How to Delete Rows from a MySQL Table by ID

Authored by

mohammed looti

January 6, 2026

The process of removing specific data records from a database is a fundamental operation in SQL. To delete rows from a MySQL table using an identifier column, such as an **id**, as the basis for selection, developers utilize the essential `DELETE` command combined with the restrictive `WHERE clause`. This combination allows you to precisely target and remove specific records that match the specified criteria, effectively purging them from the table.

It is paramount to exercise extreme caution whenever executing the `DELETE` command in a production or live environment. Unlike many data modification operations that might be transactional and reversible, data removal using `DELETE` is typically permanent. Without a recent database **backup** or the implementation of explicit transactions that can be rolled back, the data is irreversibly lost. Therefore, before committing to deletion, it is considered best practice to first use the `SELECT` command with the identical `WHERE clause` to verify that the rows being targeted are indeed the correct ones, mitigating the risk of accidental data loss.

Fundamental Methods for Deleting Rows by ID

When working within the MySQL environment, the `DELETE` statement offers several flexible ways to specify which rows should be removed based on their **ID** values. These methods rely heavily on different operators within the mandatory WHERE clause, allowing for single deletions, range deletions, list deletions, or conditional deletions based on inequality comparisons. Understanding these variations is crucial for efficient database maintenance and administration.

Below, we outline four standard operational methods used by database professionals to achieve targeted row deletion. Each method serves a specific use case, from removing a single known record to clearing out large sequential blocks of records. The syntax is highly consistent, always starting with `DELETE FROM` , followed by the specific filtering condition defined after the `WHERE` keyword.

Method 1: Deleting a Single Row Based on Exact ID Match

This is the most straightforward and frequently used method when the precise identifier of the row to be removed is known. By utilizing the equality operator (**=**) within the `WHERE clause`, the `DELETE` statement strictly targets only the single record whose primary key or identifier column exactly matches the specified numeric value. This method guarantees high precision and minimal risk, provided the ID is correctly identified.

DELETE FROM mytable WHERE id=3;

The command above instructs the database engine to scan the entire `mytable` and permanently

remove any row where the value stored in the **id** column is exactly 3. This operation is fast and efficient when the **id** column is indexed, which is generally the case for primary key columns.

Method 2: Removing Rows Within a Defined Range Using BETWEEN

When there is a need to delete a contiguous sequence of rows, perhaps related to a specific time period or log index, the `BETWEEN` operator provides an elegant solution. The `BETWEEN` clause is inclusive, meaning it targets rows whose **id** values fall between the starting and ending boundaries specified, including both the minimum and maximum values. This is particularly useful for batch cleanup operations.

DELETE FROM mytable WHERE id BETWEEN 1 AND 3;

This statement executes a DELETE operation on `mytable`, affecting all rows where the **id** is 1, 2, or 3. It is a highly efficient way to prune sequential data without requiring multiple individual `DELETE` statements or complex looping logic.

Method 3: Deleting Dispersed Rows Using the IN Operator

In scenarios where the IDs to be deleted are not sequential but are known and listed, the `IN` operator is the ideal choice. The `IN` operator checks if the value in the specified column (**id**, in this case) matches any value within the provided comma-separated list contained in the parentheses. This is significantly cleaner and faster than chaining multiple `OR` conditions.

DELETE FROM mytable WHERE id IN (1, 4, 5);

The example above targets three distinct and potentially unrelated records (IDs 1, 4, and 5) and removes them from `mytable` in a single, atomic operation. The number of IDs provided in the list can be quite large, though performance considerations should be taken into account for extremely long lists, where temporary tables or alternative strategies might sometimes be more beneficial.

Method 4: Conditional Deletion Using Inequality Operators (Greater Than / Less Than)

For clearing out old records based on a threshold--a common requirement in data archiving or logging systems--inequality operators such as greater than (**>**) or less than (**<**) are used. This allows for the deletion of all records that are older than or newer than a specific ID reference point. This is often used when ID sequences represent the order of creation, assuming higher IDs are newer records.

DELETE FROM mytable WHERE id > 4;

Executing this command will delete all rows from `mytable` where the **id** value is strictly greater than 4. If the goal is to include the row with ID 4, the greater than or equal to operator (**>=**) must be used instead. This conditional filtering mechanism is powerful for bulk operations spanning indeterminate numbers of rows.

Setting Up the Demonstration Table: The Athletes Dataset

To provide clear, practical illustrations of these four deletion methods, we will utilize a sample table named `athletes`. This table simulates a typical dataset structure, containing player identification, team association, and performance statistics. Before executing any `DELETE` commands, the table must first be created and populated with sample data. This setup ensures that we have a consistent starting point for all subsequent examples.

The `athletes` table is structured with three columns: `athleteID`, which serves as the Primary Key; `team`, which is a text field; and `points`, an integer field. The use of a designated Primary Key column (`athleteID`) is crucial as it guarantees that each row has a unique identifier, making targeted deletion highly reliable.

The following SQL script creates the table and inserts six initial rows of basketball player data:

```
-- create table
CREATE TABLE athletes (
athleteID 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, 'Celtics', 14);
INSERT INTO athletes VALUES (0003, 'Nuggets', 37);
INSERT INTO athletes VALUES (0004, 'Knicks', 19);
INSERT INTO athletes VALUES (0005, 'Warriors', 26);
INSERT INTO athletes VALUES (0006, 'Thunder', 40);

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

After successful execution of the setup script, the table content will appear as follows, serving as the baseline for all subsequent deletion demonstrations. Note that for simplicity in the output display, the `athleteID` column is referenced as `id` in the result set headers.

Initial Dataset Output (`athletes` table)

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

Now that the dataset is established, we can proceed with demonstrating each of the four identified methods for row deletion based on the **id** column.

Practical Example 1: Deleting a Single Specific Row

This demonstration illustrates the precision achieved by targeting a single record using the equality operator. We aim to permanently remove the record corresponding to athlete ID 3 (the Nuggets player) from the `athletes` table. This is achieved by ensuring the DELETE command is coupled with the exact `WHERE id = 3` condition, preventing any unintended deletions.

It is important to remember that this operation is immediate and permanent in the absence of explicit transaction handling. Before running this command in a real application, an equivalent `SELECT * FROM athletes WHERE id=3;` query should be run to confirm the target data.

DELETE FROM athletes WHERE id=3;

Upon successful execution, the record associated with ID 3 is removed. The resulting table demonstrates that only five rows remain, confirming the targeted deletion. The database engine handles the necessary index and storage updates efficiently.

+----+----------+--------+
| id | team | points |
+----+----------+--------+

| 1 | Mavs | 22 |
| 2 | Celtics | 14 |
| 4 | Knicks | 19 |
| 5 | Warriors | 26 |
| 6 | Thunder | 40 |
+----+----------+--------+

Practical Example 2: Batch Deletion Using ID Ranges (BETWEEN)

If we needed to clear out the initial set of records, perhaps because they were test data or belonged to a previous archival cycle, the `BETWEEN` operator is highly effective. In this example, we will roll back our dataset and demonstrate how to delete all rows corresponding to **id** values 1, 2, and 3 simultaneously. This approach saves query time compared to running three separate `DELETE` statements.

The `BETWEEN` operator is inclusive, meaning both endpoints (1 and 3) are included in the set of records to be deleted. Developers must always be sure that the range chosen precisely encapsulates the intended targets, as over-deletion is a common operational hazard.

DELETE FROM athletes WHERE id BETWEEN 1 AND 3;

The resulting table snapshot confirms that the range of athletes with IDs 1 through 3 has been successfully purged, leaving only the higher ID records (4, 5, and 6) in the dataset. This illustrates the utility of range-based deletion for handling sequential data removal efficiently.

Output After Range Deletion:

+----+----------+--------+
| id | team | points |
+----+----------+--------+
| 4 | Knicks | 19 |
| 5 | Warriors | 26 |
| 6 | Thunder | 40 |
+----+----------+--------+

Practical Example 3: Targeting Non-Sequential Rows Using IN

This scenario addresses the need to remove specific records that are not numerically adjacent. Suppose records 1, 4, and 5 have been flagged for removal due to data quality issues, regardless

of their position in the sequence. Using the `IN` operator allows us to specify this non-contiguous list within a single, optimized `DELETE` statement.

The power of the `IN` clause lies in its ability to handle multiple discrete targets efficiently, relying on the database index to quickly locate and remove each specified ID. The list within the parentheses must contain the exact values of the **id** column that are targeted for removal.

DELETE FROM athletes WHERE id IN (1, 4, 5);

Following this execution, the table retains only the athletes whose IDs were not present in the list (IDs 2, 3, and 6). This method is indispensable when dealing with dynamically generated lists of records, such as those returned from a subquery or application logic.

Output After List Deletion:

+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 2 | Celtics | 14 |
| 3 | Nuggets | 37 |
| 6 | Thunder | 40 |
+----+---------+--------+

Practical Example 4: Deleting Records Based on ID Thresholds

Often, data retention policies require the deletion of records created after a certain processing point, or conversely, the removal of all older records. This is achieved using strict inequality operators. Here, we demonstrate how to delete all records that have an **id** value strictly greater than 4. This effectively removes IDs 5 and 6 (Warriors and Thunder).

It is vital to distinguish between the strict inequality operator (**>** or **<**) and the inclusive inequality operator (**>=** or **<=**). Using `id > 4` excludes ID 4 itself, whereas using `id >= 4` would include ID 4 in the deletion set. Careful selection of the operator is crucial to avoid boundary errors in data management.

DELETE FROM athletes WHERE id > 4;

As anticipated, the resulting table displays only the athletes with IDs 1 through 4, confirming that all records with IDs 5 and greater were successfully eliminated. This technique offers great flexibility for automated maintenance scripts.

Output After Threshold Deletion:

```
+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 1 | Mavs | 22 |
| 2 | Celtics | 14 |
| 3 | Nuggets | 37 |
| 4 | Knicks | 19 |
+----+---------+--------+
```

Critical Safety Note on Inequality Operators

When performing conditional deletions based on ID thresholds, remember the distinction between strict and inclusive operators. Using **>=** (greater than or equal to) instead of **>** (greater than) will alter the boundary of the deletion. For instance, if you intend to delete all data from ID 4 onwards, you must use `WHERE id >= 4`. Misuse of these operators is a common source of errors resulting in unintended data deletion or retention.

Always verify the scope of your `WHERE` clause using a `SELECT` statement before executing the final DELETE command, especially when dealing with operators that target large subsets of the data, such as >, <, >=, or <=. This simple verification step dramatically enhances the safety and reliability of your database maintenance routines.

The following tutorials explain how to perform other common tasks in MySQL, such as using advanced join logic for deletion:

MySQL: How to Use DELETE with INNER JOIN