How to Perform a 3-Table Inner Join in MySQL

Authored by

mohammed looti

January 5, 2026

1. Introduction: Understanding Multi-Table Inner Joins

When working with complex relational databases, it is often necessary to combine information from multiple related tables into a single, cohesive result set. The Inner Join operation in MySQL is a fundamental tool for achieving this. Specifically, performing an Inner Join across three tables requires strategically chaining the joins together. This technique ensures that only those rows that possess matching values across all three designated tables will be included in the final result set, thereby guaranteeing accuracy and integrity in the retrieved data.

An Inner Join involving three tables is executed by using the JOIN clause to initially combine two tables, and then applying the same clause to connect the third table to the resultant dataset from the first two steps. This establishes a required relationship across all entities, where only the rows that satisfy all stipulated join conditions are ultimately returned. This process is essential for retrieving comprehensive and accurate data from multiple sources that are connected through common primary and foreign key fields.

The essence of a successful multi-table join lies in defining clear, meaningful relationships between successive tables. We start by joining the first two tables based on a common key, and subsequently, we join the third table to the resulting virtual table using another relevant key relationship. This cascading approach effectively links all entities, allowing us to generate comprehensive reports and analyze interconnected information that spans various parts of the schema.

2. The Mechanics of a Three-Table Join in MySQL

Executing an Inner Join involving three tables is achieved by iteratively applying the **INNER JOIN** keyword. This process builds upon itself, sequentially merging datasets. For instance, if Table A links to Table B via an **ID** column, and Table B links to Table C via a **team_id** column, the query must explicitly define both of these relationship paths sequentially, ensuring every record has matches in all steps.

The resulting relationship strictly ensures that the final output includes only the rows for which a match exists in the linking columns of **all three** tables. If a row in Table A matches a row in Table B, but the combined record does not have a corresponding match in Table C, that record will be excluded from the final result set. This strict matching criterion is why the Inner Join is often referred to as a restrictive join type, returning only the intersection of the datasets.

3. Essential SQL Syntax for Three-Table Joins

The standard syntax for performing a three-table Inner Join in MySQL involves listing the base

table in the **FROM** clause, followed by repeated use of the **INNER JOIN** and **ON** clauses for subsequent tables. It is crucial to specify the columns used for linking using the dot notation (`table_name.column_name`) to avoid ambiguity.

You can use the following syntax in MySQL to perform an inner join with 3 tables:

```
SELECT *
FROM athletes1
INNER JOIN athletes2
ON athletes1.id = athletes2.id
INNER JOIN athletes3
ON athletes2.team_id = athletes3.team_id;
```

This particular example performs an inner join based on matching values in the following columns:

The **id** column of **athletes1** and the **id** column of **athletes2**.
The **team_id** column of **athletes2** and the **team_id** column of **athletes3**.

The following practical example demonstrates how to use this syntax successfully.

4. Setting Up the Database Environment: Creating and Populating Athletes1

To provide a clear, practical demonstration of the three-table join, we must first establish and populate our sample tables. We will utilize three tables related to basketball player statistics. This setup mirrors a typical scenario in a normalized database architecture where related information is separated into distinct entities.

We begin by creating the `athletes1` table, which serves as our base entity. This table stores fundamental data about the players, including their unique **id**, playing position, and points scored. The inclusion of the data insertion commands ensures the table is ready for the subsequent join operation.

Suppose we create the following table named **athletes1** that contains information about various basketball players:

```
-- create table
CREATE TABLE athletes1 (
id INT NOT NULL,
position TEXT NOT NULL,
points INT NOT NULL
);
```

```
-- insert rows into table
INSERT INTO athletes1 VALUES (1, 'Guard', 13);
INSERT INTO athletes1 VALUES (2, 'Forward', 25);
INSERT INTO athletes1 VALUES (3, 'Center', 10);
INSERT INTO athletes1 VALUES (4, 'Guard', 28);
INSERT INTO athletes1 VALUES (5, 'Forward', 16);
INSERT INTO athletes1 VALUES (6, 'Center', 20);

-- view all rows in table
SELECT * FROM athletes1;
```

Output of athletes1:

```
+----+----------+--------+
| id | position | points |
+----+----------+--------+
| 1 | Guard | 13 |
| 2 | Forward | 25 |
| 3 | Center | 10 |
| 4 | Guard | 28 |
| 5 | Forward | 16 |
| 6 | Center | 20 |
+----+----------+--------+
```

5. Creating and Populating Intermediate Table Athletes2

The second table, `athletes2`, introduces the link to the team structure and tracks an additional metric, assists. Crucially, this table shares the **id** column with `athletes1`, which forms the first connection point. It also contains the **team_id** column, which will act as the link to our third table, thereby establishing the necessary linkage chain.

Then suppose we create another table named **athletes2** that contains more information about various basketball players:

```
-- create table
CREATE TABLE athletes2 (
id INT NOT NULL,
team_id INT NOT NULL,
assists INT NOT NULL
);
```

```
-- insert rows into table
INSERT INTO athletes2 VALUES (2, 011, 4);
INSERT INTO athletes2 VALUES (5, 012, 2);
INSERT INTO athletes2 VALUES (1, 013, 10);
INSERT INTO athletes2 VALUES (4, 014, 9);
INSERT INTO athletes2 VALUES (6, 015, 13);
INSERT INTO athletes2 VALUES (3, 016, 7);

-- view all rows in table
SELECT * FROM athletes2;
```

Output of athletes2:

```
+----+---------+---------+
| id | team_id | assists |
+----+---------+---------+
| 2 | 11 | 4 |
| 5 | 12 | 2 |
| 1 | 13 | 10 |
| 4 | 14 | 9 |
| 6 | 15 | 13 |
| 3 | 16 | 7 |
+----+---------+---------+
```

6. Establishing the Final Link: Table Athletes3 (Conference Data)

The final table, `athletes3`, holds information related to the teams themselves--specifically, their conference affiliation (**conf**). This table is linked exclusively via the **team_id** column shared with `athletes2`. This sequence ensures that we can trace the data path from a player's points, through their team assignment, all the way to their conference designation, satisfying the requirements for a chained three-table join.

Then suppose we create another table named **athletes3** that contains more information about various basketball players:

```
-- create table
CREATE TABLE athletes3 (
team_id INT NOT NULL,
conf TEXT NOT NULL
);
```

```
-- insert rows into table
INSERT INTO athletes3 VALUES (011, 'West');
INSERT INTO athletes3 VALUES (012, 'East');
INSERT INTO athletes3 VALUES (013, 'East');
INSERT INTO athletes3 VALUES (014, 'West');
INSERT INTO athletes3 VALUES (015, 'West');
INSERT INTO athletes3 VALUES (016, 'East');

-- view all rows in table
SELECT * FROM athletes3;
```

Output of athletes3:

```
+---------+------+
| team_id | conf |
+---------+------+
| 11 | West |
| 12 | East |
| 13 | East |
| 14 | West |
| 15 | West |
| 16 | East |
+---------+------+
```

7. Executing the Combined Query and Analyzing Results

Now that all three tables are established and correctly linked, we execute the comprehensive **INNER JOIN** query. For clarity and efficiency, we explicitly select specific fields from each table using dot notation, rather than using `SELECT *`. This query integrates athlete data (ID, points) from `athletes1`, team identification (team_id) from `athletes2`, and conference details (conf) from `athletes3`.

We can use the following syntax to retrieve the specified columns, demonstrating the successful combination of all three datasets:

```
SELECT athletes1.id, athletes1.points, athletes2.team_id, athletes3.conf
FROM athletes1
INNER JOIN athletes2
ON athletes1.id = athletes2.id
INNER JOIN athletes3
```

ON athletes2.team_id = athletes3.team_id;

Final Output:

```
+----+--------+---------+------+
| id | points | team_id | conf |
+----+--------+---------+------+
| 2 | 25 | 11 | West |
| 5 | 16 | 12 | East |
| 1 | 13 | 13 | East |
| 4 | 28 | 14 | West |
| 6 | 20 | 15 | West |
| 3 | 10 | 16 | East |
+----+--------+---------+------+
```

Notice that we're able to successfully perform an inner join using all three tables, retrieving a clean, four-column result set that precisely correlates player statistics with their respective conference affiliation.

8. Conclusion and Further Database Applications

The successful execution of this three-table join confirms the effectiveness of chained Inner Join operations in MySQL. Every row in the final result set represents a record that possessed matching key values across `athletes1`, `athletes2`, and `athletes3`. If any record lacked a corresponding entry in any of the subsequent tables, it would have been excluded, reinforcing the integrity of the data retrieved.

This method is fundamental for reporting and analytical tasks in complex database systems. By relying on sequential joins, developers can ensure that queries only return robust, complete data records, making the SQL query output highly reliable for business intelligence and application development. Mastery of the multi-table Inner Join is a core skill for any professional working with MySQL.

The following tutorials explain how to perform other common tasks in MySQL: