How to Delete Duplicate MySQL Rows and Keep the Most Recent Entry

Authored by

mohammed looti

January 6, 2026

The need to maintain data integrity often requires database administrators and developers to manage and eliminate redundant entries. A common scenario in MySQL involves identifying rows that are duplicates based on one or more columns (like a user name or team name), but ensuring that the record associated with the most recent activity--typically marked by the highest PRIMARY KEY value or a dedicated timestamp--is preserved.

While simpler methods involving functions like **DISTINCT** and **MAX** can identify the latest entries, they are often insufficient for performing a large-scale deletion operation directly. The most robust and widely accepted method for deleting older duplicate rows in MySQL, while retaining the single latest version, involves employing a specialized technique known as a **self-join** combined with the DELETE command. This approach allows the database engine to compare rows within the same table, selectively marking the older records for removal based on a chronological identifier such as an auto-incrementing **ID** column.

This article provides a comprehensive guide to executing this operation efficiently, focusing on clear syntax, practical examples, and the underlying logic of the **self-join** strategy. By utilizing this powerful SQL technique, you can clean your dataset and ensure that only the most relevant and timely records remain active in your tables.

Understanding the MySQL Self-Join Deletion Strategy

To successfully delete duplicates while prioritizing the latest entry, we must first define what constitutes a duplicate and what determines recency. A duplicate is identified when two or more rows share identical values in the columns we wish to constrain (e.g., the **team** column). Recency is determined by a separate, unique identifier, typically the row's **id**, where a higher ID value indicates a more recently created record.

The self-join technique is essential here because the standard DELETE statement is limited in its ability to reference other rows for comparison within the same operation. By joining the table to itself--using two different aliases, say `t1` and `t2`--we effectively create two copies of the data, allowing us to compare them side-by-side. Alias `t1` represents the row targeted for deletion, and `t2` represents the row against which `t1` is being compared.

The core logic hinges on the **WHERE** clause: we look for instances where `t1.team = t2.team` (identifying duplicates) AND `t1.id < t2.id` (identifying that `t1` is older than its duplicate, `t2`). When both conditions are met, the self-join identifies `t1` as a row that is both redundant and chronologically older than at least one other matching row. The command then targets and removes only the rows aliased as `t1`, ensuring that the row aliased as `t2`, which possesses the greater **id**, is preserved.

The Syntax Explained: Deleting Older Duplicates

The standard syntax in MySQL for deleting all duplicate rows in a table while retaining the one with the latest **ID** value utilizes this powerful self-join mechanism. The flexibility of MySQL allows us to specify which alias is the target of the deletion directly after the **DELETE** keyword, simplifying the operation significantly compared to some other SQL dialects.

The following structure is highly effective for this specific task. Notice how we explicitly reference the table aliases immediately after the **DELETE** and **FROM** keywords, telling MySQL exactly which set of records to remove based on the subsequent comparison logic.

DELETE t1 FROM athletes t1, athletes t2
WHERE t1.id < t2.id AND t1.team = t2.team;

In this example, which targets the hypothetical **athletes** table, we are instructing the database to remove records from the `t1` set. The condition `t1.team = t2.team` ensures that we only consider rows where the **team** name is identical, confirming a duplication exists. Crucially, the condition `t1.id < t2.id` dictates that only the record with the **lower id** (the older entry) is marked for deletion. If a row in `t1` has a duplicate in `t2`, and `t1`'s ID is smaller, it means `t2` is the latest record, and `t1` must be removed. This ensures robust deduplication, prioritizing recency as defined by the ID sequence.

Practical Demonstration: Setting Up the Sample Data

To illustrate this process, we will work with a sample table named **athletes**. This table simulates common database scenarios where records might be updated or re-inserted, leading to duplicate entries based on the primary classification (in this case, the **team** name). The table structure includes an **id** column, which serves as our reliable indicator of insertion order and recency, a **team** column, and a **points** column.

We begin by defining the table structure and populating it with a mix of unique and duplicate entries. Notice that teams like 'Mavs', 'Lakers', 'Knicks', and 'Celtics' have multiple entries, each distinguished by a unique **id**. For instance, the 'Knicks' team appears three times with IDs 4, 5, and 6, where 6 represents the latest record for that team.

The following SQL code block sets up the environment and shows the initial state of the dataset before any cleanup operations are performed. This setup is critical for verifying the success of the subsequent DELETE command.

-- create table
CREATE TABLE athletes (

```
id INT PRIMARY KEY,
team TEXT NOT NULL,
points INT NOT NULL
);

-- insert rows into table
INSERT INTO athletes VALUES (0001, 'Mavs', 22);
INSERT INTO athletes VALUES (0002, 'Mavs', 14);
INSERT INTO athletes VALUES (0003, 'Lakers', 37);
INSERT INTO athletes VALUES (0004, 'Knicks', 19);
INSERT INTO athletes VALUES (0005, 'Knicks', 26);
INSERT INTO athletes VALUES (0006, 'Knicks', 40);
INSERT INTO athletes VALUES (0007, 'Lakers', 21);
INSERT INTO athletes VALUES (0008, 'Celtics', 15);
INSERT INTO athletes VALUES (0009, 'Hawks', 18);
INSERT INTO athletes VALUES (0010, 'Celtics', 15);

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

The resulting output clearly shows the ten initial rows, highlighting the redundant team entries. The objective is now to eliminate the older entries (those with lower IDs) for each duplicate team, ensuring that the information associated with the highest ID (the latest update) for that team is the only one that persists in the table.

Output (Initial Dataset):

```
+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 1 | Mavs | 22 |
| 2 | Mavs | 14 |
| 3 | Lakers | 37 |
| 4 | Knicks | 19 |
| 5 | Knicks | 26 |
| 6 | Knicks | 40 |
| 7 | Lakers | 21 |
| 8 | Celtics | 15 |
| 9 | Hawks | 18 |
| 10 | Celtics | 15 |
```

+----+---------+--------+

Executing the Deletion Command and Reviewing Results

We are now ready to apply the self-join deletion syntax. This specific query leverages the table aliases `t1` and `t2` to identify pairs of rows where the **team** names match, and then performs the deletion on the entry with the strictly smaller **id**.

The command efficiently filters out the older records, leaving a clean dataset where each team appears exactly once, associated with the latest **id** value it attained. This operation is non-reversible, so it is always recommended to perform a backup or test this query within a transaction block before executing it on production data.

```
DELETE t1 FROM athletes t1, athletes t2
WHERE t1.id < t2.id AND t1.team = t2.team;
```

Upon execution, the MySQL server processes the join, identifies the older duplicate rows based on the `t1.id < t2.id` condition, and removes them. When we re-query the **athletes** table, we observe a significantly reduced and cleaned dataset.

Output (After Deleting Older Duplicates):

```
+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 2 | Mavs | 14 |
| 6 | Knicks | 40 |
| 7 | Lakers | 21 |
| 9 | Hawks | 18 |
| 10 | Celtics | 15 |
+----+---------+--------+
```

The result confirms that all rows with duplicate values in the **team** column have been deleted, and only the one corresponding to the latest value in the **id** column was successfully kept. For example, the 'Mavs' team originally had records with IDs 1 and 2; only ID 2, the latest, remains. Similarly, the 'Knicks' team, which possessed IDs 4, 5, and 6, has been reduced to only the record associated with ID 6. This demonstrates the precision and effectiveness of the self-join comparison in prioritizing recency.

Handling Alternative Requirements: Keeping the Earliest Entry

In certain data cleaning scenarios, the requirement might be the reverse: you need to delete duplicate rows but specifically keep the one that was inserted **first** (i.e., the row with the earliest, or lowest, **id** value). This might be necessary if the oldest entry represents a foundational record that should not be overwritten by later, potentially incomplete or flawed, updates.

Fortunately, the structure of the MySQL self-join deletion allows for this modification with minimal effort. Instead of using the less-than symbol (<) in the comparison clause, which targets the older row for deletion, we simply swap it for the greater-than symbol (>). This simple change completely reverses the deletion logic.

By specifying `t1.id > t2.id`, we are now targeting `t1` (the row to be deleted) only when its ID is **higher** than its duplicate counterpart `t2`. If a row in `t1` has a duplicate in `t2`, and `t1`'s ID is larger, it means `t1` is the later record, and it is subsequently deleted. This preserves the row with the lowest ID, which is the earliest entry.

```
DELETE t1 FROM athletes t1, athletes t2
WHERE t1.id > t2.id AND t1.team = t2.team;
```

If we were to re-run this revised query on the original dataset (assuming we rolled back the previous deletion), the resulting output would preserve only the initial record for each team.

Output (After Deleting Latest Duplicates, Keeping Earliest):

```
+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 1 | Mavs | 22 |
| 3 | Lakers | 37 |
| 4 | Knicks | 19 |
| 8 | Celtics | 15 |
| 9 | Hawks | 18 |
+----+---------+--------+
```

As demonstrated, all rows with duplicate values in the **team** column have been deleted, and only the ones with the earliest value in the **id** column were kept. For instance, for the 'Knicks' team, the record with ID 4 (the earliest entry) is now the sole survivor, confirming the successful reversal of the deletion criteria.

Considerations for Performance and Large Datasets

While the self-join method is highly effective and flexible, database administrators must consider its performance implications, especially when dealing with extremely large tables (millions or billions of rows). A self-join requires the database engine to compare potentially every row against every other row, which can be resource-intensive.

To optimize the performance of this deletion query in MySQL, it is critical to ensure proper indexing. The table must have indexes on the columns used in the join and the comparison clauses. Specifically, an index should be placed on the column identifying the duplicate group (e.g., **team**) and, ideally, a composite index covering both the duplicate column and the ordering column (e.g., `(team, id)`). This indexing allows MySQL to quickly find matching pairs without resorting to full table scans.

Furthermore, when executing large-scale DELETE operations, it is advisable to manage the workload in controlled batches if the table is exceptionally large, preventing long-running transactions that could block other database operations. Using a transaction block (`START TRANSACTION` and `COMMIT`) is also a crucial safety measure, allowing you to review the operation's impact before permanently applying the changes.

Summary of Best Practices

The self-join technique provides a powerful, standard SQL solution for advanced data deduplication within a single table in **MySQL**. Whether you need to preserve the latest record or the earliest, the core logic remains the same: use two aliases of the table and define the relationship between their primary keys based on whether you want to delete the older entry (<) or the newer entry (>).

Always ensure that the column used to determine "latest" or "earliest" is reliable and accurately reflects the order of insertion or modification. While an auto-incrementing integer **id** column is usually sufficient, using a dedicated timestamp column (like `created_at` or `updated_at`) often provides a more accurate chronological ordering, especially if records might be inserted out of sequence for any reason. If a timestamp is used, ensure the index is present on that column and the comparison is adjusted accordingly.

By understanding the nuances of the self-join and properly indexing your data, you can efficiently and safely maintain high data quality by eliminating unwanted duplicate rows while preserving your required dataset integrity.

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

MySQL: How to Use DELETE with INNER JOIN
MySQL: How to Delete Rows from Table Based on id