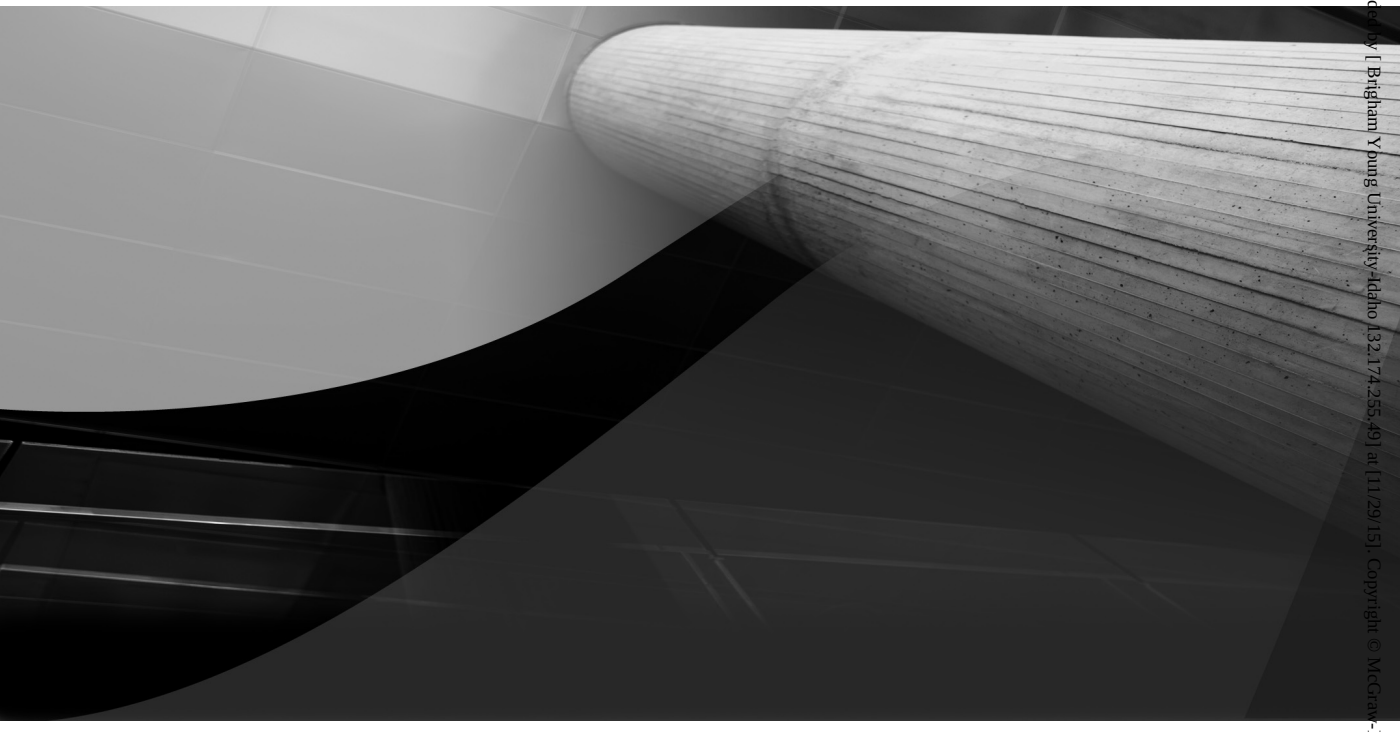


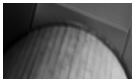
CHAPTER 10

Deleting Data

he DELETE FROM statement lets you remove data from tables and views. There are two types of DELETE FROM statements: One uses literal values or subquery comparisons in a WHERE clause; the other uses correlated results from a subquery. It is also possible to combine values or subquery results with correlated results in the WHERE clause. MySQL offers a variation on correlated statements with a multiple-table DELETE FROM statement. This chapter covers how you use both statements.

Like the UPDATE statement, a DELETE FROM statement removes all rows found in a table unless you filter what you want to remove in a WHERE clause. A DELETE FROM statement also writes redo logs, similar to INSERT and UPDATE statements, and supports bulk processing options inside Oracle's PL/SQL blocks.

The DELETE FROM statement has a closely related cousin, the DDL TRUNCATE statement. DBAs often disable constraints, copy a table's contents to a temporary table, TRUNCATE or DROP the table, re-create the table with a new storage clause, and then INSERT the old records from the temporary table. The TRUNCATE statement doesn't log deletions; it simply removes the allocated storage space in a tablespace, which makes it much faster for routine DBA maintenance tasks.



NOTE
Chapter 6 covers how to use the CREATE TABLE statement to clone a table.

The DELETE FROM statement also works on nested tables in the Oracle database. Deleting nested tables without removing the row from the table is the exception rather than the rule for a DELETE FROM statement, as you'll see in the next section.

The following two sections demonstrate the syntax for deleting rows by value matches and by correlation between two or more tables. Value matches can be literal values or ordinary subqueries. Correlation between two or more tables requires joins between tables, and MySQL includes a multiple-table DELETE FROM statement that illustrates it.

Delete by Value Matches

This DELETE FROM statement uses a table name and a WHERE clause that allows you to filter which rows you want to remove from a table. The WHERE clause works with date, numeric, and string literal values.

Here's the basic prototype for a DELETE FROM statement:

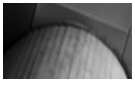
```
DELETE FROM table_name
WHERE [NOT] column_name {{= | <> | > | >= | < | <=} |
                         [NOT] {{IN | EXISTS} | IS NULL}} 'expression'
[{AND | OR } [NOT] comparison_operation] [...];
```

An actual DELETE FROM statement would look like this:

```
SQL> DELETE FROM item
  2  WHERE  item_title = 'Pirates of the Caribbean: On Stranger Tides'
  3  AND    item_rating = 'PG-13';
```

The first line sets the target table for the deletion operation. Lines 2 and 3 filter the rows to find those that will be deleted. All rows meeting those two criteria are removed from the table and immediately become invisible to the current user in transaction mode. As explained in Chapter 4, in the two-phase commit (2PC) model, the first phase removes rows from the current user's view, and the second phase removes them from the system. Between the first and second phases, other users see the deleted rows and can make decisions based on their existence—unless you've locked them in the context of a transaction.

You should lock rows that are possibly subject to deletion before running `DELETE FROM` statements. This is straightforward in a transactional database, such as Oracle. In a MySQL database, locks require more precautions because they exist only when you're working within the scope of a transaction using InnoDB-managed tables. You can use SQL cursors to lock rows when deletions run inside PL/SQL or SQL/PSM stored program units. You lock the rows in a SQL cursor by appending a `FOR UPDATE` clause. Regardless of your method of operation, failure to lock rows before deleting them can lead to insertion, update, or deletion anomalies. The anomalies can occur because other DML statements can make decisions on the unaltered rows, which are visible to other sessions before a `COMMIT` statement.



TIP
MySQL can perform this type of locking only when tables are defined using the InnoDB engine and in the scope of an explicit transaction scope.

In addition to using literal values in the `WHERE` clause, you can use ordinary subqueries. Ordinary subqueries act independently of the parent DML statement and return a `SELECT` list for comparision against values in the `DELETE FROM` statement.. These subqueries do have a restriction: they can return only a single row when you use an equality comparison operator, such as the equal (=) operator. You can also use multiple-row subqueries, but they require a lookup operator. As mentioned in Chapter 6, four lookup operators can be used: `IN`, `=ANY`, `=SOME`, and `=ALL`. The `IN`, `=ANY`, and `=SOME` operators behave similarly. They allow you to compare a column value in a row against a list of column values, and they return true if one value matches—this is like an `OR` logical operator in a procedural `IF` statement. The `=ALL` also allows you to compare a column value in a row against a list of column values, and it returns true when all values in the list match the single column value. The `=ALL` performs like an `AND` logical operator in a procedural `IF` statement. For reference, there is no standard *exclusive OR* operator in SQL.

Inside the `WHERE` clause, you can use `AND` or `OR` logical operators. The order of precedence requires that a group of logical comparisons connected by the `AND` logical operator are processed as a block before anything connected later by an `OR` logical operator.

Modifying the preceding example, let's add an `OR` logical comparison based on the release date. The following statement uses the default order of operation in the `WHERE` clause:

```
SQL> DELETE FROM item
  2  WHERE  item_title = 'Pirates of the Caribbean: On Stranger Tides'
  3  AND    item_rating = 'PG-13'
  4  OR     TRUNC(item_release_date) < TRUNC(SYSDATE,'YY');
```

This removes all rows where the literal values match the `item_title` and `item_rating` or all rows where the `item_release_date` precedes the first day of the year. You must use parentheses to change the order of operation.

Transaction Management

The basis for a transaction doesn't require specialized steps in an Oracle database, because Oracle statements are natively transactional and use a 2PC process to insert, update, or delete rows from tables. MySQL isn't transactional natively, so you must start and end a transaction scope.

The simplest example in MySQL is an insert between a `parent` and `child` table, where the `parent` table holds the primary key column and the `child` table holds a copy in a foreign key column. Here's the example:

```
START TRANSACTION;
INSERT INTO parent VALUES (NULL, 'One');
INSERT INTO child VALUES (NULL, last_insert_id(),'Two');
COMMIT;
```

The `START TRANSACTION;` statement starts a transaction scope. The first column in the `VALUES` clause for both tables is a null, which lets the automatic sequence provide the number. You can enter descriptive text in the second column in the `parent` table and third column in the `child` table. The second column in the `child` table holds a foreign key, which is a copy of a primary key column from the `parent` table. The `last_insert_id()` function in MySQL captures the sequence value used in the preceding `INSERT` statement to the `parent` table—it's similar to the `.CURRVAL` in an Oracle database. The last element is the `COMMIT` statement, which ends the transaction scope.

An alternative to the `START TRANSACTION;` statement is the `BEGIN WORK;` statement. It works the same way, as you can see:

```
BEGIN WORK;
INSERT INTO parent VALUES (NULL, 'One');
INSERT INTO child VALUES (NULL, last_insert_id(),'Two');
COMMIT;
```

In the scope of the transaction, no other session can see the inserted values until the `COMMIT` statement ends the transaction and makes changes permanent. You could substitute `DELETE FROM` statements for the `INSERT` statements, and no one would be able to see the deleted row until you committed the changes.

Let's say the business rule changes and now requires that the `item_title` matches the literal value, and either the `item_rating` matches the literal value or the `item_release_date` is less than the first day of the current year. A modified statement would look like this:

```
SQL> DELETE FROM item
  2  WHERE  item_title = 'Pirates of the Caribbean: On Stranger Tides'
  3  AND   (item_rating = 'PG-13'
  4  OR     TRUNC(item_release_date) < TRUNC(SYSDATE,'YY'));
```

The parentheses on lines 3 and 4 change the order of operation and remove only rows with matching `item_title` values and matches in other criteria.

Alternatives to values in the WHERE clause can be subqueries that return one or more column values. You can write a DELETE FROM statement when the query returns only one row, like this:

```
SQL> DELETE FROM item
  2  WHERE (item_title,item_rating) =
  3          (SELECT 'Pirates of the Caribbean: On Stranger Tides'
  4           ,       'PG-13'
  5           FROM    dual);
```

Line 2 contains an equal (=) comparison operator that works only when a single row is returned by the subquery. All queries from the dual pseudo table return one row unless a UNION or UNION ALL set operator fabricates a multiple row set. See Chapter 11 for details on the use of set operators to fabricate data sets.

The IN, =ANY, and =SOME lookup operators work when the subquery returns one or more rows. It's always best to use a lookup operator unless you want to raise an exception when the subquery returns more than one row. This type of exception signals when a prior business rule has been violated—the rule that the subquery supports.

The following demonstrates a DELETE FROM statement with a multiple column lookup operator:

```
SQL> DELETE FROM item
  2  WHERE (item_title,item_rating) IN
  3          (SELECT 'Pirates of the Caribbean: On Stranger Tides'
  4           ,       'PG-13'
  5           FROM    error_item);
```

Oracle and MySQL both support the demonstrated DELETE FROM syntax variations, which eliminates the need for a specialized MySQL section in this chapter. A separate section is required, however, to cover the use of a DELETE FROM statement when you're working with nested table elements in rows of a table in Oracle.

Delete Nested Table Row Elements

As mentioned, nested tables are object relational database management system (ORDBMS) structures. MySQL doesn't support nested tables, but Oracle does. Chapters 6 and 7 showed you how to create and alter tables with nested tables. Chapter 8 showed you how to insert nested tables, and Chapter 9 showed you how to update nested tables.

The following example builds on the employee table introduced in Chapters 6, 7, 8, and 9. The following data should be included in the table by the conclusion of Chapter 9, but it won't be formatted like the following (which was reformatted to fit on the printed page):

```
  ID Full Name     Street Address Nested Table
---- ------------- ------------------------------------------------------
   1 Yosemite Sam  ADDRESS_LIST
                   ( ADDRESS_TYPE
                     (1,STREET_LIST(...),'Oakland','CA','94612')
                   , ADDRESS_TYPE
                     (2,STREET_LIST(...),'Oakland','CA','94612')
                   )
   2 Bugs Bunny    ADDRESS_LIST
                   ( ADDRESS_TYPE
```

```
                    (1,STREET_LIST(...),'Beverly Hills','CA','90210')
                  , ADDRESS_TYPE
                    (2,STREET_LIST(...),'Beverly Hills','CA','90210')
                  )
```

Previous `DELETE FROM` statements would let you remove the row with *Yosemite Sam* or *Bugs Bunny* but not an element of the nested employee table. The `DELETE FROM` statement applied against a view of the nested table would let you remove a row element.

The following statement lets you remove a row element from the nested table:

```
DELETE FROM TABLE (SELECT e.home_address
                   FROM   employee e
                   WHERE  e.employee_id = 1) ha
WHERE  ha.address_id = 1;
```

This `DELETE FROM` statement removes row elements from the view created inside the call to the `TABLE` function. The `SELECT` list inside the runtime view returns row elements for deletion. The `WHERE` clause for the `DELETE FROM` statement identifies the row to process.

This works only on collection of user-defined object types. It doesn't work for nested tables built as collections of a scalar data type, such as a date, number, or string. You must replace the collection of a scalar data type with a new collection that doesn't include the undesired element. PL/SQL lets you read through and eliminate undesired elements from any nested table structure. While reading the records, you can capture all the records you want to keep and then update the table's collection with the locally stored collection values.

After the preceding `DELETE FROM` statement, you would hold the following in the `employee` table:

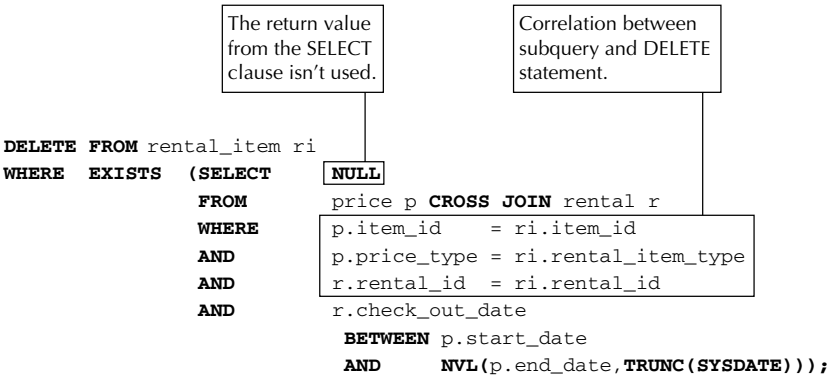
```
  ID Full Name     Street Address Nested Table
---- ------------- -------------------------------------------------
   1 Yosemite Sam  ADDRESS_LIST
                   ( ADDRESS_TYPE
                     (2,STREET_LIST(...),'Oakland','CA','94612')
                   )
   2 Bugs Bunny    ADDRESS_LIST
                   ( ADDRESS_TYPE
                     (1,STREET_LIST(...),'Beverly Hills','CA','90210')
                   , ADDRESS_TYPE
                     (2,STREET_LIST(...),'Beverly Hills','CA','90210')
                   )
```

Notice that the first nested row has been removed from the nested table in the Yosemite Sam row. Maintenance on nested tables is possible when they are collections of object types, but it's not possible when they are collections of scalar variables.

Delete by Correlated Queries

Although deletions can remove one to many rows when the conditions of the `WHERE` clause are met, they can also work with joins between tables. Like the `UPDATE` statement, the `DELETE FROM` statement supports correlated joins that allow you to work with multiple tables when deleting rows. A separate multiple-table syntax exists for MySQL, which uses standard join syntax covered in more depth in Chapter 11.

Correlated joins use the EXISTS keyword in the WHERE clause. The actual equality or inequality of the join is in the WHERE clause of the subquery. The subquery has scope access to the target table of the DELETE FROM statement, shown in the following illustration, which makes referencing it in the subquery possible.



There is one difference between Oracle and MySQL databases in this statement. The NVL function is an Oracle built-in function, and the equivalent for a MySQL database is the IFNULL function. The last line would look like this in MySQL:

```
BETWEEN p.start_date AND IFNULL(p.end_date, UTC_DATE()));
```

Correlated subqueries in the WHERE clause don't return a value, because the match occurs in the subquery's WHERE clause. That's why a NULL is frequently returned from correlated subqueries, but you can return anything you'd like. In the preceding example, two columns from the price table and one column from the rental table match three columns in the target rental_item table of the DELETE FROM statement. The three columns from the rental_item table are the natural key. Together they uniquely identify rows as the natural key. A match between the three columns and a range filter against a start and end date guarantees unique row deletions.

MySQL Delete by Multiple-Table Statement

The multiple-table DELETE FROM statement uses interesting syntax, but it is not necessary, because correlation works well. It also isn't portable, whereas correlated subuqeries are portable to other databases. Unlike the multiple-table UPDATE statement, the multiple-table DELETE FROM statement uses both ANSI SQL-89 comma-delimited tables in the FROM clause and ANSI SQL-92 with join syntax.

The two subsections cover the ANSI SQL-89 and ANSI SQL-92 syntax. They offer different approaches, but the advantage of ANSI SQL-92 is its support of outer joins.

ANSI SQL-89 Multiple-Table Statement

Here's the generic ANSI SQL-89 prototype:

```
DELETE target_table
FROM target_table, join_table alias [, join_table alias [, …]]
WHERE [NOT] column_name {{= | <> | > | >= | < | <=} |
                         [NOT] {{IN | EXISTS} | IS NULL}} 'expression'
[{AND | OR } [NOT] comparison_operation] [...];
```

In the ANSI SQL-89 approach, the multiple-table `DELETE FROM` statement doesn't support the use of table aliases for the target table but does for join tables. Join conditions are placed in the `WHERE` clause and qualified by the target table name or the join table name or alias.

In the following example, the `rental_item` table is the target table. You also must repeat the `rental_item` table in the `FROM` clause.

The following converts the previous `DELETE FROM` correlated subquery into a multiple-table `DELETE` statement:

```
DELETE rental_item
FROM   rental_item, price p, rental r
WHERE  p.item_id    = rental_item.item_id
AND    p.price_type = rental_item.rental_item_type
AND    r.rental_id  = rental_item.rental_id
AND    r.check_out_date BETWEEN p.start_date
                        AND     IFNULL(p.end_date,UTC_DATE());
```

ANSI SQL-92 Multiple-Table Statement

Here's the generic ANSI SQL-92 prototype:

```
DELETE table_alias, table_alias [, table_alias [, ...]]
FROM   table [[AS] alias] {LEFT | RIGHT | INNER | FULL} JOIN
       table [[AS] alias]
{ USING ( column [, column [, … ]]) |
    ON {table | alias}.column = {table | alias}.column
 [ AND {table | alias}.column = {table | alias}.column
 [ AND ... ]}
[ another_join [ ...]]
WHERE [NOT] column_name {{= | <> | > | >= | < | <=} |
                         [NOT] {{IN | EXISTS} | IS NULL}} 'expression'
[{AND | OR } [NOT] comparison_operation] [...];
```

In the ANSI SQL-92 approach, the multiple-table `DELETE FROM` statement does support the use of table aliases for all tables. Join conditions are placed in the `USING` or `ON` subclauses of the `FROM` clause.

The following converts the previous multiple-table `DELETE` statement into the newer syntax and demonstrates `INNER JOIN` syntax. A `INNER JOIN` in this case deletes rows from the primary table whether or not matches are found in the other tables.

```
DELETE ri, p, r
FROM   rental_item AS ri INNER JOIN price AS p
ON ri.item_id = p.item_id AND ri.rental_item_type = p.price_type
INNER JOIN rental AS r ON ri.rental_id = r.rental_id
WHERE r.check_out_date BETWEEN p.start_date AND IFNULL(p.end_date,UTC_DATE());
```

The choice of whether you delete by correlation or multiple-table syntax is yours, but correlation is the best practice and uses the most portable syntax. The multiple-table syntax is more like the syntax of joins in queries, and that's probably why developers like to use it.

Summary

This chapter covered how you delete data. As with the `INSERT` and `UPDATE` statements, you can delete based on literal values or on subqueries. Correlated subqueries let you validate related tables for matching values and generally ensure greater control and smaller procedural code segments.

Mastery Check

The mastery check is a series of true or false and multiple choice questions that let you confirm how well you understand the material in the chapter. You may check the Appendix for answers to these questions.

1. True ☐ False ☐ A `DELETE FROM` statement supports multiple-row deletes when more than one row matches the conditions of the `WHERE` clause.

2. True ☐ False ☐ A `DELETE FROM` statement has the same syntax in Oracle as MySQL when the `WHERE` clause uses values, subqueries, or correlated subqueries.

3. True ☐ False ☐ The deletion of rows is immediate in all cases.

4. True ☐ False ☐ A `TRUNCATE` statement deletes rows similar to a `DELETE FROM` statement and writes pending changes to a redo log file.

5. True ☐ False ☐ In the scope of a transaction, the `DELETE FROM` statement has two phases when the table uses a MyISAM engine.

6. True ☐ False ☐ In the scope of a transaction, the `DELETE FROM` statement has two phases when the table uses an InnoDB engine.

7. True ☐ False ☐ Oracle databases support a multiple-table `DELETE FROM` statement that uses ANSI SQL-92 syntax with `JOIN` keywords.

8. True ☐ False ☐ MySQL databases support a multiple-table `DELETE FROM` statement that uses ANSI SQL-89 syntax with `JOIN` keywords.

9. True ☐ False ☐ Oracle and MySQL support multiple-column matches from subqueries.

10. True ☐ False ☐ A non-correlated subquery always returns a null value.

11. Which of the following comparison operators work with a subquery that returns a single row with a single column?

- **A.** `=`
- **B.** `=ANY`
- **C.** `=ALL`
- **D.** `IN`
- **E.** All of the above

12. Which of the following comparison operators work with a subquery that returns a single row with multiple columns?

- **A.** =
- **B.** =ANY
- **C.** =ALL
- **D.** IN
- **E.** All of the above

13. Which of the following comparison operators work with a subquery that returns multiple different rows with multiple columns?

- **A.** =
- **B.** =ANY
- **C.** =NONE
- **D.** =ALL
- **E.** All of the above

14. A multiple-table DELETE FROM statement requires how many references to the target table?

- **A.** 0
- **B.** 1
- **C.** 2
- **D.** 3
- **E.** None of the above

15. Which type of DELETE FROM statement doesn't support table aliases for the target table?

- **A.** A DELETE FROM statement that only performs value comparisons in the WHERE clause
- **B.** A DELETE FROM statement that uses value and expression (subquery) comparisons in the WHERE clause
- **C.** A DELETE FROM statement that uses correlated subqueries in the WHERE clause
- **D.** A multiple-table DELETE FROM statement that uses joins in the FROM clause
- **E.** None of the above