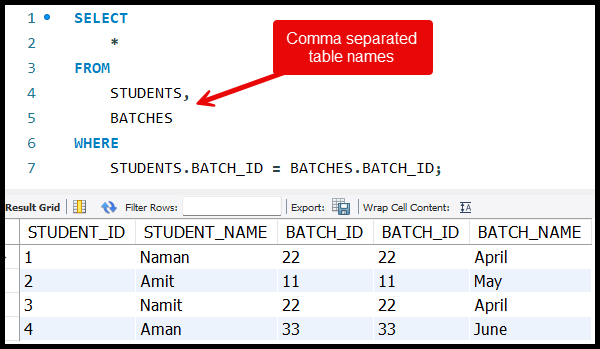
* **Scenario**: Combine students and batches tables to print every student's batch name.
  + STUDENTS **table**:



* + BATCHEStable:



* Query:



**Execution**:

* This query performs a **cross join** first, creating all possible combinations of rows from both tables.
* The WHERE clause filters out unmatched rows.

#### **Drawbacks of Implicit Joins**

1. **Inefficiency**:
   * Implicit joins involve an intermediate **cross join**:
     + All combinations of rows from both tables are generated.
     + Matched rows are selected after applying the WHERE condition.
   * For large tables, this operation is **extremely slow** and inefficient.
2. **Behind-the-Scenes Translation**:
   * The query SELECT \* FROM students, batches WHERE ... translates to:

SELECT \*

FROM STUDENTS

CROSS JOIN BATCHES

WHERE STUDENTS.BATCH\_ID = BATCHES.BATCH\_ID;

* + **Cross join** produces M×N rows, where:
    - M: Rows in the first table.
    - N: Rows in the second table.

1. **Not Suitable for Production**:
   * Small tables may not show noticeable performance issues.
   * For large datasets, implicit joins can cause:
     + High computational overhead.
     + Slow query execution in production environments.

#### **Best Practices**

* Use **explicit joins** for better performance and clarity:
  + **Inner Join**: When you want only the matching rows.
  + **Full Join**: When you want to include all rows from both tables.
  + **Example**:

SELECT \*

FROM STUDENTS

INNER JOIN BATCHES

ON STUDENTS.BATCH\_ID = BATCHES.BATCH\_ID;

* Explicit joins:
  + Avoid the unnecessary overhead of cross joins.
  + Are more readable and maintainable.