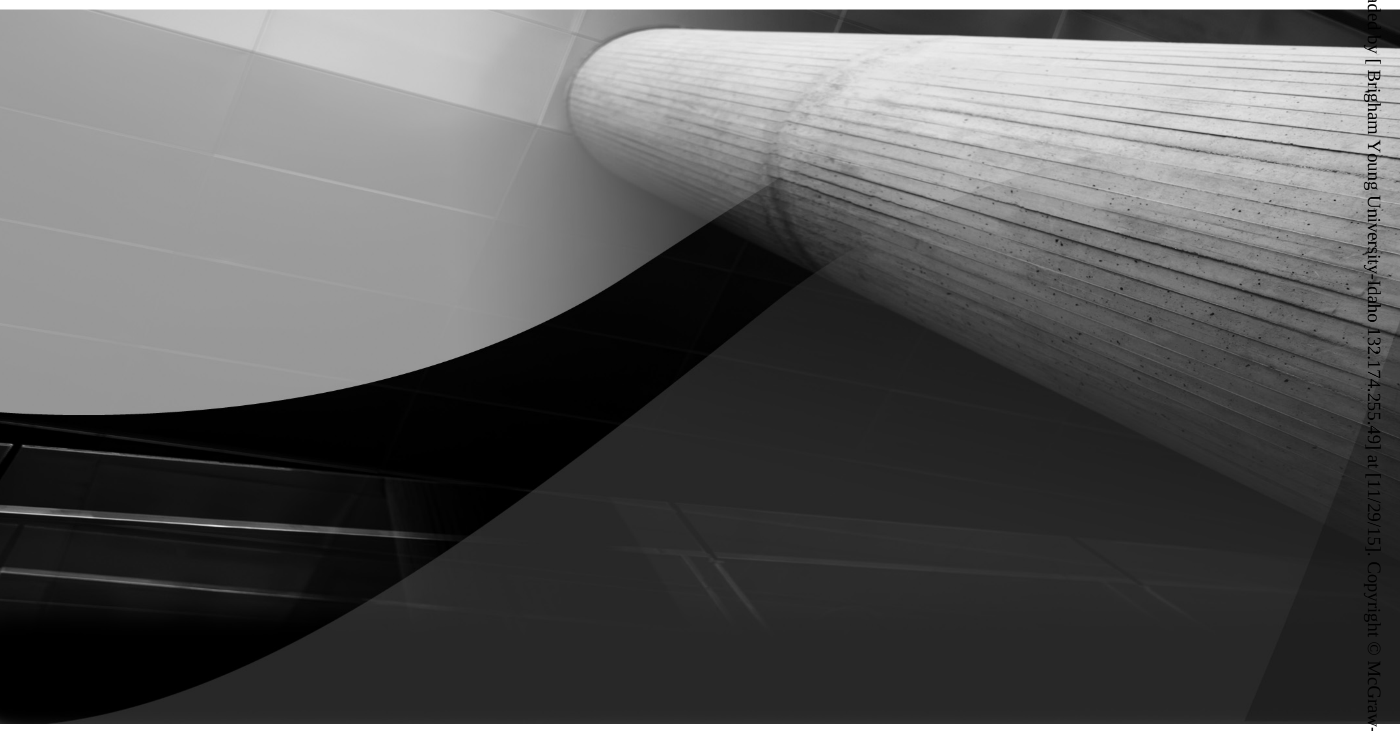

CHAPTER 8

Inserting Data

The INSERT statement lets you enter data into tables and views in two ways: via an INSERT statement with a VALUES (and in MySQL with a VALUES or SET) clause and via an INSERT statement with a query. The VALUES clause takes a list of literal (strings, numbers, and dates represented as strings), expressions (return values from functions), or variable values.

Query values are results from SELECT statements that are subqueries (covered in Chapter 11). INSERT statements work with scalar, single-row, and multiple-row subqueries. The list of columns in the VALUES clause or SELECT clause of a query (a SELECT list) must map to the positional list of columns that defines the table. That list is found in the data dictionary or catalog. Alternatively to the list of columns from the data catalog, you can provide a named list of those columns. The named list *overrides* the positional (or *default*) order from the data catalog and must provide at least all mandatory columns in the table definition. Mandatory columns are those that are not null constrained.

MySQL and Oracle databases differ in how they implement the INSERT statement. MySQL supports multiple row inserts with a VALUES clause; Oracle does not support this. Oracle and MySQL both support default and override signatures as qualified in the ANSI SQL standards. Oracle also provides a multiple-table INSERT statement, which MySQL does not provide. This chapter covers how you enter data with an INSERT statement that is based on a VALUES clause or a subquery result statement, as well as multiple-table INSERT statements.

The INSERT statement has one significant limitation: its *default signature*. The default signature is the list of columns that define the table in the data catalog. The list is defined by the position and data type of columns. The CREATE statement defines the initial default signature, and the ALTER statement can change the number, data types, or ordering of columns in the default signature.

The default prototype for an INSERT statement allows for an optional column list that overrides the default list of columns. Like methods in object-oriented programming languages (OOPLs), an INSERT statement without the optional column list constructs an instance (or row) of the table using the default constructor. The override constructor for a row is defined by any INSERT statement when you provide an optional column list. That's because it overrides the default constructor.

The generic prototype for an INSERT statement is confusing when it tries to capture both the VALUES clause and the result set from a query. Therefore, I've opted to provide two generic prototypes. The first uses the VALUES clause:

```
INSERT
INTO [{LOW_PRIORITY | DELAYED | HIGH_PRIORITY}] [IGNORE] table_name
[( column1, column2, column3, ...)]
VALUES
( value1, value2, value3, ...)
[ON DUPLICATE KEY
 UPDATE column_name1 = expression1
 [,     column_name2 = expression2
 [, ...]]];
```

MySQL also supports multiple rows in the VALUES clause, which supports this modification to the VALUES clause:

```
VALUES
 ( value1A, value2A, value3A, ...)
,( value1B, value2B, value3B, ...);
```

Overriding vs. Overloading

OOPLs have special vocabularies. You define a class in OOPLs by writing a program that outlines the rules for how to create an instance of a class and the methods available for any instance of that class.

Objects have special methods known as constructors that let you create instances. The default constructor generally has no formal parameters. When writing the code for a class, you can define constructors that override the default constructor's behavior. These user-defined constructors are known as *overriding* constructors. Objects also have methods that perform actions against the instance, and some of these methods are overloaded, which means a given method name supports different lists of formal parameters. This is known as *method overloading*.

The list of parameters in an overriding constructor is also known as the *overriding signature*, defined as something that serves to set apart or identify. The same logic makes the formal parameter list of any method a signature of that method.

The class is also known as an object *type*, which is a data type. A table in a database is an object type, because it contains a definition of what it can include. Every row is an instance of that object type.

As qualified in Chapter 1, SQL is a set-based declarative language. Declarative languages hide the implementation details while providing a means for the developer to state what should happen. The default signature to enter a row of data is read from the data catalog and compared against the list of values in a `VALUES` clause or query. The `INSERT` statement's optional column list lets you override the default signature to enter a row of data, but the database checks to ensure that you conform to any not null column-level constraints.

MySQL supports another less frequently used syntax that resembles the `UPDATE` statement. It's a named assignment prototype with a `SET` clause:

```
INSERT
INTO [{LOW_PRIORITY | DELAYED | HIGH_PRIORITY}] [IGNORE] table_name
SET column_name1 = expression1
[,  column_name2 = expression2
[, ...]]
[ON DUPLICATE KEY
 UPDATE column_name1 = expression2
 [,     column_name2 = expression2
 [, ...]]];
```

The prototype of an `INSERT` statement based on a result set from a query is shown next:

```
INSERT
INTO [{LOW_PRIORITY | DELAYED | HIGH_PRIORITY}] [IGNORE] table_name
[( column1, column2, column3, ...)]
( SELECT value1, value2, value3, ...
  FROM   some_table
  WHERE  some_column = some_value )
[ON DUPLICATE KEY
```

```
 UPDATE column_name1 = expression1
 [,     column_name2 = expression2
 [, ...]]];
```

Notice that the prototype for an `INSERT` statement with the result set from a query doesn't use the `VALUES` clause at all. A parsing error occurs when the `VALUES` clause and query both occur in an `INSERT` statement.

Default signatures present a risk of data corruption through insertion anomalies, which occur when you enter bad data in tables. Mistakes transposing or misplacing values can occur more frequently with a default signature, because the underlying table structure can change. As a best practice, always use named notation by providing the optional list of values; this should help you avoid putting the right data in the wrong place.

TIP
Inserts should always rely on named notation to help you avoid adding data in the wrong columns.

This chapter provides examples that use the default and override syntax for `INSERT` statements in both Oracle and MySQL. The Oracle sections also support multiple-table `INSERT` statements and a `RETURING INTO` clause, which is an extension of the ANSI SQL standard. Oracle uses the `RETURNING INTO` clause to manage large objects and some of the features of Oracle's dynamic SQL. Note that Oracle also supports a bulk `INSERT` statement, which is covered in Chapter 13 because it requires knowledge of PL/SQL.

Insert by Values

Inserting by the `VALUES` clause is the most common type of `INSERT` statement. It's most useful when interacting with single row inserts. You typically use this type of `INSERT` statement when working with data entered through end user web forms. In some cases, users can enter more than one row of data using a form, which occurs, for example, when a user places a meal order in a restaurant and the meal and drink are treated as order items. The restaurant order entry system would enter a single row in the `order` table and two rows in the `order_item` table (one for the meal and the other for the drink). How the code enters the multiple order items can differ between an Oracle and MySQL database, because MySQL supports multiple items in the `VALUES` clause, while Oracle doesn't. For example, an Oracle implementation might open a loop and process two dynamic `INSERT` statements (see Chapter 13), while a MySQL implementation might create, prepare, execute, and de-allocate a prepared statement that inserts two rows in a single `INSERT` statement (see Chapter 14).

The next two subsections explain how to use `INSERT` statements with the `VALUES` clause in an Oracle or MySQL database.

Oracle Insert by Values

Oracle supports only a single row insert through the `VALUES` clause. Multiple row inserts require an `INSERT` statement from a query. This section covers single row `INSERT` statements and the "Oracle Insert by Query" section covers multiple row inserts.

The `VALUES` clause of an `INSERT` statement accepts scalar values, such as strings, numbers, and dates. It also accepts calls to arrays, lists, or user-defined object types, which are called *flattened objects*. Oracle supports VARRAY as arrays and nested tables as lists. They can both contain elements of a scalar data type or user-defined object type. Chapter 6 shows you how to create these data types.

The following sections discuss how you use the VALUES clause with scalar data types, how you convert various data types, and how you use the VALUES clause with nested tables and user-defined object data types.

Inserting Scalar Data Types

The basic syntax for an INSERT statement with a VALUES clause can include an optional *override signature* between the table name and VALUES keyword. With an override signature, you designate the column names and the order of entry for the VALUES clause elements. Without an override signature, the INSERT signature checks the definition of the table in the database catalog. The positional order of the column in the data catalog defines the positional, or default, signature for the INSERT statement. You can discover the structure of a table in Oracle or MySQL with the DESCRIBE command issued at the SQL*Plus or MySQL Monitor command line:

```
DESCRIBE table_name;
```

The semicolon is unnecessary in Oracle but required by the MySQL Monitor. Semicolons are execution commands and are covered in Chapter 2.

You'll see the following after describing the rental table in SQL*Plus:

```
 Name                                      Null?    Type
 ----------------------------------------- -------- --------
 RENTAL_ID                                 NOT NULL NUMBER
 CUSTOMER_ID                               NOT NULL NUMBER
 CHECK_OUT_DATE                            NOT NULL DATE
 RETURN_DATE                                        DATE
 CREATED_BY                                NOT NULL NUMBER
 CREATION_DATE                             NOT NULL DATE
 LAST_UPDATED_BY                           NOT NULL NUMBER
 LAST_UPDATE_DATE                          NOT NULL DATE
```

The rental_id column is a *surrogate key*, or an artificial numbering sequence. The combination of the customer_id and check_out_date serves as a *natural key*, because a DATE data type is a date-time value. If it were only a date, the customer would be limited to a single entry for each day, and limiting customer rentals to one per day isn't a good business model.

The basic INSERT statement would require that you look up the next sequence value before using it. You should also look up the surrogate key column value that maps to the row where your unique customer is stored in the contact table. For this example, assume the following facts:

- Next sequence value is 1086
- Customer's surrogate key value is 1009
- Current date-time is represented by the value from the SYSDATE function
- Return date is the fifth date from today
- User adding and updating the row has a primary (surrogate) key value of 1
- Creation and last update date are the value returned from the SYSDATE function.

An `INSERT` statement must include a list of values that match the positional data types of the database catalog, or it must use an override signature for all mandatory columns.

You can now write the following `INSERT` statement, which relies on the default signature:

```
SQL> INSERT INTO rental
  2  VALUES
  3  ( 1086
  4  , 1009
  5  , SYSDATE
  6  , TRUNC(SYSDATE + 5)
  7  , 1
  8  , SYSDATE
  9  , 1
 10  , SYSDATE);
```

If you weren't using `SYSDATE` for the date-time value on line 5, you could manually enter a date-time with the following Oracle proprietary syntax:

```
  5  , TO_DATE('15-APR-2011 12:53:01','DD-MON-YYYY HH24:MI:SS')
```

The `TO_DATE` function is an Oracle-specific function. The generic conversion function would be the `CAST` function. The problem with a `CAST` function by itself is that it can't handle a format mask other than the database defaults (`'DD-MON-RR'` or `'DD-MON-YYYY'`)—for example, consider this syntax:

```
  5  ,  CAST('15-APR-2011 12:53:02' AS DATE)
```

It raises the following error:

```
  5  ,  CAST('15-APR-2011 12:53:02' AS DATE) FROM dual
        *
ERROR at line 1:
ORA-01830: date format picture ends before converting entire input string
```

You actually need to double cast this type of format mask when you want to store it as a `DATE` data type. The working syntax casts the date-time string as a `TIMESTAMP` data type before recasting the `TIMESTAMP` to a `DATE`, like so:

```
  5  ,  CAST(CAST('15-APR-2011 12:53:02' AS TIMESTAMP) AS DATE)
```

Before you could have written the preceding `INSERT` statement, you would need to run some queries to find the values. You would secure the next value from a `rental_s1` sequence in an Oracle database with the following command:

```
SQL> SELECT   rental_s1.NEXTVAL FROM dual;
```

This assumes two things, because sequences are separate objects from tables. First, code from which the values in a table's surrogate key column come must appear in the correct sequence. Second, a sequence value is inserted only once into a table as a primary key value.

In place of a query that finds the next sequence value, you would simply use a call against the `.NEXTVAL` pseudo column in the `VALUES` clause. You would replace line 3 with this:

```
  3  ( rental_s1.NEXTVAL
```

The `.NEXTVAL` is a pseudo column, and it instantiates an instance of a sequence in the current session. After a call to a sequence with the `.NEXTVAL` pseudo column, you can also call back the prior sequence value with the `.CURRVAL` pseudo column.

NOTE
Sequences are separate objects from tables, and your code ensures that only the appropriate sequence maps to the correct table.

Assuming the following query would return a single row, you can use the `contact_id` value as the `customer_id` value in the `rental` table:

```
SQL> SELECT   contact_id
  2  FROM     contact
  3  WHERE    last_name = 'Potter'
  4  AND      first_name = 'Harry';
```

Taking three steps like this is unnecessary, however, because you can call the next sequence value and find the valid `customer_id` value inside the `VALUES` clause of the `INSERT` statement. The following `INSERT` statement uses an override signature and calls for the next sequence value on line 11. It also uses a scalar subquery to look up the correct `customer_id` value with a scalar subquery on lines 12 through 15.

```
SQL> INSERT INTO rental
  2  ( rental_id
  3  , customer_id
  4  , check_out_date
  5  , return_date
  6  , created_by
  7  , creation_date
  8  , last_updated_by
  9  , last_update_date )
 10  VALUES
 11  ( rental_s1.nextval
 12  ,(SELECT   contact_id
 13    FROM     contact
 14    WHERE    last_name = 'Potter'
 15    AND      first_name = 'Harry')
 16  , SYSDATE
 17  , TRUNC(SYSDATE + 5)
 18  , 1
 19  , SYSDATE
 20  , 3
 21  , SYSDATE);
```

When a subquery returns two or more rows because the conditions in the `WHERE` clause failed to find and return a unique row, the `INSERT` statement would fail with the following message:

```
,(SELECT   contact_id
  *
ERROR at line 3:
ORA-01427: single-row subquery returns more than one row
```

In fact, the statement could fail when two or more "Harry Potter" names exist in the data set, because three columns make up the natural key of the contact table. The third column is the `member_id`, and all three should be qualified inside a scalar subquery to guarantee that it returns only one row of data.

Handling Oracle's Large Objects Oracle's large objects present a small problem when they're not null constrained in the table definition. In this case, you must insert empty object containers or references when you perform an `INSERT` statement.

Assume, for example, that you have the following three large object columns in a table:

```
 Name                                Null?    Type
 ------------------------------- -------- ----------------------
 ITEM_DESC                         NOT NULL CLOB
 ITEM_ICON                         NOT NULL BLOB
 ITEM_PHOTO                                 BINARY FILE LOB
```

The `item_desc` column uses a `CLOB` (Character Large Object) data type, and it is a required column; it could hold a lengthy description of a movie, for example. The `item_icon` column uses a `BLOB` (Binary Large Object) data type, and it is also a required column. It could hold a graphic image. The `item_photo` column uses a binary file (an externally managed file). It could hold a null or a reference to an external graphic image; fortunately, the `item_photo` column isn't a required column.

Oracle provides two functions that let you enter an empty large object:

```
empty_blob()
empty_clob()
```

Although you could insert a null value in the `item_photo` column, you can also enter a reference to an Oracle database virtual directory file. Here's the syntax to enter a valid `BFILE` name with the `BFILENAME` function call:

```
 10   , BFILENAME('VIRTUAL_DIRECTORY_NAME', 'file_name.png')
```

You can insert a large character or binary stream into `BLOB` and `CLOB` data types by using the stored procedures and functions available in the `DBMS_LOB` package. These operations require a working knowledge of PL/SQL programming, which isn't covered until Chapter 13, where you'll see an example that inserts a large `CLOB` column.

You can use an `EMPTY_CLOB` function or a string literal up to 32,767 bytes long in a `VALUES` clause. You must use the `DBMS_LOB` package when you insert a string that is longer than 32,767 bytes. That also changes the nature of the `INSERT` statement and requires that you append the `RETURNING INTO` clause. Here's the prototype for this Oracle proprietary syntax:

```
INSERT INTO some_table
[( column1, column2, column3, ...)]
VALUES
( value1, value2, value3, ...)
RETURNING column1 INTO local_variable;
```

The `local_variable` is a reference to a procedural programming language. It lets you insert a character stream into a target `CLOB` column or a binary stream into a `BLOB` column.

DBA Heads-up on Large Object Storage

CLOB and BLOB columns are stored with the rest of a row when they're smaller than 4000 bytes. Larger versions are stored out-of-line, which means they're placed in a contiguous space that is away from the rest of the related row.

DBAs often designate a special tablespace for the storage clauses of BLOB and CLOB columns. This extra step is beneficial, because large objects change less frequently and consume a lot of storage. They're also generally on different backup schedules than other transactional columns in a table.

Capturing the Last Sequence Value Sometimes you insert into a series of tables in the scope of a transaction, like those described in Chapter 4. In this scenario, one table gets the new sequence value (with a call to `sequence_name.NEXTVAL`) and enters it as the surrogate primary key, and another table needs a copy of that primary key to enter into a foreign key column. While scalar subqueries can solve this problem, Oracle provides the `.CURRVAL` pseudo column for this purpose.

The steps to demonstrate this behavior require a `parent` and `child` table. The `parent` table is defined as follows:

```
Name                                    Null?    Type
--------------------------------------- -------- --------------
PARENT_ID                               NOT NULL NUMBER
PARENT_NAME                                      VARCHAR2(10)
```

The `parent_id` column is the primary key for the parent table. You include the `parent_id` column in the child table. In the child table, the `parent_id` column holds a copy of a valid primary key column value as a foreign key to the parent table.

```
Name                                    Null?    Type
--------------------------------------- -------- --------------
CHILD_ID                                NOT NULL NUMBER
PARENT_ID                                        NUMBER
PARENT_NAME                                      VARCHAR2(10)
```

After creating the two tables, you can manage inserts into them with the `.NEXTVAL` and `.CURRVAL` pseudo columns. The sequence calls with the `.NEXTVAL` insert primary key values, and the sequence calls with the `.CURRVAL` insert foreign key values.

You would perform these two `INSERT` statements as a group:

```
SQL> INSERT INTO parent
  2  VALUES
  3  ( parent_s1.NEXTVAL
  4  ,'One Parent');

SQL> INSERT INTO child
  2  VALUES
  3  ( child_s1.NEXTVAL
  4  , parent_s1.CURRVAL
  5  ,'One Child');
```

The `.CURRVAL` pseudo column for any sequence fetches the value placed in memory by call to the `.NEXTVAL` pseudo column. Any attempt to call the `.CURRVAL` pseudo column before the `.NEXTVAL` pseudo column raises an ORA-02289 exception. The text message for that error says the sequence doesn't exist, which actually means that it doesn't exist in the scope of the current session. Line 4 in the insert into the `child` table depends on line 3 in the insert into the `parent` table.

You can use comments in `INSERT` statements to map to columns in the table. For example, the following shows the technique for the `child` table from the preceding example:

```
SQL> INSERT INTO child
  2  VALUES
  3  ( child_s1.NEXTVAL     -- CHILD_ID
  4  , parent_s1.CURRVAL    -- PARENT_ID
  5  ,'One Child')          -- CHILD_NAME
  6  /
```

Comments on the lines of the `VALUES` clause identify the columns where the values are inserted. A semicolon doesn't execute this statement, because a trailing comment would trigger a runtime exception. You must use the semicolon or forward slash on the line below the last `VALUES` element to include the last comment.

TIP
A comment on the last line of any statement requires that you exclude the semicolon and place it or a forward slash on the next line.

Data Type Conversions

Oracle supports a series of conversion functions that let you convert data types from one type to another. The generic SQL conversion function is `CAST`, which lets you convert the following data types.

Convert from BINARY_FLOAT or BINARY_DOUBLE Data Type to `BINARY_FLOAT`, `BINARY_DOUBLE`, `CHAR`, `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `NCHAR`, `NVARCHAR`

Convert from CHAR or VARCHAR2 Data Type to `BINARY_FLOAT`, `BINARY_DOUBLE`, `CHAR`, `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `DATE`, `TIMESTAMP`, `INTERVAL`, `RAW`, `ROWID`, `UROWID`, `NCHAR`, `NVARCHAR`

Here's an example of converting a string literal date into a timestamp:

```
CAST('14-FEB-2011' AS TIMESTAMP WITH LOCAL TIME ZONE)
```

This example works because the date literal conforms to the default format mask for a date in an Oracle database. A nonconforming date literal would raise a conversion error. Many possibilities are available because you can organize the valid elements of dates many ways. A nonconforming date literal should be converted by using the `TO_DATE` or `TO_TIMESTAMP` function, because each of these lets you specify an overriding date format mask value, such as this conversion to a `DATE` data type:

```
TO_DATE('2011-02-14', 'YYYY-MM-DD')
```

Or this conversion to a `TIMESTAMP` data type:

```
TO_TIMESTAMP('2011-02-14 18:11:28.1500', 'YYYY-MM-DD HH24:MI:SS.FF')
```

Converting to an `INTERVAL` data type is covered in the next section, because you first must extract a time property as a number. It's also possible that implicit casting of a numeric string can change the base data type to an integer for you. The method of implicit or explicit conversion depends on how you get the initial data value.

Convert from NUMBER Data Type to `BINARY_FLOAT`, `BINARY_DOUBLE`, `CHAR`, `VARCHAR2`, `NUMBER`, `DATE`, `TIMESTAMP`, `NCHAR`, `NVARCHAR`

Interval conversions are a bit more complex, because you need more than one function to convert them. Typically, you pull the value from a `DATE` or `TIMESTAMP` data type and `EXTRACT` the element of time by identifying its type before converting that value to an `INTERVAL` type. The following provides an example:

```
NUMTODSINTERVAL(EXTRACT(MINUTE FROM some_date), 'MINUTE')
```

You will use this type of built-in function layering frequently in some situations. It's always a better approach to understand and use the built-in functions before you write your own stored functions.

Convert from DATETIME or INTERVAL Data Type to `CHAR`, `VARCHAR2`, `DATE`, `TIMESTAMP`, `DATE`, `TIMESTAMP`, `INTERVAL`, `NCHAR`, `NVARCHAR`

Convert from RAW Data Type to `CHAR`, `VARCHAR2`, `RAW`, `NCHAR`, `NVARCHAR`

Convert from ROWID or UROWID Data Type to `CHAR`, `VARCHAR2`, `ROWID`, `UROWID`, `NCHAR`, `NVARCHAR`

NOTE
You cannot cast a `UROWID` to a `ROWID` in the `UROWID` of an index organized table.

Convert from NCHAR or NVARCHAR2 Data Type to `BINARY_FLOAT`, `BINARY_DOUBLE`, `NUMBER`, `NCHAR`, `NVARCHAR`

Inserting Arrays and Nested Tables

The ability to insert arrays and nested tables in an Oracle database is an important feature made possible by the *object-relational* technology of the database. You can access these embedded structures only through the containing table, which makes them much like inner classes in object-oriented programming. From a database modeling perspective, they're ID-dependent data sets, because the only relationship is through the row of the containing table.

You can walk through a simple design and development exercise by creating a collection of a scalar data type, a table that contains the data type, and then an `INSERT` statement to populate the table with data. You create the user-defined collection data type by using this syntax:

```
CREATE TYPE number_array IS VARRAY(10) OF NUMBER;
```

After you have the user-defined collection type, create a table that uses it and a sequence for an automatic numbering column, like so:

```
SQL> CREATE TABLE sample_nester
  2  ( nester_id    NUMBER
  3  , array_column NUMBER_ARRAY);
SQL> CREATE SEQUENCE sample_nester_s1;
```

You enter values into the table by calling a collection constructor. Calls are made to the data type name, not the column name, like so:

```
SQL> INSERT INTO sample_nester
  2  VALUES
  3  ( sample_nester_s1.NEXTVAL
  4  , NUMBER_ARRAY(0,1,2,3,4,5,6,7,8,9));
```

Here are formatting instructions and an ordinary query against this table:

```
SQL> COLUMN ARRAY_COLUMN FORMAT A44
SQL> SELECT * from sample_nester;
```

which returns the following:

```
 NESTER_ID ARRAY_COLUMN
---------- --------------------------------------------
         1 NUMBER_ARRAY(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

The value in the `ARRAY_COLUMN` is a call to the user-defined collection data type's constructor. This collection is a simple array of ten numbers. You can join the `nester_id` column against all ten elements in the collection with the following syntax:

```
SQL> SELECT nester_id
  2  ,      collection.column_value
  3  FROM   sample_nester CROSS JOIN TABLE(array_column) collection;
```

The `TABLE` function extracts the element of the collection into a SQL result set, which can then be treated like a normal set of rows from any table. Oracle always returns collections of base scalar data types into a `COLUMN_VALUE` column. The results from the sample query follow:

```
 NESTER_ID COLUMN_VALUE
---------- ------------
         1            0
         1            1
         1            2
 ...
         1            8
         1            9
```

Here's a multilevel `INSERT` statement based on the employee table from Chapter 6. The following inserts a single row that contains an `address_list` collection of two instances of the `address_type` user-defined object type, which in turn holds a collection of a `street_list` nested table of variable-length strings:

```
SQL> INSERT INTO employee
  2  ( employee_id
  3  , first_name
  4  , last_name
  5  , home_address )
  6  VALUES
  7  ( employee_s1.NEXTVAL
  8  ,'Sam'
  9  ,'Yosemite'
 10  , address_list(
 11      address_type( 1
 12                  , street_list('1111 Broadway','Suite 322')
 13                  ,'Oakland'
 14                  ,'CA'
 15                  ,'94612')
 16    , address_type( 2
 17                  , street_list('1111 Broadway','Suite 525')
 18                  ,'Oakland'
 19                  ,'CA'
 20                  ,'94612')));
```

Lines 10 through 20 insert a nested table (list) of two `address_type` instances. The sequence numbers are manually entered because this type of design would always start elements of a nested table with a sequence value of 1. This is an implementation of an ID-dependent relationship. The nested table is accessible only through the row of a table, and as such, it acts only when connected with the containing row.

NOTE
Nested tables are complex to access and model. As mentioned in Chapter 6, they also create type chaining, which presents maintenance headaches during major software releases. If you opt to use them, you should have a good reason for adding the complexity.

Multiple-Table Insert Statements

As mentioned, Oracle SQL syntax lets you perform multiple-table inserts with the `INSERT` statement. A multiple-table `INSERT` statement can be useful when you receive an import source file that belongs in more than one table. You should be aware of some caveats with this type of multiple-table insert. For example, a one-to-one mapping must exist between all the data in the same row. When you have a one-to-many relationship between columns in the single import source table, the `MERGE` statement is a better solution, as discussed in Chapter 12.

Here's the prototype for the multiple table `INSERT` statement:

```
INSERT {ALL | FIRST}
 [WHEN comparison_clause THEN ]
  INSERT INTO table_name_one
  ( column_list )
  VALUES
  ( value_list )
[[WHEN comparison_clause THEN ]
  INSERT INTO table_name_two
  ( column_list )
  VALUES
  ( value_list ) ]
 [ ... ]
 [ELSE
  INSERT INTO table_name_else
  ( column_list )
  VALUES
  ( value_list ) ]
 query_statement;
```

A multiple-table insert can be performed in three ways: One uses the `ALL` keyword but excludes the `WHEN` clauses. The second uses the `ALL` keyword while including the `WHEN` clauses. The last uses a `FIRST` keyword instead of the `ALL` keyword and inserts to the first table before moving to the second, and so forth.

The multiple-table `INSERT` statement variations are discussed in the following subsections, but they're all supported by the following three tables and sequences.

The `rank_index` is the first table, and it contains a string for the military service, such as Army, Navy, Marines, and Air Force, with their accompanying abbreviated and full-titled ranks.

```
-- Create the rank_index table and sequence.
CREATE TABLE rank_index
( rank_id         NUMBER
, rank_service    VARCHAR2(10)
, rank_short_name VARCHAR2(4)
, rank_full_name  VARCHAR2(30));
CREATE SEQUENCE rank_index_s;
```

The `soldier` and `sailor` tables are target tables for the inserts from the multiple-table `INSERT` statement. They contain the rank and name of soldiers and sailors, respectively:

```
-- Create the soldier table and sequence.
CREATE TABLE soldier
( soldier_id    NUMBER
, soldier_rank  VARCHAR2(4)
, soldier_name  VARCHAR2(20));
CREATE SEQUENCE soldier_s;

-- Create the sailor table and sequence.
CREATE TABLE sailor
( sailor_id    NUMBER
, sailor_rank  VARCHAR2(4)
, sailor_name  VARCHAR2(20));
CREATE SEQUENCE sailor_s;
```

All the examples get their data from a `SELECT` statement that uses the dual pseudo table and a fabricated result set. The `INSERT ALL` or `INSERT FIRST` statement inserts the data into one or both of the target tables. The tables are unconstrained because constraints aren't required for the examples.

Multiple-Table INSERT ALL Without WHEN Clauses Our first example shows you how to insert data into multiple tables without any qualifications. Values from each row returned by the query are inserted into the `soldier` and `sailor` tables. Unfiltered multiple-table `INSERT` statements put rows into all tables referenced by an `INTO` clause. In this regard, the `INSERT ALL` statement works like a *switch* statement with fall through in C#, C++, or Java. (*Fall through* means that after meeting the condition of one case statement, all subsequent case statements are valid and their code blocks also run.)

```
SQL> INSERT ALL
     INTO soldier
     VALUES
     (soldier_s.NEXTVAL,service_rank,service_member_name)
     INTO sailor VALUES
     (sailor_s.NEXTVAL,service_rank,service_member_name)
     SELECT 'MSG' AS service_rank
     ,      'Ernest G. Bilko' AS service_member_name FROM dual
     UNION ALL
     SELECT 'CPO' AS service_rank
     ,      'David Vaught' AS service_member_name FROM dual;
```

No `ELSE` block is used in the example. This type of statement would also run the `ELSE` block and perform any insert found in it.

Queries against the target tables show you that both rows are inserted into both tables:

```
SQL> SELECT * FROM soldier;

SOLDIER_ID SOLD SOLDIER_NAME
---------- ---- --------------------
         1 MSG  Ernest G. Bilko
         2 CPO  David Vaught

SQL> SELECT * FROM sailor;

 SAILOR_ID SAIL SAILOR_NAME
---------- ---- --------------------
         1 MSG  Ernest G. Bilko
         2 CPO  David Vaught
```

This type of statement is useful when you want to put data from one row of a table or view into multiple tables. Unfiltered `INSERT ALL` statements don't let you choose among a set of tables (like a filtered `INSERT ALL` statement), and they're used less than filtered statements.

Multiple-Table INSERT ALL with WHEN Clauses The multiple-table `INSERT ALL` statement also works with `WHEN` clauses that determine into which table they'll insert. The logic can include subqueries, as shown in the following example:

```
SQL> INSERT ALL
     WHEN service_rank IN (SELECT rank_short_name
                           FROM   rank_index
                           WHERE  rank_service = 'ARMY') THEN
```

```
  5  INTO soldier
  6  VALUES
  7  (soldier_s.NEXTVAL,service_rank,service_member_name)
  8  WHEN service_rank IN (SELECT rank_short_name
  9                        FROM   rank_index
 10                        WHERE  rank_service = 'NAVY') THEN
 11  INTO sailor
 12  VALUES
 13  (sailor_s.NEXTVAL,service_rank,service_member_name)
 14  SELECT 'MSG' AS service_rank
 15  ,      'Ernest G. Bilko' AS service_member_name FROM dual
 16  UNION ALL
 17  SELECT 'CPO' AS service_rank
 18  ,      'David Vaught' AS service_member_name FROM dual;
```

The WHEN clause on line 2 checks whether the service rank belongs in the Army. It inserts into the soldier table any row in which the query's service_rank value matches the subquery's rank_short_name value. The second WHEN clause does the same kind of evaluation against Navy ranks. Any rows that don't match one of the two criteria are discarded because there is no ELSE clause.

Queries against the target tables yield the following results:

```
SQL> SELECT * FROM soldier;

SOLDIER_ID SOLD SOLDIER_NAME
---------- ---- --------------------
         1 MSG  Ernest G. Bilko

SQL> SELECT * FROM sailor;

 SAILOR_ID SAIL SAILOR_NAME
---------- ---- --------------------
         2 CPO  David Vaught
```

The filtered INSERT ALL places rows from one source query into one or only the correct tables. This is the best practice—or at least the most frequently used version of the statement.

Multiple-Table INSERT FIRST with WHEN Clauses The INSERT FIRST works differently from the INSERT ALL statement. The INSERT FIRST inserts data into the first table only when it meets a WHEN clause condition. This means it performs like a switch statement in C#, C++, or Java where fall through is disabled. For your reference (in case you don't write programs in those languages), you disable fall through by adding a break statement in each case statement's code bock. The break statement signals completion and forces an exit from the switch statement. The FIRST keyword effectively does that for all WHEN clause statement blocks.

Here's an example using the concept of conscripts (draftees). The first one goes to the Army (line 2), the next four go to the Navy (line 6), and any others get to go home without serving in the military:

```
SQL> INSERT FIRST
  2  WHEN id < 2 THEN
  3  INTO soldier
```

```
VALUES
(soldier_s.NEXTVAL,'PVT',draftee)
WHEN id BETWEEN 2 AND 5 THEN
INTO sailor
VALUES
(sailor_s.NEXTVAL,'SR',draftee)
SELECT 1 AS ID,'John Sanchez' AS draftee FROM dual
UNION ALL
SELECT 2 AS ID,'Michael Deegan' AS draftee
FROM dual
UNION ALL
SELECT 3 AS ID,'Jon Voight' AS draftee FROM dual;
```

You'll see in the result set that only two draftees went in the Navy, so there weren't enough conscripts drafted today. The statement will need to be rewritten tomorrow for the new batch of draftees unless the rules change every day. The queries and results are shown next:

```
SQL> SELECT * FROM soldier;

SOLDIER_ID SOLD SOLDIER_NAME
---------- ---- --------------------
         1 PVT  John Sanchez

SQL> SELECT * FROM sailor;

 SAILOR_ID SAIL SAILOR_NAME
---------- ---- --------------------
         2 SR   Michael Deegan
         3 SR   Jon Voight
```

My suggestion is that `INSERT FIRST` statement is probably suited to dynamic creation inside Native Dynamic SQL (NDS). You can see examples of NDS in Chapter 13.

MySQL Insert by Values

`INSERT` statements in the MySQL database are used similarly to how they're used in the Oracle database. MySQL does let you insert multiple rows with a single `VALUES` clause, which isn't supported by the Oracle database. On the other hand, MySQL is a relational database and doesn't support object types or nested tables, as does Oracle 11*g*.

The following sections cover inserting scalar data types and performing data type conversions. MySQL treats all data types, including large character and binary strings, as scalar data types.

Inserting Scalar Data Types

MySQL lets you insert one or more rows through a `VALUES` clause. You don't have to call a sequence to use the clause during inserts, because sequences are a table property in MySQL. The `INSERT` statement mirrors more or less that of an Oracle database, because `INSERT` statements adhere to the ANSI SQL standard.

You can include an overriding signature in a MySQL `INSERT` statement just as you can in Oracle. You can exclude the single column designed with the `AUTO_INCREMENT` phrase, as well as any optional columns. The column created with an `AUTO_INCREMENT` phrase is a *surrogate* *primary key column*. Like Oracle, the INSERT statement checks the definition of the table in the data catalog. It checks the catalog when you exclude a column list and verifies constraints when you attempt to insert data.

Mirroring the steps from the Oracle discussion, you can use the DESCRIBE command to view the table definition from the catalog. The semicolon is required when describing a table in the MySQL Monitor, like so:

```
DESCRIBE table_name;
```

Using our rental table in My SQL requires a couple data type changes. That's because MySQL's DATE is a date, not a date-time data type. The DATETIME data type in MySQL is a date-time type equivalent to the DATE data type in an Oracle database. Here's the definition of the rental table (without the default value column) in MySQL:

```
+------------------+------------------+------+-----+----------------+
| Field            | Type             | Null | Key | Extra          |
+------------------+------------------+------+-----+----------------+
| rental_id        | int(10) unsigned | NO   | PRI | auto_increment |
| customer_id      | int(10) unsigned | NO   | MUL |                |
| check_out_date   | time             | NO   |     |                |
| return_date      | datetime         | NO   |     |                |
| created_by       | int(10) unsigned | NO   | MUL |                |
| creation_date    | time             | NO   |     |                |
| last_updated_by  | int(10) unsigned | NO   | MUL |                |
| last_update_date | time             | NO   |     |                |
+------------------+------------------+------+-----+----------------+
```

Like the Oracle table, the rental_id column is a surrogate primary key. The extra column description tells you that it's an automatically generated sequence value. Sequences are positive integers or doubles, which means they're defined as unsigned integers or unsigned doubles. It's a convention, and a recommendation, that surrogate primary keys use integers until they approach the 4 billion limit of the data type.

The syntax for this INSERT statement with literal values doesn't differ much from the Oracle equivalent, except for the MySQL proprietary NOW(), ADDDATE(), and UTC_DATE() function calls.

```
INSERT INTO rental
VALUES
( 1
, 2
, NOW()
, ADDDATE(UTC_DATE(),INTERVAL 5 DAY)
, 1
, NOW()
, 1
, NOW());
```

You can discover the next sequence value by using this syntax in MySQL Monitor:

```
show table status like 'rental'\G
```

The \G displays output vertically with the column names on the left and values on the right, as covered in Chapter 2. The show command displays this after one insert into the table:

```
***************** 1. row *****************
           Name: rental
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 1
 Avg_row_length: 16384
    Data_length: 16384
Max_data_length: 0
   Index_length: 49152
      Data_free: 8388608
 Auto_increment: 2
    Create_time: 2011-01-14 22:18:59
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
```

The important difference between the surrogate primary key column in Oracle and the same in MySQL deals with the auto-incrementing columns in MySQL. Although you do need to provide the column placeholder in the VALUES clause in MySQL, you can simply enter NULL and the correct sequence will be entered. This isn't true in Oracle, unless you've deployed an ON INSERT trigger to capture the sequence value and mapped it to the rental_id column.

In short, here is the preferred way to write an INSERT statement that uses the default signature and literal values or expression results (such as the dates from the NOW function or the date from the ADDDATE function). The INSERT statement uses a VALUES clause in MySQL:

```
INSERT INTO rental
VALUES
( null
, 2
, NOW()
, ADDDATE(UTC_DATE(),INTERVAL 5 DAY)
, 1
, NOW()
, 1
, NOW());
```

At this point, you know how to perform an INSERT statement with a null value and the default signature. It is preferred that you use a list of columns to designate what you're inserting in the VALUES clause. An INSERT statement that uses an overriding signature that excludes the auto-incrementing column follows:

```
INSERT INTO rental
( customer_id
, check_out_date
, return_date
, created_by
, creation_date
, last_updated_by
, last_update_date )
VALUES
( SELECT   c.contact_id
  FROM     contact c JOIN member m
  ON       c.member_id = m.member_id
  WHERE    c.first_name = 'Harry'
  AND      IFNULL(c.middle_name,'x') = IFNULL(null,'x')
  AND      c.last_name = 'Potter'
  AND      m.account_number = 'SLC-000006' )
, NOW()
, ADDDATE(UTC_DATE(),INTERVAL 5 DAY)
, 1
, NOW()
, 1
, NOW());
```

NOTE
Remember that you can use scalar subqueries in MySQL, but they do not exist in Oracle. Subqueries in the `VALUES` *clause can return only a single row when matched with literal values.*

You can also rewrite this in MySQL to work with the `SET` clause. The only problem with the `SET` clause, however, is that it's not portable to other databases. Here's an example of the syntax to `INSERT` with the `SET` clause:

```
INSERT INTO rental
SET customer_id = ( SELECT c.contact_id
                    FROM   contact c JOIN member m
                    ON     c.member_id = m.member_id
                    WHERE  c.first_name = 'Harry'
                    AND    IFNULL(c.middle_name,'x') = IFNULL(null,'x')
                    AND    c.last_name = 'Potter'
                    AND    m.account_number = 'SLC-000006' )
, check_out_date = NOW()
, return_date = ADDDATE(UTC_DATE(),INTERVAL 5 DAY)
, created_by = 1
, creation_date = NOW()
, last_updated_by = 1
, last_update_date = NOW();
```

The `customer_id` value comes from a scalar subquery (see Chapter 11) against the `contact` and `member` tables. The `WHERE` clause contains the business elements that help identify the surrogate primary key (the `contact_id` column) for the `contact` table.

Data Type Conversions

Like the Oracle database, MySQL supports the CAST and CONVERT functions and the BINARY operator. The CAST function works with the following data types: BINARY, CHAR, DATE, DATETIME, SIGNED or UNSIGNED INTEGER, and TIME. The CONVERT function lets you modify a strings character set, and the BINARY operator is shorthand for a CAST function that changes a character to a binary string.

CAST Function The CAST function also lets you specify a character set that you want to apply through the casting operation. Here's the prototype for the CAST function:

```
CAST(expression_or_variable AS data_type [CHARACTER SET character_set])
```

You would convert a double precision number to a string like so:

```
CAST(111.4586 AS CHAR(30))
```

Alternatively, you can change the character set when you cast a double to a character, like this:

```
CAST(111.4586. AS CHAR(30) CHARACTER SET utf8)
```

NOTE

Casting a string to a DATE, DATETIME, or TIME data type has some added restrictions. The string must conform to the default string literal for a date, which is YYYY-MM-DD (a four-digit year, two-digit month, and two-digit day). Any attempt to convert a different format mask fails and returns a null value.

CONVERT Function The CONVERT function is slightly different from the CAST function. Here's the prototype:

```
CONVERT([character_set] string_value USING character_set)
```

To convert a string from the default character set to a Unicode character set, use the following syntax:

```
CONVERT('Hello mate!' USING utf8)
```

BINARY Operator The difference between a function and operator in this case is parentheses: there aren't any in the operator. Here's the syntax to convert a string to a binary string:

```
BINARY 'Hello Mate!'
```

Insert by Queries

Inserting data from queries eliminates the VALUES clause—doing so actually replaces the clause with the query. The default and override signatures remain the same. Matching the columns returned by the SELECT statement replaces matching VALUES clause columns against the override column list or table definition.

The prototype changes slightly for the `INSERT` statement when the source becomes the results from a query rather than a list of values. The `VALUES` keyword must be dropped when you use a query or the `INSERT` statement fails.

The following prototype works for Oracle and MySQL databases and includes options for joining more than one table on various conditions and uses ANSI SQL-92 syntax:

```
INSERT INTO table_name
[( column_name1, column_name2, ...)]
 ( SELECT    column_value1, column_value2, ...
   FROM      table1
[[ JOIN table2 ON table1.column_name1 = table2.column_name2] ... ]
 [ WHERE     some_logical_conditions ]);
```

The prototype provides an optional list of column names for an override signature. Then it provides a single table query, optional join with potentially more joins, and ultimately a `WHERE` clause to filter rows. The column names of the query do not need to match the column names of the table to which the values are being added. Only the data types of the query's columns must match the data types of the table's columns. Fixed- and variable-length strings in queries must contain no more than the number of bytes or characters allowed by the table's definition.

`INSERT` statements from queries generally collect information from a series of tables. The number of rows returned for all columns must match. A problem occurs when literal values must map to a multiple row data set. The solution to this problem is to multiply the single row of the literal values against the multiple rows of the data set using a `CROSS JOIN`, which produces a Cartesian Product (every row in one table matched to every row in the other).

For example, let's say you need to enter an active price flag of *Y* with an amount of *$5* and an inactive price flag of *N* with an amount of *$3* for all merchandise in your item table. You need to multiply the single row of numeric and string literals by the number of rows in the table before inserting them into the price table. The solution would be an `INSERT` statement from a query that fabricates (or produces) a data set not found in your raw data, like this:

```
SQL>  INSERT INTO price
     (SELECT   price_s1.NEXTVAL
      ,        i.item_id
      ,        dt.active_flag
      ,        dt.amount
      FROM    (SELECT   'Y' AS active_flag
               ,         5  AS amount
               FROM      dual
               UNION ALL
               SELECT   'N' AS active_flag
               ,         3  AS amount
               FROM      dual) dt CROSS JOIN item i);
```

The in-line view creates a two-column by two-row table with the values `'Y'` and 5 in one row and the values `'N'` and `3` in the other row. The number of rows in the item table will be multiplied by the two rows in the fabricated table and the combined and balanced row set joined by a sequence value for each row in the outer `SELECT` clause. The only change required to implement this sample in a MySQL database switches the `price_s1.NEXTVAL` pseudo column with a `null` value.

NOTE
The `CROSS JOIN` is a forward reference to Chapter 11. If you're new to SQL and want to understand the full nature of the Oracle and MySQL examples, you should check out the basics of join mechanics in Chapter 11.

The next two sections demonstrate the syntax for `INSERT` statements with queries, with two approaches shown: the first shows a query that contains all the mandatory columns, and the other shows a query that contains only some of the mandatory columns with additional values that come from string literals. The only differences between Oracle and MySQL are the functions that can convert a query return column to match the definition of a table column and the handling of automatic numbering through sequences.

Oracle Insert by Queries

Here we'll show a query that contains all the columns necessary to insert a row in a table and a query that doesn't contain all the columns necessary for the `INSERT` statement. The latter mixes string literals with query results through a `CROSS JOIN` operation, which fabricates data sets.

The `INSERT` statement will put the data into a table defined as follows:

```
Name                          Null?    Type
----------------------------- -------- ----------------------------
KNIGHT_ID                     NOT NULL NUMBER
KNIGHT_NAME                   NOT NULL VARCHAR2(20)
KINGDOM_ALLEGIANCE_ID         NOT NULL NUMBER
```

The `knight` table has three columns. The `knight_id` column is first and holds a surrogate key column that would store a sequence value. The `knight_name` and `kingdom_allegiance_id` columns hold a natural key, which is unique. That means there can be only one *Peter the Magnificent* with allegiance to a given kingdom stored in the `kingdom` table, because the `kingdom_allegiance_id` column holds a foreign key that points to the `kingdom_id` column in the `kingdom` table. The following conventions are used in this model: Primary key columns are surrogate key columns, and they use the name of the table plus an `_id` suffix; foreign key columns can have names that differ from their related primary key column names.

The query finds knights in the `available_knight` table. The following `INSERT` statement puts the data into the table:

```
SQL> INSERT INTO knight
     ( knight_id
     , knight_name
     , kingdom_allegiance_id )
     ( SELECT   knight_s1.NEXTVAL
       ,        knight_name
       ,       (SELECT   kingdom_id
                FROM     kingdom
                WHERE    kingdom_name = 'Narnia')
       FROM     available_knight);
```

The query gets the values from the `available_knight` table, and a subquery on lines 5 through 10 gets the correct foreign key value from the `kingdom` table. Inside the query, a call to the `knight_s1` sequence value ensures the surrogate key is automatically populated.

An alternative syntax would exclude the surrogate key column, because an `ON INSERT` database trigger populates the column. Database triggers support this activity effectively, and many developers use them for this purpose.

The use of an override signature means you can reshuffle the order of columns to fit particular business needs. Be sure to make the same changes in the override signature that you make in the query. The number of possibilities varies with the complexity of queries required to solve the business problems.

MySQL Insert by Queries

Leveraging the same example from the "Oracle Insert by Queries" section, you need only change the first column value returned by the query. The `.NEXTVAL` pseudo column is an Oracle-only solution. In MySQL, you would perform the query with a `null` value as the first column of the query, like so:

```
mysql> INSERT INTO knight
    -> ( knight_id
    ->  , knight_name
    ->  , kingdom_allegiance_id )
    ->  ( SELECT   null
    ->    ,        knight_name
    ->    ,       (SELECT   kingdom_id
    ->             FROM     kingdom
    ->             WHERE    kingdom_name = 'Narnia')
    ->    FROM     available_knight);
```

An alternative example would implement an override signature that excludes the surrogate key (or auto-incrementing) column. The syntax would then be as follows:

```
mysql> INSERT INTO knight
    -> (  knight_name
    ->  , kingdom_allegiance_id )
    ->  ( SELECT   knight_name
    ->    ,       (SELECT   kingdom_id
    ->             FROM     kingdom
    ->             WHERE    kingdom_name = 'Narnia')
    ->    FROM     available_knight);
```

The power of inserting from queries is substantial. It lets you collect data from many sources and then insert rows of data in a single statement.

Summary

In this chapter, you learned about inserting data. You can insert a row at a time with a `VALUES` clause and many rows at a time with a subquery. The syntax for these statements differs little between the Oracle and MySQL databases. The most significant issue is handling sequence values.

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 `INSERT` statement supports multiple row inserts through the `VALUES` clause in Oracle.
2. **True** ☐ **False** ☐ An `INSERT` statement supports multiple row inserts through the `VALUES` clause in MySQL.
3. **True** ☐ **False** ☐ The list of columns in an override signature must match the number and data types of the list of values in the `VALUES` clause.
4. **True** ☐ **False** ☐ You can insert data into nested tables with an `INSERT` statement in an Oracle database.
5. **True** ☐ **False** ☐ In an Oracle database, you can use a `.CURRVAL` pseudo column in a `VALUES` clause provided the sequence has been placed in scope through an earlier `.NEXTVAL` pseudo column call.
6. **True** ☐ **False** ☐ A null value that maps to an auto-incrementing column in a MySQL database inserts the next number in the table's sequence.
7. **True** ☐ **False** ☐ In MySQL, you can use an override signature only when you want to insert sequence values.
8. **True** ☐ **False** ☐ An `INSERT` statement that uses a query can insert one to many rows of data in both Oracle and MySQL databases.
9. **True** ☐ **False** ☐ MySQL supports the `.NEXTVAL` pseudo column.
10. **True** ☐ **False** ☐ Both Oracle and MySQL databases let you use subqueries in the `VALUES` clause of an `INSERT` statement.
11. Which of the following data types requires a built-in function call to put a long variable length character string in a `VALUES` clause of an `INSERT` statement?
    - **A.** `BLOB`
    - **B.** `MEDIUMCLOB`
    - **C.** `TEXT`
    - **D.** `CLOB`
    - **E.** `VARCHAR`
12. In an Oracle database, how many rows can you insert through a `VALUES` clause?
    - **A.** 1
    - **B.** 2
    - **C.** 3
    - **D.** 4
    - **E.** Many

13. In an MySQL database, how many rows can you insert through a `VALUES` clause?

- **A.** 1
- **B.** 2
- **C.** 3
- **D.** 4
- **E.** Many

14. When you use a query instead of a `VALUES` clause, which of the following is true about the query?

- **A.** The query can contain scalar subqueries.
- **B.** The query can return only one row at a time.
- **C.** The query must return a unique data set.
- **D.** The query can't be the product of a join between two or more tables.
- **E.** The query is independent of the table structure, which means you can return more columns than the table will accept.

15. Which of the following can't be put in a `VALUES` clause?

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