How to Insert Data into a MySQL Table from Another Table

Authored by

mohammed looti

January 6, 2026

Introduction to Efficient Data Migration in MySQL

One of the most common requirements in database management is the need to transfer large volumes of data efficiently from one table to another. When working within the MySQL relational database system, this task is handled by the powerful **INSERT INTO SELECT** statement. This structure allows developers and administrators to bypass slow, cumbersome data exporting and importing processes, offering a streamlined, purely SQL-based solution for data migration and consolidation. By integrating the results of a standard `SELECT` query directly into the insertion process, we can ensure high performance and reliability when moving records across schemas or within the same database. Understanding this combined statement is fundamental for optimizing database operations that involve duplicating, archiving, or merging datasets.

The core advantage of employing the **INSERT INTO SELECT** mechanism is its flexibility and control. This technique enables precise specification of both the target columns in the destination table and the source columns from which the data will be drawn. Crucially, the source and destination tables do not need to share identical column names, provided the data types and order align during the selection process. Furthermore, the inherent structure of the statement allows for immediate data transformation or filtering, which can be accomplished using standard SQL clauses, such as `JOIN` or `GROUP BY`, directly within the `SELECT` subquery. This capability elevates the statement beyond simple copying, making it a robust tool for sophisticated data manipulation tasks.

Key Benefits of Using the INSERT INTO SELECT Statement

The implementation of the INSERT INTO SELECT statement provides several critical advantages over alternative manual methods. Primarily, it offers significant savings in both time and human effort, especially when dealing with databases containing hundreds or thousands of records. Instead of requiring developers to extract data into temporary files, process them externally, and then reload them, the transfer is executed entirely within the database engine. This atomic operation dramatically reduces the potential for manual errors and improves execution speed, which is vital in high-traffic or time-sensitive environments.

A secondary, but equally important, benefit revolves around maintaining high levels of data consistency and integrity. Because the data is transferred directly from the source table to the destination table without intermediate external processes, the risk of data corruption, truncation, or transformation errors is minimized. The database system handles the transaction internally, often ensuring that the entire operation either succeeds completely or fails completely, depending on the transaction settings. This reliability is paramount in applications where data accuracy is non-negotiable, such as financial or critical infrastructure systems.

Analyzing the Syntax Structure

To perform a seamless insertion of rows based on a query result, you must follow a specific syntax pattern in MySQL. This syntax clearly delineates the target table, the columns intended to receive the new data, and the criteria used to select the data from the source table. Note that the number and data types of the columns listed in the `INSERT INTO` clause must exactly match the number and data types of the columns returned by the `SELECT` query.

The basic structure used to insert rows from one table into another is as follows:

```
INSERT INTO athletes1 (athleteID, team, points)
SELECT athleteID, team_name, total_points
FROM `athletes2`;
```

This illustrative example demonstrates how data is extracted from the source table named **athletes2**, specifically using the columns **athleteID**, **team_name**, and **total_points**. These selected values are then mapped and inserted sequentially into the corresponding columns--**athleteID**, **team**, and **points**--of the destination table, **athletes1**. Even though the names `team_name` and `team` differ, the insertion is successful because they occupy the correct corresponding positions and have compatible data types.

The subsequent sections will provide a practical, step-by-step illustration of how to implement this syntax effectively within a live database scenario, ensuring a clear understanding of the process from table creation to final data verification.

Practical Example: Defining the Destination Table Schema (athletes1)

For our demonstration, we will begin by establishing the schema for our destination table, **athletes1**. This table is designed to store basic profile and statistical information for basketball players. It is critical to define the column names and data types accurately, as these definitions dictate how the incoming data from the source table will be received and stored.

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

```
-- create table
CREATE TABLE athletes1 (
athleteID INT PRIMARY KEY,
team TEXT NOT NULL,
points INT NOT NULL
);
```

```
-- insert rows into table
INSERT INTO athletes1 VALUES (0001, 'Mavs', 22);
INSERT INTO athletes1 VALUES (0002, 'Warriors', 14);
INSERT INTO athletes1 VALUES (0003, 'Nuggets', 37);

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

Output of the initial dataset in athletes1:

```
+-----------+----------+--------+
| athleteID | team | points |
+-----------+----------+--------+
| 1 | Mavs | 22 |
| 2 | Warriors | 14 |
| 3 | Nuggets | 37 |
+-----------+----------+--------+
```

The **athletes1** table is initialized with three existing records. Notice the definition of `athleteID` as the **PRIMARY KEY**, which enforces uniqueness. The `team` column is defined as **TEXT** and `points` as **INT**, both having a **NOT NULL** constraint. When we insert data using **INSERT INTO SELECT**, the source data must satisfy these constraints, or the operation will fail.

Defining the Source Table Schema (athletes2)

Next, we must prepare the source table, **athletes2**, which holds additional player data that we intend to append to **athletes1**. This setup highlights a common real-world scenario where data intended for a single destination might originate from multiple tables with slightly different schema designs. Specifically, the column names here (`team_name` and `total_points`) differ from the destination table, illustrating the necessary mapping requirement in the `SELECT` clause.

We create another table named **athletes2** that contains information about more basketball players:

```
-- create table
CREATE TABLE athletes2 (
athleteID INT PRIMARY KEY,
team_name TEXT NOT NULL,
total_points INT NOT NULL
);
```

```
-- insert rows into table
INSERT INTO athletes2 VALUES (0004, 'Lakers', 19);
INSERT INTO athletes2 VALUES (0005, 'Celtics', 26);
INSERT INTO athletes2 VALUES (0006, 'Thunder', 40);

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

Output of the initial dataset in athletes2:

```
+-----------+-----------+-------------+
| athleteID | team_name | total_points |
+-----------+-----------+-------------+
| 4 | Lakers | 19 |
| 5 | Celtics | 26 |
| 6 | Thunder | 40 |
+-----------+-----------+-------------+
```

The **athletes2** table contains three new records, starting with ID 4. Since `athleteID` is the primary key in **athletes1**, we must ensure that the IDs being inserted (4, 5, 6) do not clash with the existing IDs (1, 2, 3), thus avoiding a primary key violation error during the insertion process. The next step is to execute the full transfer command, mapping `athleteID` to `athleteID`, `team_name` to `team`, and `total_points` to `points`.

Executing a Full Data Merge Using INSERT INTO SELECT

Our objective now is to seamlessly insert all rows present in **athletes2** into **athletes1**, effectively merging the two datasets into the destination table. This requires us to explicitly name the columns in the `INSERT INTO` part and list the corresponding, matched columns in the `SELECT` part.

We utilize the following syntax to perform the complete merge operation:

```
-- insert rows from athletes2 into athletes1
INSERT INTO athletes1 (athleteID, team, points)
SELECT athleteID, team_name, total_points
FROM `athletes2`;

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

+-----------+----------+--------+
```

```
| athleteID | team | points |
+-----------+----------+--------+
| 1 | Mavs | 22 |
| 2 | Warriors | 14 |
| 3 | Nuggets | 37 |
| 4 | Lakers | 19 |
| 5 | Celtics | 26 |
| 6 | Thunder | 40 |
+-----------+----------+--------+
```

As clearly demonstrated by the final output, all three rows originating from **athletes2** have been successfully appended to the existing records in **athletes1**. The resulting table now contains six distinct athlete profiles, confirming the successful execution of the full data migration using the **INSERT INTO SELECT** method.

Implementing Conditional Insertion using the WHERE Clause

A significant benefit of using the **INSERT INTO SELECT** structure is the ability to filter the source data before it is inserted into the destination table. By incorporating the WHERE clause into the `SELECT` query, we can enforce specific business rules, ensuring that only records meeting certain criteria are migrated. This is essential for tasks like archiving old data, migrating high-priority records, or performing data cleanup during a transfer.

In this example, we will assume **athletes1** has been reset to its initial state (containing only IDs 1, 2, and 3). We now use the following syntax to insert only those rows from **athletes2** where the athlete's total score is greater than 20:

```
-- insert rows from athletes2 into athletes1
INSERT INTO athletes1 (athleteID, team, points)
SELECT athleteID, team_name, total_points
FROM `athletes2`
WHERE total_points > 20;

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

```
+-----------+----------+--------+
| athleteID | team | points |
+-----------+----------+--------+
| 1 | Mavs | 22 |
```

2	Warriors	14
3	Nuggets	37
5	Celtics	26
6	Thunder	40
+----------+----------+--------+

Upon reviewing the final output of **athletes1**, we observe that only two rows (IDs 5 and 6) were successfully inserted from **athletes2**. The record corresponding to ID 4 (Lakers, 19 points) was filtered out because its `total_points` value (19) did not satisfy the condition defined by the `WHERE total_points > 20` clause. This conditional filtering demonstrates how powerful the **INSERT INTO SELECT** statement is for targeted data population.

Further Exploration of MySQL Data Manipulation Techniques

Mastering the **INSERT INTO SELECT** technique is essential for effective database management in MySQL. This capability allows for complex data workflows, including data auditing, reporting dataset generation, and streamlined consolidation of fragmented information. However, database proficiency extends to many other techniques crucial for daily operation.

For those looking to deepen their understanding of MySQL and related data operations, several other common tasks and tutorials are highly recommended to expand your skill set beyond basic insertion and selection capabilities.

Understanding various SQL JOIN types (INNER, LEFT, RIGHT) for complex query building.

Techniques for updating records in one table based on data in another using `UPDATE JOIN` statements.

Strategies for handling duplicates when merging large datasets.

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