

CHAPTER 9

Updating Data



he UPDATE statement lets you change data in tables and views by resetting values in one or more columns. A single UPDATE statement can change one, many, or all rows in a table. The new values can come from literal values, variables, or correlated query results. *Correlation* is the matching of one set of data with another through join operations. This chapter discusses equijoin (equality value joins); Chapter 11 examines join options in detail.

Oracle and MySQL implement basic UPDATE statement syntax in a similar way, but Oracle supports a record update and large object interface that aren't available in MySQL. The large object interface appends the RETURNING INTO clause to UPDATE statements. MySQL supports a multiple table UPDATE statement, a priority flag, and a LIMIT phrase, which are not supported in Oracle.

This chapter covers how you do the following:

- Update by values and queries
- Update by correlated queries

An UPDATE statement's most important behavior is that it works against all rows in a table unless a WHERE clause or correlated join limits the number of rows. This means you should always limit which rows should be changed by providing a WHERE clause or correlated join. The list of columns in the SET clause of an UPDATE statement is the expression list.

Changes by the UPDATE statement are hidden until committed in an Oracle database, but are immediately visible in nontransactional databases such as MySQL. Therefore, you must start a transaction scope in MySQL and work with tables that use the InnoDB engine when you want to use ACID-compliant UPDATE statements.

Update by Values and Queries

Some UPDATE statements use date, numeric, or string literals in the SET subclause. The SET subclause can work with one to all columns in a table, but you should never update a primary surrogate key column. Any update of the externally known identifier column risks compromising the referential integrity of primary and foreign keys.

The generic UPDATE statement prototype with values resetting column values looks like this:

```
UPDATE some_table
SET   column_name = 'expression'
[,   column_name = 'expression' [, ...]
WHERE [NOT] column_name { { = | <> | > | >= | < | <= } |
                           [NOT] { { IN | EXISTS } | IS NULL } } 'expression'
[{AND | OR} [NOT] comparison_operation] [...];
```

The target table of an UPDATE statement can be a table or updateable view. An expression can be a numeric or string literal or the return value from a function or subquery. The function or subquery must return only a single row of data that matches the data type of the assignment target. The right operand of the assignment can contain a different data type when its type can be implicitly cast to the column's data type, or explicitly cast to it with the CAST or proprietary built-in function. In the generic example, a subquery needs to return a single column and row (this type of subquery is a *scalar subquery* or *SQL expression*). Ellipses replace multiple listing in the SET and WHERE clauses.

The WHERE clause lets you evaluate truth or non-truth, which is the purpose of each comparison operation. The comparison operators in the prototype are divided into sets of related operators by using curly braces—first the math comparisons, then the set and correlation comparisons, and finally the null comparison. The {AND | OR} [NOT] are logical operators. The AND operator evaluates the truth of two comparisons, or, with enclosing parentheses, the truth of sets of comparison operations. The OR operator evaluates the truth of one or the other comparison operator, and it employs short-circuit evaluation (the statement is true when the first comparison is true). The negation operator (NOT) checks whether a statement is false.

An actual UPDATE statement against an item table would look like this when you enter the actual movie name in lieu of a placeholder value:

```
SQL> UPDATE item
2 SET item_title = 'Pirates of the Caribbean: On Stranger Tides'
3 , item_rating = 'PG-13'
4 WHERE item_title = 'Pirates of the Caribbean 4';
```

Variations to this syntax exist in Oracle, but this is the basic form for UPDATE statements in both databases. Specifics for Oracle and MySQL are covered in the following sections.

Oracle Update by Values and Queries

The biggest difference between Oracle and other databases is that Oracle allows you to reset record structures, not just columns. Recall from Chapter 6 that the definition of a table is equivalent to the definition of a record structure, and a record structure is a combination of two or more columns (or fields).

The prototype of an UPDATE statement for Oracle differs from the generic profile, as you can see:

```
UPDATE {some_table | TABLE(query_statement)}
SET {column_name = 'expression' | DEFAULT |
    (column_list) = (expression_list)} [, ...]
WHERE [NOT] {column_name | (column_list)}
    {{= | <> | > | >= | < | <=} |
    [NOT] {IN | =ANY | =SOME | =ALL } |
    [NOT] {IS NULL | IS SET} | [NOT] EXISTS} {'expression' | (expression_
list)}
[ {AND | OR } [NOT] comparison_operation] [...]
[RETURNING {column_name | (column_list)}
    INTO {local_variable | (variable_list)}];
```

Oracle extends the target of the UPDATE statement from a table or view (traditionally a named query inside the data catalog) to a result set. In Oracle's lexicon, the result set is formally an aggregate result set—that is, the result set acts like a normal query's return set in processing memory (inside the SGA, or System Global Area). The TABLE function makes this possible. (The TABLE function was previously known as the THE function—that's no joke, but ancient history from Oracle 8i, along with some error messages that have never been updated.)

Oracle also extends the behavior of assignment in the SET operator by allowing you to assign a record structure to another record structure. A (data) record structure in the SET operator is any list of two or more columns from the table definition, which is less than the complete data structure of the table or its definition in the data catalog. Ellipses replace the continuing list of possible elements in the SET and WHERE clauses.

The WHERE clause comparison operators are also expanded in an Oracle database. They are separated by curly braces, like the generic prototype, with math comparisons, set comparisons, null comparisons, and correlations. Set comparisons act as lookup operators and are covered in the “Multiple Row Subqueries” section of Chapter 11, and correlation is explained in the “Correlated Queries” section of the same chapter.

The RETURNING INTO clause allows you to shift a reference to columns that you’ve updated but not committed into variables. Those variables are critical to how you update large objects in the database.

Here’s an example of how you would use Oracle’s record structure assignment operation in a SET clause:

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

The values reset the columns item_title and item_rating on all lines where item_title is “Pirates of the Caribbean 4.” The subquery uses string literals inside a query against the dual table. This is straightforward and not much different from the comma-delimited SET clauses for each column. You might wonder why you should bother with implementing this twist on the other syntax. That’s a great question! There’s not much added value with date, numeric, or string literals from the pseudo table dual, but the value occurs when the source is a row returned from a query. The record structure syntax allows you to assign a row’s return values directly from a single-row subquery with multiple columns to a row of the target table.

Here’s an example of an assignment from a subquery to record structure:

```
SQL> UPDATE item
2 SET (item_title, item_rating) =
3     (SELECT item_title, item_rating
4         FROM import_item
5         WHERE item_barcode = 'B004A8ZWUG')
6 WHERE item_title = 'Pirates of the Caribbean 4';
```

The item_title and item_rating values from the subquery are assigned to the equivalent columns in the item table when the item_title column holds the string literal value. The power of this type of assignment increases when you add correlation, because you can process sets of data in a single UPDATE statement. (That’s covered in the “Update by Correlated Queries” section later in this chapter.)

Two specialized forms of UPDATE statements are included in the Oracle 11g database. One works with collections of object types, and the other works with scalar and large object types. The ability to use the result of the TABLE function inside an UPDATE statement lets you update nested tables (collections of object types). A RETURNING INTO clause supports scalar and large objects by returning the values or references from the UPDATE statement to the calling scope. The calling scope is the SQL*Plus session in the examples but could be an external program written in PL/SQL or C, C++, C#, or Java. This technique gives you access to recently updated values without your having to requery the table, and in the case of large objects, it allows you to read and write to large objects through a web application.

RETURNING INTO Clause

You can append the `RETURNING INTO` clause to any `UPDATE` statement. The `RETURNING INTO` clause lets you retrieve updated column values into locally scoped variables so that you can avoid requerying the columns after the `UPDATE` statement.

To demonstrate this concept, even the smallest example uses session-level bind variables. The bind variables eliminate the need for a procedural programming language such as Java or PHP to demonstrate the concept.

Recall from Chapter 2 that SQL*Plus commands declare session-level bind variables. This example requires a pair of session-level variables to act as target of the `RETURNING INTO` clause. You can declare these two bind variables with this syntax:

```
SQL> VARIABLE bv_title VARCHAR2(60)
SQL> VARIABLE bv_rating VARCHAR2(60)
```

The following demonstrates an `UPDATE` statement that uses the `RETURNING INTO` phrase:

```
SQL> UPDATE item
2 SET      (item_title,item_rating) =
3          (SELECT 'Pirates of the Caribbean: On Stranger Tides'
4             ,      'PG-13'
5             FROM dual)
6 WHERE item_title = 'Pirates of the Caribbean 4'
7 RETURNING item_title, item_rating INTO :bv_title, :bv_rating;
```

The values updated into the table are returned in the local variables. They can be displayed by using SQL*Plus formatting and a query:

```
COLUMN bv_title  FORMAT A44 HEADING ":bv_title"
COLUMN bv_rating FORMAT A12 HEADING ":bv_rating"
SELECT :bv_title AS bv_title, :bv_rating AS bv_rating FROM dual;
```

The `HEADING` value is enclosed in double quotes so that a colon can be used in the column titles. This returns the literal values from the query against the `dual` table:

```
:bv_title                                :bv_rating
-----
Pirates of the Caribbean: On Stranger Tides PG-13
```

Note that the `RETURNING INTO` phrase has several restrictions:

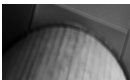
- It fails when updating more than a single row.
- It fails when the expression list includes a primary key or other `NOT NULL` column when a `BEFORE UPDATE` trigger is defined on the table.
- It fails when the expression list includes a `LONG` data type.
- It fails when the `UPDATE` statement is parallel processing or working against a remote object.
- It is disallowed when updating a view that has an `INSTEAD OF` trigger.

Returning scalar, `BLOB`, or `CLOB` data types is the most effective way to leverage the `RETURNING INTO` phrase. The `RETURNING INTO` phrase is very advantageous in web applications. A web

application would implement a stored procedure (see Chapter 13) to start a transaction context and pass a reference for updating a CLOB column. The following (explained in more detail later) provides an example of such a procedure:

```
SQL> CREATE OR REPLACE PROCEDURE web_load_clob_from_file
2  ( pv_item_id      IN      NUMBER
3  , pv_descriptor IN OUT CLOB ) IS
4  BEGIN
5      -- A FOR UPDATE makes this a DML transaction.
6      UPDATE      item
7      SET         item_desc = empty_clob()
8      WHERE       item_id = pv_item_id
9      RETURNING  item_desc INTO pv_descriptor;
10 END web_load_clob_from_file;
11 /
```

The `pv_descriptor` parameter in the procedure’s signature on line 3 uses an `IN OUT` mode of operation, which is a *pass-by-reference* mechanism. It effectively enables the sending of a reference to the CLOB column out to the calling program. The `RETURNING INTO` clause assigns the reference to the parameter on line 9. With the reference, the external program can then update the CLOB column.



NOTE
Check Chapter 8 in Oracle Database 11g PL/SQL Programming (McGraw-Hill, 2008) for the details on how to write this type of programming logic.

Nested Tables

Nested tables are lists, which are like arrays but without a maximum number of rows. As such, lists mimic database tables when they’re defined by object types. Object types act like record data structures in an Oracle database. This is possible because Oracle is an object relational database.

The original SQL design didn’t consider the concept of object types or collections of object types. This leaves Oracle with the responsibility to fit calls to these object types within SQL extensions. The interface is rather straightforward but has limitations as to what you can perform on nested tables and arrays through `INSERT` and `UPDATE` statements. You can insert or update complete nested tables, but you can replace only certain elements of the nested tables. PL/SQL lets you access and manipulate the elements of nested tables and arrays.

This example revisits the `employee` table from Chapters 6 and 8. Here’s the definition of the table:

Name	Null?	Type
-----	-----	-----
EMPLOYEE_ID		NUMBER
FIRST_NAME		VARCHAR2 (20)
MIDDLE_NAME		VARCHAR2 (20)
LAST_NAME		VARCHAR2 (20)
HOME_ADDRESS		ADDRESS_LIST

How to Write to a CLOB Column from PHP

Here's an example that might help you better understand this feature. Although this book isn't about how to write PHP to work with an Oracle or MySQL database, PHP can show you how to capture the handle from the `web_load_clob_from_file` procedure.

Here's part of a PHP function to write a large file to a CLOB column:

```
if ($c = @oci_connect(SCHEMA,PASSWD,TNS_ID)) {
    // Declare input variables.
    (isset($_POST['id'])) ? $id = (int) $_POST['id'] : $id = 1021;
    (isset($_POST['title'])) ? $title = $_POST['title'] : $title = "Harry #1";

    // Declare a PL/SQL statement and parse it.
    $stmt = "BEGIN web_load_clob_from_file(:id,:item_desc); END;";
    $s = oci_parse($c,$stmt);

    // Define a descriptor for a CLOB and variable for CLOB descriptor.
    $rlob = oci_new_descriptor($c,OCI_D_LOB);
    oci_define_by_name($s,':item_desc',$rlob,SQLT_CLOB);

    // Bind PHP variables to the OCI types.
    oci_bind_by_name($s,':id',$id);
    oci_bind_by_name($s,':item_desc',$rlob,-1,SQLT_CLOB);

    // Execute the PL/SQL statement.
    if (oci_execute($s,OCI_DEFAULT)) {
        $rlob->save($item_desc);
        oci_commit($c); }

    // Release statement resources and disconnect from database.
    oci_free_statement($s);
    oci_close($c); }

else {
    // Assign the OCI error and manage error.
    $errorMessage = oci_error();
    print htmlentities($errorMessage['message'])."<br />";
    die(); }
```

The four boldfaced lines make reading and writing to the CLOB column possible. The lines, respectively, define an anonymous PL/SQL block as a statement, create a socket, map the placeholder and the statement to the socket, and write the CLOB through the socket to the file.

If you're interested in learning more about PHP and the Oracle database, you can check *Oracle Database 10g Express Edition PHP Web Programming* (McGraw-Hill, 2006). The first 12 chapters cover PHP and the last 3 cover Oracle's OCI library.

The `home_address` column is a user-defined type (UDT) collection named `address_list`. To avoid your having to flip back to Chapter 7, I'll explain here: The `address_list` UDT holds an `address_type` UDT (object type that acts like a record data structure), and the `address_type` UDT holds another nested table of a scalar variable. This means the table holds multiple nested tables.

You can also describe the `address_list` UDTs with the `DESCRIBE` command in SQL*Plus:

```
address_list TABLE OF ADDRESS_TYPE
```

Name	Null?	Type
ADDRESS_ID		NUMBER
STREET_ADDRESS		STREET_LIST
CITY		VARCHAR2 (30)
STATE		VARCHAR2 (2)
POSTAL_CODE		VARCHAR2 (10)

This collection is a nested table. You can tell that because it says `TABLE OF`. A UDT array would print a `VARRAY(n) OF` phrase before the UDT name. This example includes a nested `street_list` collection. You can describe it the same way and it shows the following:

```
street_list TABLE OF VARCHAR2(30)
```

As mentioned, this type of table structure is called *multiple table nesting*. It is inherently complex. This type of design also presents migration issues when you want to modify the UDTs. You must put the data some place, drop the table, and then add the UDTs in the reverse order of how you created them—at least until you arrive at the UDT that you want to change. After making the change, you'll need to re-create all data types and tables and migrate the data back into the new table.

When you perform an update, you need to replace the entire nested table element. You can read out what's there and identify where the change goes through PL/SQL or some other procedural language that leverages the Oracle Call Interface (OCI), Open Database Connectivity (ODBC), or Java Database Connectivity (JDBC). This assumes you've written the code logic to capture all existing data and dynamically construct an `UPDATE` statement. Native Dynamic SQL (NDS) lets you dynamically create these types of SQL statements, which are like prepared statements in MySQL. You can read about NDS in Chapter 13 and prepared statements in Chapter 14.

In Chapter 8, we inserted a row into the `EMPLOYEE` table. With a forward reference to material in Chapter 11, here's how you would extract the information from the nested tables into an ordinary result set of scalar columns:

```
-- These SQL*Plus commands format the columns for display.
```

```
COLUMN employee_id FORMAT 999 HEADING "ID|EMP"
COLUMN full_name    FORMAT A16 HEADING "Full Name"
COLUMN address_id   FORMAT 999 HEADING "ID|UDT"
COLUMN st_address   FORMAT A16 HEADING "Street Address"
COLUMN city         FORMAT A8  HEADING "City"
COLUMN state        FORMAT A5  HEADING "State"
COLUMN postal_code  FORMAT A5  HEADING "Zip|Code"
```

```
SQL> SELECT      e.employee_id
              2 ,      e.first_name || ' ' || e.last_name AS full_name
```



```

3      ,      st.address_id
4      ,      sa.column_value AS st_address
5      ,      st.city
6      ,      st.state
7      ,      st.postal_code
8 FROM    employee e CROSS JOIN
9          TABLE(e.home_address) st CROSS JOIN
10         TABLE(street_address) sa
11 ORDER BY 2, 3;

```

This SELECT statement uses the cross join to extract nested table material to the single containing row that holds it. In this process, the cross join makes copies of the content of the single row for each row of the nested table. This example first unwinds `street_address` within `home_address`, and then `home_address` within the container `employee` table. It returns four rows, because there are two rows in each of the nested tables and only one row in the sample table. Cross joins yield Cartesian products, which are the number of rows in one set times the other set, or in this case the multiplied product of rows in three sets ($1 \times 2 \times 2 = 4$). The statement above renders the following output:

ID	ID	Zip
EMP Full Name	UDT Street Address	City State Code
1 Sam Yosemite	1 1111 Broadway	Oakland CA 94612
1 Sam Yosemite	1 Suite 322	Oakland CA 94612
1 Sam Yosemite	2 1111 Broadway	Oakland CA 94612
1 Sam Yosemite	2 Suite 525	Oakland CA 94612

Let's assume you want to change the Suite 322 in the second row to Suite 521. The UPDATE statement would look like this when you replace the entire structure:

```

SQL> UPDATE employee e
2 SET    e.home_address =
3        address_list(
4            address_type( 1
5                            , street_list('1111 Broadway','Suite 322')
6                            , 'Oakland'
7                            , 'CA'
8                            , '94612')
9            , address_type( 2
10                           , street_list('1111 Broadway','Suite 521')
11                           , 'Oakland'
12                           , 'CA'
13                           , '94612'))
14 WHERE  e.first_name = 'Sam'
15 AND    e.last_name = 'Yosemite';

```

The syntax to replace the content of a UDT uses the name of the data type as an object constructor and then provide a list. Lines 4 to 8 are highlighted to demonstrate the constructor for an `address_type` UDT. Line 10 is separately highlighted to show the constructor for a `street_list` UDT. In the preceding statement, a comma-delimited list lets you construct nested tables. Needless to say, you probably want to nest only data that changes infrequently and that fails to merit its own table.

252 Oracle Database 11g & MySQL 5.6 Developer Handbook

You can also replace only an element of the nested `address_type` by using some complex `UPDATE` syntax. The `UPDATE` statement is complex because it uses a query to find a nested table in one row of the `employee` table. The `TABLE` function then casts the object collection into a SQL result set (formally, an aggregate result set). This type of result set can also be called an inline view, runtime table, derived table, or common table expression. The `UPDATE` statement lets you change the city value for the first element of the `address_type` UDT in the `address_list` collection:

```
SQL> UPDATE TABLE (SELECT e.home_address
2          FROM employee e
3          WHERE e.employee_id = 1) e
4 SET e.city = 'Fremont'
5 WHERE e.address_id = 1;
```

Unfortunately, the city was correct for the address but *Suite 521* is wrong. It should be *Suite 522*. There is no way to replace only one element of a varray or nested table of a scalar data type. An attempt would use the same cross joining logic shown earlier in the query that unfolds nested tables, like so:

```
SQL> UPDATE TABLE (SELECT addr.street_address
2          FROM employee e CROSS JOIN TABLE(e.home_address) addr
3          WHERE e.employee_id = 1
4          AND addr.address_id = 1)
5 SET column_value = 'Suite 522'
6 WHERE column_value = 'Suite 521';
```

Although the query returns the expected result set, the assignment in the `SET` clause fails. You can't make an assignment to the default `column_value` column returned by an unwound nested table of a scalar data type. It raises an ORA-25015 error:

```
SET column_value = 'Suite 522'
*
ERROR at line 5:
ORA-25015: cannot perform DML ON this nested TABLE VIEW COLUMN
```

The error documentation does not seem to explain why it doesn't work. Hazarding a guess, I think that collections of scalar data types are handled differently than collections of UDTs. At least, there's a difference between them because scalar collections work in the result cache PL/SQL functions while collections of UDTs don't.

The following lets you reset the city and replaces the nested address element:

```
SQL> UPDATE TABLE (SELECT e.home_address
2          FROM employee e
3          WHERE e.employee_id = 1) e
4 SET e.street_address = street_list('1111 Broadway','Suite 522')
5 , e.city = 'Oakland'
6 WHERE e.address_id = 1;
```

Line 4 stores a complete constructor of the scalar collection. It's not terribly difficult when only a few elements exist, but it becomes tedious with long lists. The alternative to an `UPDATE` statement like these is PL/SQL, which allows you to navigate the collections element-by-element and then process the individual list elements.

In general, with collections, you store nested tables, varrays, and object types the same way—with a call to their object type, which serves as a constructor method. You pass values inside the parentheses as actual parameters (also called arguments). The `TREAT` function lets you instantiate these in memory.

NOTE

Chapter 13 shows you how to implement a basic object type, and it and the SQL statements that support it are discussed there. Chapter 10 in Oracle Database 11g PL/SQL Programming Workbook or Chapter 14 in Oracle Database 11g PL/SQL Programming (McGraw-Hill, 2008) cover the details of object types.

Large Objects

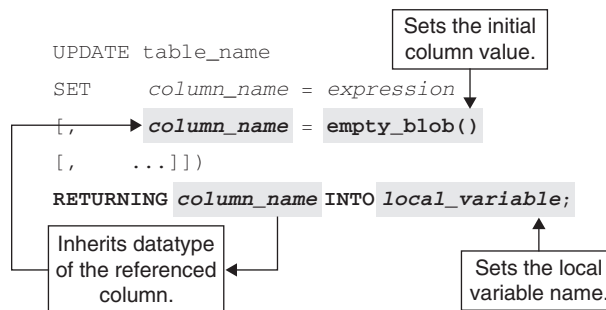
Large objects present complexity in Oracle, because you need to load them by segments. After all, they can grow to 32 terabytes in size. BLOB and CLOB are the only two data types stored physically inside the database. The other large object type, BFILE, is a locator that points to an external directory location (see Chapter 12) and filename. The first argument is a call to a virtual directory that you've created in the database, and the second argument is the relative filename.

An ordinary `UPDATE` statement handles changes to BFILE locators like this:

```
SET = bfilename('virtual_directory_name', 'relative_file_name')
```

The BLOB and CLOB data types require special handling. The most common need is to overwrite the column value. That's because these are binary or character streams and they're seldom simply edited.

The following illustration shows you how to update a BLOB column. BLOB columns larger than 4 kilobytes are stored out-of-line from the transactional table, because they're infrequently changed and less frequently backed-up. It is also common practice to change these columns by themselves after any updates to the scalar columns of a table.



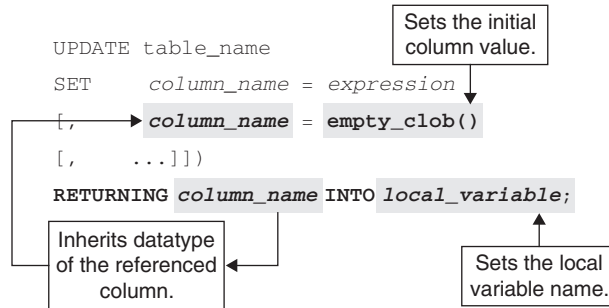
This prototype statement demonstrates the syntax if you were to update a single row, and the `WHERE` clause is excluded to simplify the illustration. The first step of an update to a BLOB column is re-initialization, which occurs with an `EMPTY_BLOB()` function call. The second phase maps the column name in a row to an external stream (receiving end), which uses the column name as an identifier. The third step assigns the stream (originating end) to a local program variable.

254 Oracle Database 11g & MySQL 5.6 Developer Handbook

Basically, this is a socket communication between a program and the database, and it lasts until all segments have been loaded into the column.

The local variable in this example can be a PL/SQL variable or any external OCI, ODBC, or JDBC programming language variable. The local variable data type must support a mapped relationship to the native Oracle data type.

The CLOB data type works the same way, and as you can see in the next illustration, there's no SQL statement difference. Only the call to the `EMPTY_CLOB()` differs, but it's an important difference that you shouldn't overlook.



The `XML_TYPE` data types use the `EMPTY_CLOB()` function to clear the columns' contents. After all, `XML_TYPE` columns are specializations of the `CLOB` data type.

MySQL Update by Values

MySQL updates by value just as Oracle `UPDATE` statements do—at least, they do this the same way when assigning values to single columns in the `SET` subclause. The interesting thing about MySQL is that you don't require specialized handling for large binary or character strings. This simplifies things considerably.

MySQL does give you some added options that aren't available in Oracle `UPDATE` statements. The options let you do the following:

- Set the runtime priority for an `UPDATE` statement.
- Ignore errors in an `UPDATE` statement.
- Add an `ORDER BY` clause to an `UPDATE` statement.
- Set a `LIMIT` on rows processed by an `UPDATE` statement.

Given the number of additional options, here's a MySQL-specific prototype:

```

UPDATE [LOW_PRIORITY] [IGNORE] table_name
SET   column_name = expression [, ...]
WHERE [NOT] {column_name | expression} = {expression | DEFAULT}
[AND | OR] [NOT] {column_name | expression} = {expression | DEFAULT} [...]
[ORDER BY {select_list_element | position_id} [ASC|DESC] [, ... ]]
[LIMIT number_of_rows];

```

The IGNORE keyword lets the UPDATE statement complete when it encounters errors. The LOW_PRIORITY keyword lets you defer UPDATE statement processing behind other priority tasks. Note that it's not a good idea to use the LOW_PRIORITY keyword unless you're performing maintenance and interactive users are disconnected from the system, because deferred processing can lead to transactional inconsistencies. Ellipses indicate that you can add elements to the SET and WHERE clauses.

A new small table seems appropriate to show some of the features of the MySQL UPDATE statement with a value clause. The following creation statement defines the table:

```
mysql> CREATE TABLE teeshirt
-> ( teeshirt_id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
-> , teeshirt_slogan VARCHAR(60)
-> , teeshirt_size VARCHAR(10) DEFAULT 'Medium');
```

A DEFAULT value is assigned to the third column. An INSERT statement uses the DEFAULT value whenever an override signature excludes the column name. An UPDATE statement uses the default value whenever it is explicitly assigned to a column in a SET clause. Here's a data set with five rows:

teeshirt_id	teeshirt_slogan	teeshirt_size
1	Spartans	Large
2	Cardinals	X-Large
3	Cal Bears	X-Large
4	Bruins	2X-Large
5	Trojans	Large

The following update statement changes the *Cal Bears* slogan string to *Bears* and uses the default to set the tee shirt size to *Medium*:

```
mysql> UPDATE teeshirt
-> SET teeshirt_slogan = 'Bears'
-> , teeshirt_size = DEFAULT
-> WHERE teeshirt_slogan = 'Cal Bears';
```

A query of the modified data set returns the following changes for the row identified by a 3 in the teeshirt_id column:

teeshirt_id	teeshirt_slogan	teeshirt_size
3	Cal Bears	Medium

The LIMIT clause restricts how many rows are impacted by the UPDATE statement. This is useful when you want to update only a certain number of rows, but it's most useful when you combine it with the ORDER BY clause, which allows you to pre-sort the records before applying the LIMIT clause. This way, the LIMIT clause is targeted based on the ordering of rows.

The following update will set the tee shirt size to *Medium* for the two highest surrogate key ID values because the `ORDER BY` clause performs a descending sort:

```
mysql> UPDATE    teeshirt
-> SET          teeshirt_size = DEFAULT
-> ORDER BY     teeshirt_id DESC
-> LIMIT 2;
```

The `ORDER BY` and `LIMIT` clauses were the only restrictions on which columns were changed by the `UPDATE` statement. Here are the modified results:

teeshirt_id	teeshirt_slogan	teeshirt_size
1	Spartans	Large
2	Cardinals	X-Large
3	Bears	Medium
4	Bruins	Medium
5	Trojans	Medium

Although this technique is possible, it’s generally a good idea to use it only when you’re working with changes to a small data set. The `WHERE` clause should generally filter the rows changed by the `UPDATE` statement.

Update by Correlated Queries

Correlated queries let you change data in columns based on the join to other tables. The join statement is the point of correlation—comparison of two or more objects. The join statement is always in the `WHERE` clause of the correlated subquery, because it has scope access to the containing DML statement. Basically, you can think of the `UPDATE` statement as calling a correlated subquery, like a function. A `UPDATE` statement calls any correlated subquery once for every row it processes.

Oracle and MySQL databases support correlated `UPDATE` statements. MySQL also supports multiple-table `UPDATE` statements that act like correlated queries. The section on MySQL correlated queries covers a multiple-table `UPDATE` statement. Although correlated `UPDATE` statements work almost exactly alike in both Oracle and MySQL, SQL functions differ between the two databases.

Oracle Correlated Queries

A subquery that matches a value directly works as the right operand in a `SET` clause assignment. Sometimes, finding the value requires matching column values from other tables against the rows of the table you’re updating. For example, when you need to identify a rental item’s price on a rental agreement and an inventory item, you would need to cross join the `rental` and `item` tables before filtering the results through joins to the `rental_item` table. At least, that’s what you need to do in the book’s sample application, because the `rental_item` table is an association table. It resolves a logical many-to-many relationship between the `rental` and `item` tables. In the `rental_item` association table, some columns belong to the relationship between the rental and item tables, such as prices associated with specific rental item and rental agreements.

A correlated UPDATE statement lets SQL find all potential matches and update all affected prices in the `rental_item` table. Basically, the computer sorts out all the relationships shared through the `rental_item` association table.

The following example illustrates the correlated UPDATE statement that accomplishes this:

```
UPDATE rental_item ri
SET    ri.rental_item_price =
      (SELECT p.amount
       FROM price p CROSS JOIN rental r
       WHERE p.item_id = ri.item_id
       AND   p.price_type = ri.rental_item_type
       AND   r.rental_id = ri.rental_id
       AND   r.check_out_date
              BETWEEN p.start_date
              AND NVL(p.end_date, TRUNC(SYSDATE)) );
```

Correlation between subquery and UPDATE statement.

The `CROSS JOIN` result set for the `price` and `rental` tables is every row in one table matched against every row in the other table. A date range filter finds all rows between a `start_date` and `end_date`, or the current date. The remaining rows in the result set are then matched through an inner join to the table being updated. The range join operation (technically a range non-equijoin) works because a current date value is substituted for a possible null `end_date` value.



NOTE

Most data models that use time-events to bracket unique sets leave the current value with an open end date. The implementation of null as the open end date is the more common modeling solution.

TIP

The `TRUNC` function shaves the fractional time component off a date-time data type and converts the value to a date-time of midnight. There is no pure date data type in an Oracle database.

A correlated UPDATE statement such as this changes the values of multiple rows. No `WHERE` clause is required in this type of update, because the inner join finds only the rows that should be changed. Inner joins are equijoin—a join based on the equality of values from two columns or two sets of columns (see Chapter 11 for more on joins).

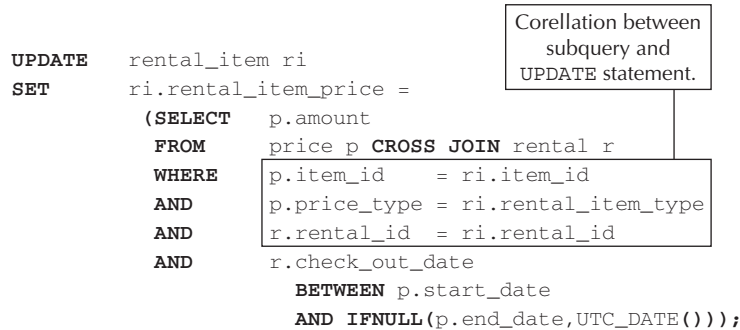
MySQL Correlated Queries

MySQL supports correlated UPDATE statements similar to Oracle, except you can update a single column only through the `SET` clause with a specific correlated UPDATE statement. MySQL also supports multiple-table UPDATE statements that act like correlated queries.

Building on the example from the Oracle section, only one change is required for this to work in MySQL. The Oracle proprietary `NVL` function must be replaced by the MySQL proprietary

IFNULL function. Both perform the same task: they replace a null value with a default value. Since TRUNC and SYSDATE are also proprietary to an Oracle database, you use the UTC_DATE function in MySQL. The NOW function would also work but raises a warning message when it's implicitly cast to a date, which is a loss of precision from the MySQL Server's perspective.

An example of the correlated UPDATE statement for MySQL is shown in the following illustration:



```
UPDATE rental_item ri
SET ri.rental_item_price =
    (SELECT p.amount
     FROM price p CROSS JOIN rental r
     WHERE p.item_id = ri.item_id
     AND p.price_type = ri.rental_item_type
     AND r.rental_id = ri.rental_id
     AND r.check_out_date
        BETWEEN p.start_date
        AND IFNULL(p.end_date, UTC_DATE()));
```

Similar to the explanation in the Oracle discussion, this query resolves through the three column joins between the updated `rental_item` table and columns from the cross joined product of the `price` and `rental` tables. This join updates only the `rental_item` table rows that match the criteria in the subquery, which is filtered by *rentals* that fall within a certain *price date range*.

The next section shows the multiple-table UPDATE statement available in MySQL. Multiple-table UPDATE statements are not portable to other database servers.

Multiple-Table Update Statement

The multiple-table UPDATE statement uses ANSI SQL-92 syntax for joins, which uses keywords such as JOIN and CROSS JOIN. A JOIN is shorthand notation for an INNER JOIN. Inner joins add the rows from one table to another table where one or a set of columns hold values that match. Chapter 11 covers join syntax in more detail.

Here's the prototype for a multiple-table UPDATE statement:

```
UPDATE [LOW_PRIORITY] [IGNORE] target_table_name table_alias
[{{INNER | LEFT | RIGHT} JOIN table_name table_alias
  {ON table_alias.column_name = table_alias.column_name
   [AND {table_alias.column_name = table_alias.column_name |
        table_alias.column_name BETWEEN table_alias.column_name
        AND table_alias.column_name} |
   USING(column_name [, column_name [, ...]])} [...]]
SET column_name = expression [, ...]
WHERE [NOT] {column_name | expression} = {expression | DEFAULT}
[{{AND | OR} [NOT] {column_name | expression} = {expression | DEFAULT} [...]]];
```

The multiple-table UPDATE statement prototype supports LOW PRIORITY and IGNORE like the other UPDATE statements. Table aliases should be used for clarity, and they're supported. When you opt not to use table aliases, be sure to use table names to qualify columns or you'll raise an ambiguous column error.

You can perform INNER, LEFT, or RIGHT JOIN operations in multiple-table UPDATE statements, and you can check Chapter 11 for more on join mechanics. It is possible to have joins to multiple tables, and you can also use the BETWEEN operator to filter cross join result sets. The SET and WHERE clauses work the same as in other UPDATE statements.

Refactoring the preceding correlated subquery, it can be written as the following multiple-table UPDATE statement:

```
UPDATE rental_item ri
JOIN rental r ON ri.rental_id = r.rental_id
JOIN price p ON ri.item_id = p.item_id
              AND ri.rental_item_type = p.price_type
              AND r.check_out_date BETWEEN
                  p.start_date AND IFNULL(p.end_date, UTC_DATE())
SET ri.rental_item_price = p.amount;
```

UPDATE statements typically work with tables. This changes with multiple-table UPDATE statements, because a series of tables are joined together to create a temporary result set. This is done by a *partial* inner join between the rental_item table and both the rental and price tables. The completion of the join between the rental_item and price table includes a range filter that finds rows based on the occurrence of a checkout date between a start and end date for a row in the price table. The IFNULL function guarantees that the upward range element isn't a null value, which would cause the statement to fail.

The join syntax for a multiple-table UPDATE statement is what you would see in a SELECT statement when querying data. For this reason alone, it is probably more widely used than a correlated statement. As mentioned, multiple-table UPDATE statements aren't portable to other databases. I'd recommend that you use it to sort out what should go into a correlated UPDATE statement and then write the correlated UPDATE statement for portability's sake.

Summary

This chapter covered methods for updating data. The UPDATE statement lets you change column values by direct assignment of literal values for one to many rows of a table. If you exclude a filtering WHERE clause from an UPDATE statement, all rows in the table are updated. The UPDATE statement also lets you correlate against rows in other tables. MySQL lets you perform multiple-table UPDATE statements, which mirror correlated update statements but have syntax more similar to that found in SELECT statements.

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 ☐ An UPDATE statement supports multiple row changes with only one set of date, numeric, or string literals.
2. True ☐ False ☐ An UPDATE statement supports DEFAULT values in Oracle databases only.

3. **True** ☐ **False** ☐ The `SET` clause must contain a list of all mandatory columns.
4. **True** ☐ **False** ☐ A multiple-table `UPDATE` statement in MySQL performs as a correlated `UPDATE` statement.
5. **True** ☐ **False** ☐ MySQL supports a nested table `UPDATE` statement.
6. **True** ☐ **False** ☐ A `BLOB` or `CLOB` requires the `RETURNING INTO` phrase to work with procedural programming modules.
7. **True** ☐ **False** ☐ The `IGNORE` keyword lets an Oracle or MySQL `UPDATE` statement run successfully even when it encounters errors.
8. **True** ☐ **False** ☐ A `LOW_PRIORITY` assignment lets an `UPDATE` statement run behind other DML statements in the database.
9. **True** ☐ **False** ☐ MySQL supports the `RETURNING INTO` phrase in `UPDATE` statements.
10. **True** ☐ **False** ☐ A multiple column and row `UPDATE` is possible in either Oracle or MySQL databases.
11. Which of the following data types requires a `RETURNING INTO` phrase to interact with procedural programs (multiple answers possible)?
 - A. `BLOB`
 - B. `MEDIUMCLOB`
 - C. `TEXT`
 - D. `CLOB`
 - E. `VARCHAR`
12. In an Oracle database, what are the maximum number of rows that you can update through a `SET` clause when values are assigned by literals?
 - A. 1
 - B. 2
 - C. 3
 - D. 4
 - E. Many
13. In a MySQL database, what are the maximum number of rows that you can update through a `SET` clause when values are assigned by literals?
 - A. 1
 - B. 2
 - C. 3
 - D. 4
 - E. Many

14. When you use a correlated query instead of a `SET` clause in Oracle or MySQL, what filters the rows updated?
- A. The `WHERE` clause
 - B. The `SET` clause
 - C. The joins between the updated table and correlated subquery
 - D. The `ORDER BY` clause
 - E. The `LIMIT` clause
15. What can't you change with an `UPDATE` statement in an Oracle database?
- A. A collection of a UDT
 - B. An element of a UDT
 - C. A collection of a scalar data type
 - D. An element of a scalar data type
 - E. None of the above

