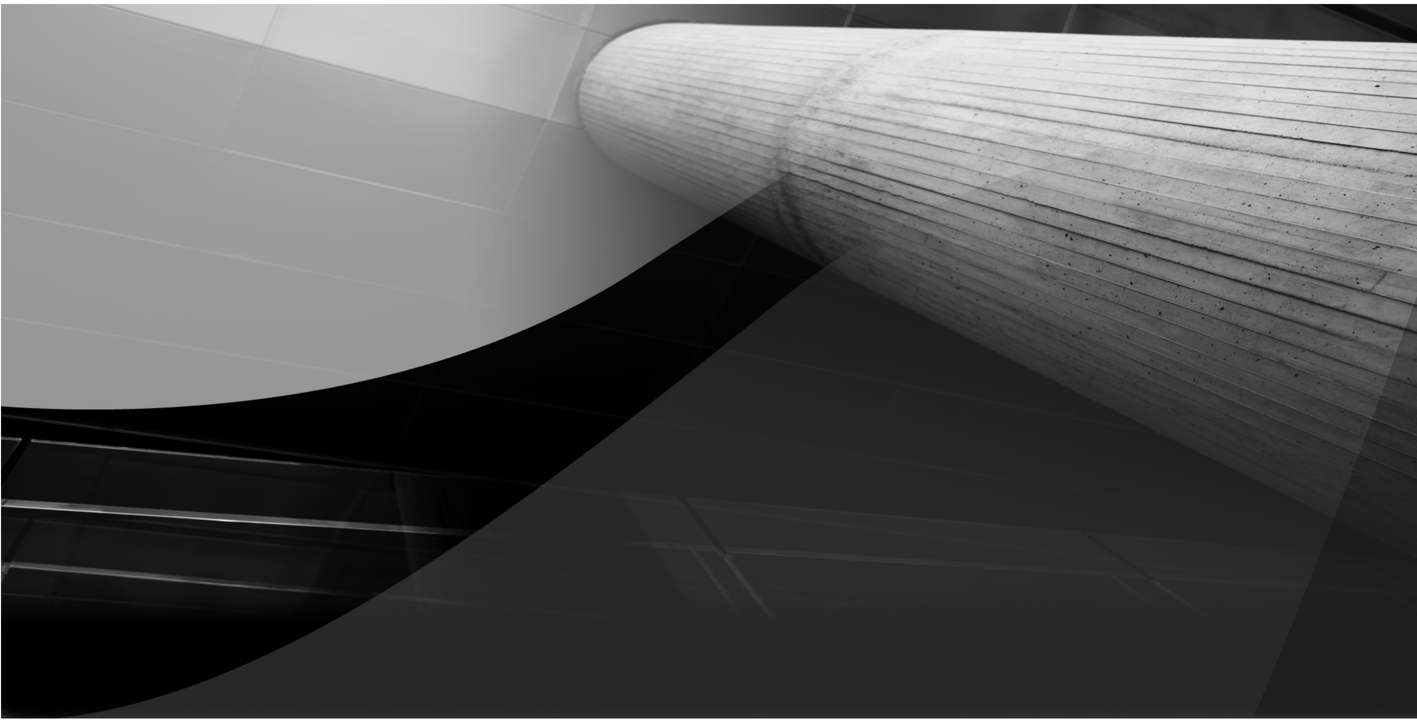


CHAPTER 11

Querying

The SQL `SELECT` statement lets you query data from the database. In many of the previous chapters, you've seen examples of queries. Queries support several different types of subqueries, such as nested queries that run independently or correlated nested queries. Correlated nested queries run with a dependency on the outer or containing query.

This chapter shows you how to work with column returns from queries and how to join tables into multiple table result sets. Result sets are like tables because they're two-dimensional data sets. The data sets can be a subset of one table or a set of values from two or more tables. The `SELECT` list determines what's returned from a query into a result set. The `SELECT` list is the set of columns and expressions returned by a `SELECT` statement. The `SELECT` list defines the record structure of the result set, which is the result set's first dimension. The number of rows returned from the query defines the elements of a record structure list, which is the result set's second dimension.

You filter single tables to get subsets of a table, and you join tables into a larger result set to get a superset of any one table by returning a result set of the join between two or more tables. Joins expand the `SELECT` list by adding columns or expressions, and set operations expand the number of rows returned.

This chapter covers three topics:

- Query results
- Join results
- Views: stored queries

The query results section shows you how to manage column values; perform concatenation or parsing, math, and date calculations; and execute data type casting operations. It also shows you how to perform conditional logic and aggregation operations.

The join results section shows you how to perform cross, inner, natural, left outer, right outer, and full outer joins between tables. It demonstrates both the ANSI SQL-89 and ANSI SQL-92 syntax for joins. This section also covers how to expand Oracle's nested tables into two-dimensional result sets and how to use the `INTERSECT`, `UNION`, `UNION ALL`, and `MINUS` set operators.

You'll examine the basic flow of operation for `SELECT` statements, which applies to both databases. The biggest difference between `SELECT` statements in Oracle and MySQL databases involve their built-in functions. Built-in functions let you perform specialized behaviors such as casting from one type to another, date mathematics, and null replacements. You'll also understand how to create views, which are stored queries.

Query Results

A `SELECT` statement (query) reads differently from how it acts. In English, a *query* selects something from a table, or a set of tables, where certain conditions are true or untrue. Translating that English sentence into programming instructions is the beauty and complexity of SQL. Although English seems straightforward, queries work in a different event order. The event order also changes with different types of queries.

Queries can be divided into three basic types:

- Queries that return columns or results from columns
- Queries that aggregate, or queries that return columns or results from columns by adding, averaging, or counting between rows

- Queries that return columns or results selectively (filtered by conditional expressions such as *if statements*), and these types of queries may or may not aggregate result sets

You can return column values or expressions in the SELECT list. Column values are straightforward, because they're the values in a column. Expressions aren't quite that simple. Expressions are the results from calculations. Some calculations involve columns and string literal values, such as concatenated results (strings joined together to make a big string), parsed results (substrings), or the mathematical result of the columns, literal values, and function returns. Mathematical results can be calculated on numeric or date data types and returned as function results from several built-in functions in both databases.

You can also be selective in your SELECT list, which means you can perform if-then-else logic in any column. The selectivity determines the resulting value in the final result set. Result sets are also formally called *aggregate results* because they've been assembled by SELECT statements.

Here's the basic prototype for a SELECT list:

```
SELECT {column_name | literal_value | expression } AS alias [,     {...}]]
WHERE [NOT] column_name {{= | <> | > | >= | < | <=} |
                           [NOT] {{IN | EXISTS} | IS NULL}} 'expression'
[{AND | OR } [NOT] comparison_operation] [...];
```

You can return three things as an element in the *SELECT* list: a column value from a table or view, a literal value, and an expression. The *column value* is easy to understand, because it's the value from the column—but what is its data type? A column returns the value in its native data type when you call the query from a procedural programming language, such as C, C#, C++, Java, PL/SQL, or SQL/PSM, or as a subquery. Subqueries are queries within queries and are covered in the "Subqueries" section later in this chapter. A column returns a string when you call the query from SQL*Plus or MySQL Monitor, and it is written to a console or a file. *Literal values* must have a column alias when you want to reuse the value in a procedural program or as a subquery result, and in those cases the values must be a string or number. *Expressions* are more difficult because they're the result of processing operations, such as concatenation or calculation, or they return results from built-in or user-defined functions.

The next three examples show you how the types of queries work. All examples use queries from a single table to let you focus on the differences between types.

Queries that Return Columns or Results from Columns

Figure 11-1 shows how a query returns a result set of column values in the SELECT list. You can see how the elements are labeled and processed and the figure helps you visualize table aliases, column aliases, basic comparison operations, and the basic order of clauses within the SELECT statement.

The following list qualifies the ANSI SQL pattern for processing a single table query:

- It finds a table in the FROM clause.
- It *optionally* assigns a table alias as a runtime placeholder for the table name.
- It gets the table definition from the data catalog to determine the valid column names (not shown in the figure because it's a hidden behavior).
- If a table alias is present (and it is), it optionally maps the alias to the table's data catalog definition.
- It filters rows into the result set based on the value of columns in the WHERE clause.

- The list of columns in the `SELECT` clause filters the desired columns from the complete set of columns in a row.
- If an `ORDER BY` clause occurs in the query, rows are sorted by the designated columns.

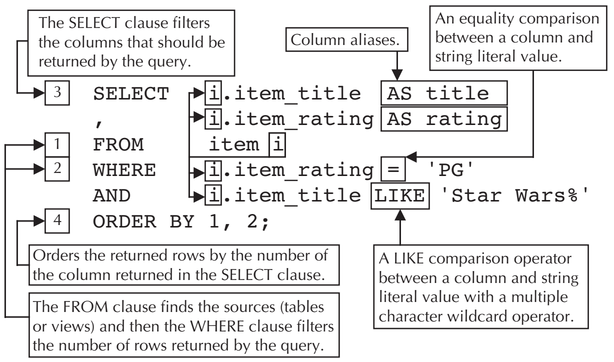
Figure 11-1 also demonstrates table and column aliases. The table alias is generally unnecessary when writing a query against a single table. It is useful and necessary when you want to avoid typing complete table names to disambiguate column names that are the same in two or more tables. Because the `FROM` clause is read first, all references to the item table are mapped to `i` in the rest of the query. This means that a reference to `item.item_title` would not be found.

TIP
The `AS` keyword is optional when setting column aliases but ensures clarity that an alias follows it. Consistent use increases typing but decreases support costs.

Column aliases shorten the `item_title` and `item_rating` column names to *title* and *rating*, respectively. Aliases let you use shorter or more descriptive words for columns in a specific use case. Sometimes the shorter words aren't appropriate as column names because they're too general, such as *title*. The `AS` keyword is optional in both Oracle and MySQL databases, but I recommend that you use it, because the clarity can simplify maintenance of queries. Just note that `AS` works only with column aliases and would create a statement parsing error if you tried to use it before a table alias.

NOTE
The `AS` keyword cannot precede a table alias in Oracle; it can precede only a column alias. MySQL supports an AS keyword for a table alias.

In our example, we can modify the `SELECT` list to return an expression by concatenating a string literal of "MPAA:" (Motion Picture Association of America) to the `item_rating` column.



```
3  SELECT   i.item_title   AS title
   ,        i.item_rating  AS rating
1  FROM     item i
2  WHERE    i.item_rating  =   'PG'
   AND      i.item_title   LIKE 'Star Wars%'
4  ORDER BY 1, 2;
```

The SELECT clause filters the columns that should be returned by the query.
Column aliases.
An equality comparison between a column and string literal value.
Orders the returned rows by the number of the column returned in the SELECT clause.
The FROM clause finds the sources (tables or views) and then the WHERE clause filters the number of rows returned by the query.
A LIKE comparison operator between a column and string literal value with a multiple character wildcard operator.

FIGURE 11-1. *Queries that return columns or results from columns*

Concatenating strings is like gluing them together to form a big string. It would look like this in Oracle using a *piped concatenation*:

```
SELECT   i.item_title AS title
,        'MPAA: ' || i.item_rating AS rating
```

The two vertical bars (||) are *pipes*, and when you use them to glue strings together, it's known as *piped concatenation*. MySQL doesn't support piped concatenation, so you use a built-in function to glue strings together, like so:

```
SELECT   i.item_title AS title
,        CONCAT('MPAA: ',i.item_rating) AS rating
```

The CONCAT built-in function returns a string by concatenating the call parameters sequentially. Notice that literals and column values are equal call parameters to the function. Both of these would return results like these:

```
TITLE                     RATING
------------------------- ----------
Star Wars I               MPAA: PG
Star Wars II              MPAA: PG
Star Wars III             MPAA: PG
```

In another context, you can perform mathematical operations and string formatting. The following SELECT list retrieves a transaction_date and a transaction_amount column from the transaction table:

```
SQL> SELECT   t.transaction_date
  2  ,        TO_CHAR(t.transaction_amount,'90.00') AS price
  3  ,        TO_CHAR(t.transaction_amount * .0875,'90.00') AS tax
  4  ,        TO_CHAR(t.transaction_amount * 1.0875,'90.00') AS total
  5  FROM     transaction t
  6  WHERE    t.transaction_date = '10-JAN-2009';
```

The TO_CHAR function formats the final number as a string. The 90.00 format mask instructs the display as follows: a 9 means display a number when present and ignore the number when it is not present; 0 means display a number when it is present and display a 0 when no number is present. Inside the TO_CHAR function on lines 3 and 4, the column value is multiplied by numeric literals that represent sales tax and price plus sales tax. The query would produce output like so:

```
Date      Price  Tax    Total
--------- ------ ------ ------
10-JAN-09   9.00   0.79   9.79
10-JAN-09   3.00   0.26   3.26
10-JAN-09   6.00   0.53   6.53
```

The output is left-aligned, which means it's formatted as a number, because strings are displayed as right-aligned.

The equivalent formatting function in MySQL is the FORMAT function, which performs like this:

```
,        FORMAT(t.transaction_amount * .0875,2) AS tax
```

The FROM clause takes a single table or a comma-separated list of tables when writing queries in ANSI SQL-89 format. The FROM clause takes tables separated by join keywords and their join criterion or criteria in ANSI SQL-92 syntax.

The WHERE clause performs two types of comparisons. One is an equality comparison of two values, which can come from columns, literals, or expressions. The other is an inequality comparison, which can check when one value is found in another (such as a substring of a larger string); when one value is greater than, greater than or equal to, less than, or less than or equal to another; when one value isn't equal to another value; when one value is in a set of values; or when one value is between two other values. You can also state a negative comparison, such as WHERE NOT. The WHERE NOT comparison acts like a *not equal to* operation.

Two specialized operators let you limit the number of rows returned by a query. Oracle supports a ROWNUM pseudo column and MySQL supports the LIMIT function. You use ROWNUM in Oracle to retrieve only the top five rows, like this:

```
WHERE rownum <= 6;
```

Here's the LIMIT function in MySQL:

```
WHERE LIMIT 5;
```

These are handy tools when you've presorted the data and know where to cut off the return set. When you forget to sort, the results generally don't fit what you're looking for.

The data set in the table determines whether the query returns unique or non-unique data—that is, there could be multiple rows with an item_title of "Star Wars: A New Hope," and they would be returned because they match the criteria in the WHERE clause. You can use the DISTINCT operator to suppress duplicates without altering the logic of the WHERE clause (see Figure 11-2).

Regular Expression Alternatives

Both Oracle and MySQL databases provide regular expression alternatives to the LIKE comparison operator. They aren't cross-portable, which makes the LIKE comparison operator the more generic or application-neutral approach.

Oracle Regular Expression Alternative Oracle provides a variation on the generic SQL LIKE comparison with the REGEXP_LIKE function. The last line of the query in Figure 11-1 could use the following in an Oracle database:

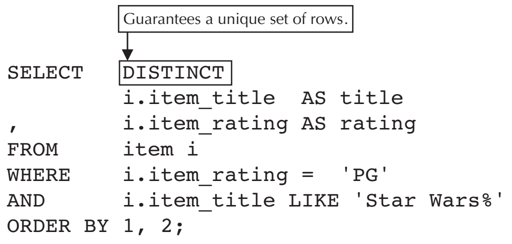
```
AND      REGEXP_LIKE(i.item_title,'^Star Wars.+');
```

More information on the regular expression functions provided by the Oracle Database is in Appendix E of *Oracle Database 11*g *PL/SQL Programming*.

MySQL Regular Expression Alternative MySQL also provides a variation on the generic SQL LIKE comparison operator. It uses the REGEXP operator. The last line of the query in Figure 11-1 could use the following in a MySQL database:

```
AND      i.item_title REGEXP '^Star Wars.+';
```

The regular expression match in both cases uses the carat (^) symbol (above the 6 on most keyboards) at the beginning to indicate the beginning of the string. They both likewise use the dot-plus (.+), which says look for any number of other trailing characters.



```
SELECT   DISTINCT
         i.item_title  AS title
,        i.item_rating AS rating
FROM     item i
WHERE    i.item_rating =  'PG'
AND      i.item_title LIKE 'Star Wars%'
ORDER BY 1, 2;
```

FIGURE 11-2. *Query that returns distinct columns or results from columns*

There is no difference between Oracle and MySQL when using the `DISTINCT` operator, which sorts the set to return a unique set—one copy of every row. Other than an incremental sort and disposal of duplicate rows, the query in Figure 11-2 performs more or less the same steps as the query shown in Figure 11-1. The next two subsections discuss subqueries and *in-line views*, also known as *runtime* or *derived tables*.

Subqueries

Subqueries are any `SELECT` statement nested within another DML statement, such as `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements. Subqueries have been demonstrated in earlier chapters because they're very useful.

Four types of basic subqueries can be used:

- **Scalar subqueries** Return only one column and one row of data
- **Single-row subqueries** Return one or more columns in one row of data
- **Multiple-row subqueries, ordinary subqueries, or subqueries** Return one or more columns in one or more rows of data
- **Correlated subqueries** Return nothing, but they effect a join between the outer DML statement and correlated subquery

Another subquery can be used inside the `FROM` clause of a `SELECT` statement. This type of subquery is actually a *runtime view*, or *derived table*. It isn't technically a subquery. The following sections describe the uses and occurrences of subqueries in DML statements.

Scalar Subqueries Scalar subqueries return only one thing: one column from one row. They return a single value from a query. That's because scalar variables are numbers, dates, strings, and timestamps. Scalar data types are like primitive data types in the Java programming language.

Scalar subqueries are much like functions. You put comparative statements in the `WHERE` clause to find a single row, similar to defining formal parameters in a function. Then you return a single column in the `SELECT` clause, which inherits its data type from the data catalog. The `SELECT` clause designates the return data type of a function just like the return keyword in procedural programming languages.

You can use scalar subqueries in the following places in DML statements:

- The `VALUES` clause of an `INSERT` statement
- The `SELECT` clause of a `SELECT` statement

- The SET clause of an UPDATE statement
- The WHERE clause of a SELECT, UPDATE, or DELETE statement

NOTE
Chapter 8 shows you how to use scalar subqueries in the VALUES clause. Chapter 9 shows you how to use scalar subqueries in the SET and WHERE clauses of an UPDATE statement, and Chapter 10 shows you how to use them in the WHERE clause of a DELETE statement.

Single-Row Subqueries Single-row subqueries return one or more columns from a single row. This is more or less like returning a record data type. You can apply the same analogy of comparative statements in the WHERE clause mapping to formal parameter definitions and the return type mapping to the list of columns in the SELECT clause. When you exclude the scalar behaviors of a single-row subquery, the following uses remain:

- The SET clause of an UPDATE statement
- The WHERE clause of a SELECT, UPDATE, or DELETE statement

NOTE
Chapter 9 shows the use of single-row subqueries in the SET clause of UPDATE statements. Chapters 9 and 10 show the respective use of single-row subqueries in the WHERE clauses of UPDATE and DELETE statements.

Multiple-Row Subqueries Multiple-row subqueries are frequently called ordinary subqueries or just subqueries. These subqueries return one to many columns and rows of data. That means they return result sets that mimic two-dimensional tables.

You can use multiple-row subqueries only in the WHERE clause of SELECT, UPDATE, or DELETE statements. You also must use a valid lookup comparison operator, such as the IN, =ANY, =SOME, or =ALL operators. These operators act like a chain of logical OR comparisons in a WHERE clause, because they look to see if the leftmost operand in the comparison is found in the list of possible values. The leftmost operand can be a single column or a record data type comprising two or more columns.

A couple of quick examples to qualify these behaviors might help. These examples use the pseudo dual table to keep them bare bones. The list of values inside a set of parentheses is the same as the value set returned by a multiple-row subquery. Here's a standard use of a logical or lookup comparison that deals with a single column:

```
SELECT 'True Statement' FROM dual
WHERE  'Lancelot' IN ('Arthur','Galahad','Lancelot');
```

This is equivalent to the chaining of logical or statements in the WHERE clause, like this:

```
SELECT 'True Statement' FROM dual
WHERE 'Lancelot' = 'Arthur'
OR    'Lancelot' = 'Galahad'
OR    'Lancelot' = 'Lancelot';
```

The syntax doesn't change much when you make the comparison of a record type to a list of record types. The only other change is the substitution of the =ANY lookup operator for the IN operator. As you can see, the lookup operators work the same way in this example:

```
SELECT 'True Statement' FROM dual
WHERE ('Harry Potter and the Chamber of Secrets','PG') =ANY
      (('Harry Potter and the Sorcerer's Stone','PG')
      ,('Harry Potter and the Chamber of Secrets','PG')
      ,('Harry Potter and the Prisoner of Azkaban','PG'));
```

which would work like this with a set of logical OR comparisons:

```
SELECT 'True Statement' FROM dual
WHERE (('Harry Potter and the Order of the Phoenix','PG-13') =
        ('Harry Potter and the Sorcerer's Stone','PG')
OR     ('Harry Potter and the Order of the Phoenix','PG-13') =
        ('Harry Potter and the Chamber of Secrets','PG')
OR     ('Harry Potter and the Order of the Phoenix','PG-13') =
        ('Harry Potter and the Prisoner of Azkaban','PG'));
```

The =ALL lookup operator is different, because it checks whether a scalar or record data type is found in all instances of a list. This means it works on a logical and comparison basis. This statement

```
SELECT 'True Statement' FROM dual
WHERE 'Lancelot' =ALL ('Lancelot','Lancelot','Lancelot');
```

is roughly equivalent to this:

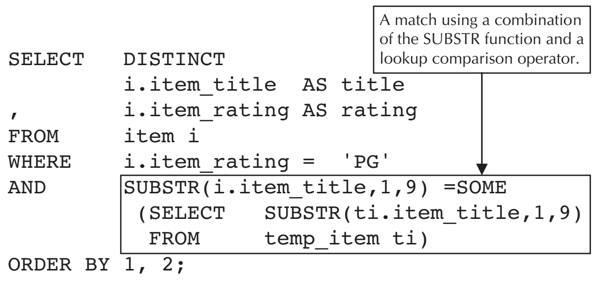
```
SELECT 'True Statement' FROM dual
WHERE 'Lancelot' = 'Lancelot'
AND   'Lancelot' = 'Lancelot'
AND   'Lancelot' = 'Lancelot';
```

Although these examples use lists of literal values, you could substitute multiple-row subqueries. In many cases, this type of comparison is unnecessary because the same logic can be resolved through ordinary join statements.

The only problem with lookup comparison operators is that they don't easily extend the behavior of the LIKE comparison operator. Figure 11-2 introduced a LIKE operator against a string literal with a wildcard operator. When the literal value is replaced by a subquery, the comparison no longer works when the query returns more than one row. It would fail with an ORA-01427 error in Oracle, which tells you a "single-row subquery returns more than a one row."

You can fix this behavior by doing two things. Substitute an IN, =ANY, or =SOME lookup comparison operator for the LIKE comparison operator, and use the SUBSTR function to make the comparison against exact matches. This allows you to match the substrings that should be the same. This particular match (shown in Figure 11-3) starts at the first character, which is position 1, because characters in strings are 1-based, not 0-based, in databases, and use the first nine characters.

Oracle and MySQL both support the SUBSTR function. MySQL implements the SUBSTR function as an alias for the SUBSTRING built-in, which works exactly like the equivalent function in Oracle.


A match using a combination of the SUBSTR function and a lookup comparison operator.

```
SELECT   DISTINCT
         i.item_title  AS title
,        i.item_rating AS rating
FROM     item i
WHERE    i.item_rating =  'PG'
AND      SUBSTR(i.item_title,1,9) =SOME
          (SELECT   SUBSTR(ti.item_title,1,9)
           FROM     temp_item ti)
ORDER BY 1, 2;
```

FIGURE 11-3. *Wildcard comparison against multiple row subquery*

Correlated Subqueries Correlated subqueries join the inside query to a value returned by each row in the outer query. As such, correlated subqueries act as function calls made for each row returned by the outer query. The rule of thumb on correlated subqueries requires that you join on uniquely indexed columns for optimal results. Ordinary subqueries typically outperform correlated subqueries when you can't join on uniquely indexed columns.

Correlated subqueries appear to return something when they're inside the SET clause of an UPDATE statement. If you flip back to Chapter 9, you'll see that a multiple-row subquery actually returns the value based on a match between the row being updated and a nested correlated subquery. The actual update is performed by a multiple-row subquery, not a correlated subquery. Correlated subqueries can't return values through the SELECT list; they can only match results in their WHERE clause.

You can use correlated subqueries in the following:

- The SELECT list
- The SET clause of an UPDATE statement
- The WHERE clause of a SELECT, UPDATE, or DELETE statement

The multiple-row subquery example extends the behavior of Figure 11-2. The multiple-row subquery runs once for the outer query and returns a list of values. The IN, =ANY, or =SOME lookup operator lets you perform a lookup to determine whether a variable is found in the list of returned values.

Although less efficient, a correlated subquery can be used to solve this type of comparison problem with a regular expression. Oracle lets you perform it with the REGEXP_LIKE function, like this:

```
AND      EXISTS
  (SELECT NULL
   FROM   temp_item ti
   WHERE  REGEXP_LIKE(i.item_title,'^'||SUBSTR(ti.item_title,1,9)||'.+'));
```

Or you can do it like this with MySQL's REGEXP operator:

```
AND      EXISTS
   (SELECT NULL
    FROM   temp_item ti
    WHERE  i.item_title REGEXP CONCAT('^',SUBSTR(ti.item_title,1,9),'.+'));
```

In these correlated query examples, you see Oracle's piped concatenation model and MySQL's CONCAT function. Oracle's resolution is case sensitive while MySQL's is case insensitive.

In-line Views

An in-line view is a query inside the FROM clause of a query or inside a WITH clause. The WITH clause is newer and was introduced in the ANSI SQL-1999 standard. There's no implementation of the WITH clause in MySQL at the time of writing. In-line views are also labeled as runtime or derived tables, and Microsoft calls them Common Table Expressions (CTE).

The query in a FROM or WITH clause dynamically creates a view at runtime. It's possible that the same in-line view can be used in multiple places within a large query. When an in-line view appears in multiple places within a query, it is run multiple times. This is inefficient and unnecessary when the WITH clause is supported in the database. The WITH clause provides an in-line view with a named reference, runs it only once, and lets you use the name reference in more than one place in the query.

Here's a sample of an in-line view in the FROM clause:

```
SQL> SELECT  c.first_name||' '||c.last_name AS person
  2  ,       inline.street_address
  3  ,       inline.city
  4  ,       inline.state_province
  5  FROM    contact c INNER JOIN
  6         (SELECT   a.contact_id
  7          ,        sa.street_address
  8          ,        a.city
  9          ,        a.state_province
 10          FROM     address a INNER JOIN street_address sa
 11          ON       a.address_id = sa.address_id) inline
 12  ON      inline.contact_id = c.contact_id;
```

The in-line view is on lines 6 through 11 and the join between the inline view and contact table is on line 12. The in-line view must return the foreign key column in the SELECT list for it to be used later in the join on line 12. Failure to return the key column in the SELECT list of the in-line view would leave nothing to use in a JOIN statement.

The only change required to run the preceding example in a MySQL database can be made on line 1. MySQL doesn't support piped concatenation as Oracle does, so you need to replace the concatenating line with the following in MySQL:

```
mysql> SELECT  CONCAT(c.first_name,' ',c.last_name) AS person
```

Although it's possible to enable piped concatenation in MySQL, don't bother. Even when you enable it, you can't perform piped concatenation inside call parameters to other functions. That more or less means it works only some of the time. It's better just to recognize that the CONCAT function in MySQL is your friend and use it. It's recursive and takes however many arguments you require, as opposed to Oracle's implementation that takes only two arguments, like this:

```
SELECT CONCAT('Not ',CONCAT('a recursive ','function.')) AS result
FROM dual;
```

This prints the following:

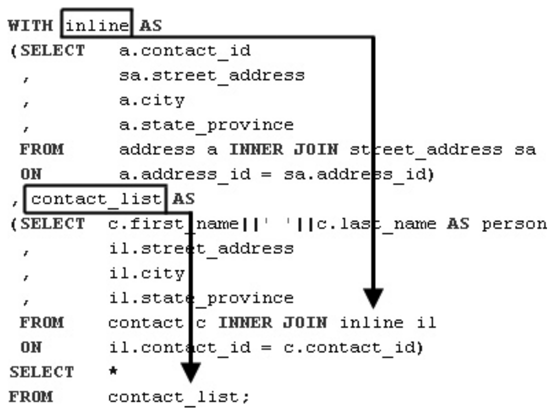
```
RESULT
-------------------------
Not a recursive function.
```

The preceding in-line view can be refactored to work with a `WITH` clause, like this:

```
SQL> WITH inline AS
  2  (SELECT   a.contact_id
  3   ,        sa.street_address
  4   ,        a.city
  5   ,        a.state_province
  6   FROM     address a INNER JOIN street_address sa
  7   ON       a.address_id = sa.address_id)
  8  SELECT  c.first_name||' '||c.last_name AS person
  9  ,       inline.street_address
 10  ,       inline.city
 11  ,       inline.state_province
 12  FROM    contact c INNER JOIN inline inline
 13  ON      inline.contact_id = c.contact_id;
```

Line 12 references the inline name of the in-line view, which is on lines 1 to 7. Line 12 identifies an `INNER JOIN` between the contact table and in-line view, and line 13 provides the criteria to match values between the table and view. Large queries have a tendency to reuse in-line views in multiple places, which isn't a good thing. In-line views must be run each time they're encountered in the query. The `WITH` clause fixes this performance nightmare, because the query is run once, given a name, and then the result sets are usable anywhere else in the query. The `WITH` clause should always be used unless your code must be portable to MySQL, which doesn't support it.

It's also possible to have multiple in-line views. You list them with a designated name in a comma-delimited list. Figure 11-4 shows a small example of a `WITH` clause that promotes the ordinary query to a view. The lines and arrows help you see the use of the names for the in-line views in the query.



```
WITH inline AS
(SELECT   a.contact_id
 ,        sa.street_address
 ,        a.city
 ,        a.state_province
 FROM     address a INNER JOIN street_address sa
 ON       a.address_id = sa.address_id)
, contact_list AS
(SELECT  c.first_name||' '||c.last_name AS person
 ,       il.street_address
 ,       il.city
 ,       il.state_province
 FROM    contact c INNER JOIN inline il
 ON      il.contact_id = c.contact_id)
SELECT   *
FROM     contact_list;
```

FIGURE 11-4. *With clause*

The benefit of the `WITH` clause is that it runs once and can be used multiple times in the scope of the query. The only downside with the syntax is that it's not portable between Oracle and MySQL, but that might change someday.

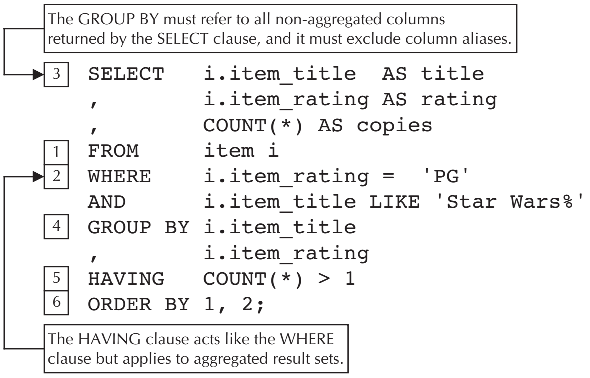
Queries that Aggregate

Aggregation is one of those buzz words in databases. Aggregation means counting, adding, and grouping results of `COUNT`, `SUM`, `AVERAGE`, `MIN`, and `MAX` functions. Aggregation queries add one or two more clauses than those presented in Figure 11-1. Figure 11-5 shows the `GROUP BY` and `HAVING` clauses.

The `GROUP BY` clause must refer to all non-aggregated columns in the `SELECT` list, because they're not unique and there's no sense in returning all the rows when you need only one row with the non-unique columns and the aggregated result. The `GROUP BY` instructs the database to do exactly that: return only distinct versions of non-unique columns with the aggregated result columns. As you can see in Figure 11-5, the `GROUP BY` clause runs after the query has identified all rows and columns. The `COUNT` function takes an asterisk (`*`) as its single argument. The `*` represents an indirection operator that points to rows returned by the query. The `*` is equivalent to the `ROWID` pseudo column in Oracle. It counts rows whether a row contain any values or not.

NOTE
The ROWID is one of the places where the concept of indirection and a pointer shows itself in databases.

After the database returns the aggregated result set, the `HAVING` clause filters the result set. In the example, it returns only those aggregated results that have two or more non-unique `item_title` and `item_rating` rows in the table. The `ORDER BY` then sorts the return set.



The GROUP BY must refer to all non-aggregated columns returned by the SELECT clause, and it must exclude column aliases.

```
3 SELECT   i.item_title  AS title
  ,        i.item_rating AS rating
  ,        COUNT(*) AS copies
1 FROM     item i
2 WHERE    i.item_rating =  'PG'
  AND      i.item_title LIKE 'Star Wars%'
4 GROUP BY i.item_title
  ,        i.item_rating
5 HAVING   COUNT(*) > 1
6 ORDER BY 1, 2;
```

The HAVING clause acts like the WHERE clause but applies to aggregated result sets.

FIGURE 11-5. *Order of operation on aggregate queries*

Oracle's ROWID Pseudo Column

Oracle also supports a ROWID pseudo column that is the physical address for the row. If you want, you can substitute the ROWID pseudo column for the asterisk(*) in the COUNT function.

The following list qualifies the ANSI SQL pattern for processing a single table query with aggregation and a GROUP BY and HAVING clause:

- It finds a table in the FROM clause.
- It *optionally* assigns a table alias as a runtime placeholder for the table name.
- It gets the table definition from the data catalog to determine the valid column names.
- If a table alias is present (and it is), it optionally maps the alias to the table's data catalog definition.
- It filters rows into the result set based on the value of columns in the WHERE clause.
- The list of columns in the SELECT clause filters the desired columns from the complete set of columns in a row.
- The aggregation function triggers a check for a GROUP BY clause when non-aggregated columns are returned in the SELECT list, and then aggregates results.
- The HAVING operator filters the result set from the aggregation or the GROUP BY aggregation.
- If an ORDER BY clause occurs in the query, rows are sorted by the designated columns.

We'll work through the basic aggregation steps most developers use frequently. They cover the COUNT, SUM, AVERAGE, MAX, and MIN functions. The following discussions use two sets of ordinal and cardinal numbers (some values not displayed to save space) that are stored in the ordinal table, like so:

```
        ID LIST_SET             LIST_NAME  LIST_VALUE
---------- -------------------- ---------- ----------
         1 Value Set A          Zero                0
         2 Value Set A          One                 1
         3 Value Set A          Two                 2
         4 Value Set A          Three               3
         5 Value Set A          Four                4
         6 Value Set A          Five                5
         7 Value Set A          Six                 6
         8 Value Set A          Seven               7
         9 Value Set A          Eight               8
        10 Value Set A          Nine                9
        11 Value Set A
        12 Value Set B          Zero                0
        13 Value Set B          One                 1
        ...
        21 Value Set B          Nine                9
        22 Value Set B
```

You've been exposed to the data set to help you understand how the aggregation functions work in the following subsections.

Aggregate Columns Only

The COUNT function has two behaviors: counting by reference and counting by value. They differ on how they treat null values. You count the number of physical rows when you count by reference, and you count the physical values when you count by value.

The count by reference example counts the number of rows in the ordinal table, like this:

```
SQL> SELECT COUNT(*) AS number_of_rows FROM ordinal;
```

It returns the following:

```
NUMBER_OF_ROWS
--------------
            22
```

The count by value example counts the values in the list_value column. The list_value column contains two null values. The column name is substituted for the asterisk, like this:

```
SQL> SELECT COUNT(list_value) AS number_of_values FROM ordinal;
```

It returns the following:

```
NUMBER_OF_VALUES
----------------
              20
```

The return set is 2 less than the number of rows. That's because the COUNT function doesn't count null values. You can also count all values (which is the default performed in the preceding example) or distinct values only. Both approaches exclude null values.

The following query demonstrates counting using the default, an explicit ALL, and DISTINCT number of values found in the list_name and list_value columns:

```
SQL> SELECT COUNT(list_name) AS default_number
  2  ,      COUNT(ALL list_name) AS explicit_number
  3  ,      COUNT(DISTINCT list_value) AS distinct_number
  4  FROM   ordinal;
```

Here are the results:

```
DEFAULT_NUMBER EXPLICIT_NUMBER DISTINCT_NUMBER
-------------- --------------- ---------------
            20              20              10
```

Notice that the COUNT function returns the same number with or without the ALL keyword. That's because the default is ALL, which is provided when you don't use it. It also counts the occurrences of strings or numbers. You count each individual element when the ALL is specified, not just the unique set of elements. The DISTINCT keyword forces a unique sort of the data set before counting the results.

The `SUM`, `AVG`, `MAX`, and `MIN` functions work only with numbers. The following demonstrates the `SUM` and `AVG` functions against the `list_value` column:

```
SQL> SELECT SUM(ALL list_value) AS sum_all
  2  ,      SUM(DISTINCT list_value) AS sum_distinct
  3  ,      AVG(ALL list_value) AS avg_all
  4  ,      AVG(DISTINCT list_value) AS avg_distinct
  5  FROM   ordinal;
```

Here's the result set:

```
   SUM_ALL SUM_DISTINCT    AVG_ALL AVG_DISTINCT
---------- ------------ ---------- ------------
        90           45        4.5          4.5
```

The sum of two sets of the ordinal numbers is 90, and one set is 45. The average of `ALL` or the `DISTINCT` set is naturally the same.

The next example runs the `MAX` and `MIN` functions:

```
SQL> SELECT MIN(ALL list_value) AS min_all
  2  ,      MIN(DISTINCT list_value) AS min_distinct
  3  ,      MAX(ALL list_value) AS max_all
  4  ,      MAX(DISTINCT list_value) AS max_distinct
  5  FROM   ordinal;
```

It produces these results:

```
   MIN_ALL MIN_DISTINCT    MAX_ALL MAX_DISTINCT
---------- ------------ ---------- ------------
         0            0          9            9
```

The minimum or maximum of two sets of the same group of numbers is always the same. The minimum is 0 and the maximum is 9 for ordinal numbers.

Aggregate and Non-aggregate Columns

The principal of returning aggregate and non-aggregate columns starts with understanding that you get only one row when you add a column of numbers. By extension, you get one row for every type of thing you count. A real-world example of that would be counting a bag of fruit. You separate the fruit into groups, such as apples, oranges, pears, and apricots. Then you count the number of each type of fruit.

The following counts the number of rows and values for each unique value in the `list_set` column:

```
SQL> SELECT   list_set AS grouping_by_column
  2  ,        COUNT(*)
  3  ,        COUNT(list_value)
  4  FROM     ordinal
  5  GROUP BY list_set;
```

And here are the results of this query:

```
GROUPING_BY_COLUMN     COUNT(*) COUNT(LIST_VALUE)
-------------------- ---------- -----------------
Value Set A                  11                10
Value Set B                  11                10
```

The results tells us that we have 11 rows in each group and only 10 values, which means each group has one row that contains a null value. You change the `SELECT` list and `GROUP BY` clause when you want to identify the rows with the null values.

The following query returns a 0 when the `list_value` column is null and a 1 otherwise:

```
SQL> SELECT   list_set AS grouping_by_not_null
  2  ,        list_name AS group_by_null_too
  3  ,        COUNT(*)
  4  ,        COUNT(list_value)
  5  FROM     ordinal
  6  WHERE    list_set = 'Value Set A'
  7  GROUP BY list_set
  8  ,        list_name;
```

And here are the results from the query:

```
GROUPING_BY_NOT_NULL GROUP   COUNT(*) COUNT(LIST_VALUE)
-------------------- ----- ---------- -----------------
Value Set A          Zero           1                 1
Value Set A          Five           1                 1
Value Set A          Three          1                 1
Value Set A          Four           1                 1
Value Set A          One            1                 1
Value Set A          Two            1                 1
Value Set A          Eight          1                 1
Value Set A          Nine           1                 1
Value Set A          Seven          1                 1
Value Set A          Six            1                 1
Value Set A                         1                 0
```

The only problem with the return set is that the cardinal numbers aren't in numeric order. That requires a special `ORDER BY` clause with a `CASE` statement. You could add the following to the last query to get them sorted into numeric order:

```
  9  ORDER BY CASE
 10             WHEN list_name = 'Zero'  THEN 0
 11             WHEN list_name = 'One'   THEN 1
 12             WHEN list_name = 'Two'   THEN 2
 13             WHEN list_name = 'Three' THEN 3
 14             WHEN list_name = 'Four'  THEN 4
 15             WHEN list_name = 'Five'  THEN 5
 16             WHEN list_name = 'Six'   THEN 6
 17             WHEN list_name = 'Seven' THEN 7
 18             WHEN list_name = 'Eight' THEN 8
 19             WHEN list_name = 'Nine'  THEN 9
 20           END;
```

This type of `ORDER BY` clause lets you achieve numeric ordering without changing any of the data. Note that null values are always sorted last in an ascending sort and first in a descending sort.

NOTE
Ascending sorts put nulls last while descending sorts put nulls first.

The following query demonstrates the `GROUP BY` for the `SUM`, `AVG`, `MAX`, and `MIN` functions:

```
SQL> SELECT   list_set AS grouping_by_not_null
  2  ,        SUM(list_value) AS ordinal_sum
  3  ,        AVG(list_value) AS ordinal_avg
  4  ,        MIN(list_value) AS ordinal_min
  5  ,        MAX(list_value) AS ordinal_max
  6  FROM     ordinal
  7  GROUP BY list_set;
```

It displays the following:

```
GROUPING_BY_NOT_NULL ORDINAL_SUM ORDINAL_AVG ORDINAL_MIN ORDINAL_MAX
-------------------- ----------- ----------- ----------- -----------
Value Set A                   45         4.5           0           9
Value Set B                   45         4.5           0           9
```

This returns the expected result set from the functions. They naturally match for each set of ordinal numbers. If you were to alter the data set, you could get different results.

Queries that Return Columns or Results Selectively

Queries that return columns or results selectively depend on conditional logic. Originally, SQL wasn't designed to contain any conditional logic. Oracle introduced conditional logic as an extension to the definition in the 1980s. It implemented the `DECODE` function. Other vendors followed Oracle's lead, and MySQL implements the `IF` function. Finally, the ANSI SQL definition added the `CASE` operator, which you saw illustrated in an `ORDER BY` clause in the preceding section.

You need to understand the different proprietary solutions and the `CASE` operator before you see an exhibit of selective aggregation. The following sections discuss the Oracle proprietary `DECODE` function, the MySQL `IF` function, and the `CASE` statement before discussing selective aggregation.

Oracle Proprietary DECODE Statement

The `DECODE` function allows you to perform if-then-else logic on equality matches. It doesn't support inequalities except as the else condition of an equality comparison. You can nest `DECODE` functions as call parameters to a `DECODE` function. The `DECODE` function is not portable. As a best practice, avoid the `DECODE` function. As a reality check, millions of lines of code use the `DECODE` function, which makes its coverage essential and learning it unavoidable.

Here's the prototype for the `DECODE` function:

```
DECODE(expression, search, result
                [, search, result [, ... ]]
                [, default])
```

The function requires at least an expression or column return value, a search value, and a result before an optional default value. You can also add any number of pairs of search values and results before the optional default value.

Some SQL*Plus formatting is necessary to get a clean test in Oracle. You can check back to Chapter 2 for instructions on these formatting commands:

```
-- Set null to a visible string.
SET NULL "<Null>"
-- Set column formatting for an alphanumeric string.
COLUMN "Test 1" FORMAT A9
COLUMN "Test 2" FORMAT A9
COLUMN "Test 3" FORMAT A9
COLUMN "Test 4" FORMAT A9
COLUMN "Test 5" FORMAT A9
COLUMN "Test 6" FORMAT A9
COLUMN "Test 7" FORMAT A9
```

The following single test case covers the outcome possibilities of a `DECODE` function:

```
SQL> SELECT   DECODE('One','One','Equal') AS "Test 1"
  2  ,        DECODE('One','Two','Equal') AS "Test 2"
  3  ,        DECODE('One','One','Equal','Not Equal') AS "Test 3"
  4  ,        DECODE('One','Two','Equal','Not Equal') AS "Test 4"
  5  ,        DECODE('One','Two','Equal'
  6                        ,'Three','Equal') AS "Test 5"
  7  ,        DECODE('One','Two','Equal'
  8                        ,'Three','Equal','Not Equal') AS "Test 6"
  9  ,        DECODE('One','Two','Equal'
 10                        ,'Three','Equal'
 11                        ,'One','Equal','Not Equal') AS "Test 7"
 12  FROM     dual;
```

The query returns the following results:

```
Test 1    Test 2    Test 3    Test 4    Test 5    Test 6    Test 7
--------- --------- --------- --------- --------- --------- ---------
Equal     <Null>    Equal     Not Equal <Null>    Not Equal Equal
```

Tests 1 and 2 These tests use the `DECODE` function as an if-then block of code. The first call parameter to the `DECODE` function is what you're trying to match with the second call parameter. When they match, the third call parameter is returned. When they fail to match, the null value is returned.

TIP
The mnemonic for an if-then block is three call parameters to the `DECODE` function.

This behavior is handy when you're counting success in a selective aggregation model, because the following would count only individuals who are single:

```
SQL> WITH inline AS
  2  (SELECT 'Single' AS marital_status FROM dual
  3   UNION ALL
  4   SELECT 'Single' AS marital_status FROM dual
  5   UNION ALL
  6   SELECT 'Married' AS marital_status FROM dual)
  7  SELECT COUNT(DECODE(inline.marital_status,'Single',1)) AS Single
  8  FROM   inline;
```

It returns the number of single rows:

```
    SINGLE
----------
         2
```

This demonstrates that it selectively counts results based on matches and nonmatches in the fabricated table of three rows. What it can't do is provide the count of nonmatches.

Tests 3 and 4 These tests use the `DECODE` function as an if-then-else block of code. The first, second, and third call parameters work like tests 1 and 2. The thing that's different with an `else` condition is that something meaningful is returned when the first call argument fails to match the second call argument.

NOTE
The mnemonic for an if-then-else block is four call parameters to the `DECODE` function.

This behavior is even better than the results from tests 1 and 2 for selective aggregation. It lets you count matches and nonmatches by using two columns instead of one in the `SELECT` list. You can count the single and married rows, like so:

```
SQL> WITH inline AS
  2  (SELECT 'Single' AS marital_status FROM dual
  3   UNION ALL
  4   SELECT 'Single' AS marital_status FROM dual
  5   UNION ALL
  6   SELECT 'Married' AS marital_status FROM dual)
  7  SELECT COUNT(DECODE(inline.marital_status,'Single',1)) AS Single
  8  ,      COUNT(DECODE(inline.marital_status,'Married',1)) AS Married
  9  FROM   inline;
```

The two columns now give us this more meaningful result:

```
    SINGLE    MARRIED
---------- ----------
         2          1
```

This demonstrates that you can ask two sides of the same question by putting the results in separate columns. This is a form of SQL transformation.

Tests 5, 6, and 7 These tests use the DECODE function as if-then-elseif and if-then-elseif-else blocks of code. They act more or less like traditional switch statements, except Oracle doesn't support fall-through behavior. Fall-through behavior would return a result for the first and every subsequent case statement where the conditions were met.

The first, second, and third call parameters work like tests 1 through 4, but call parameters 4 and 5, 6 and 7, 8 and 9, and so forth would be cases. The lack of an even number of parameters means there's no default case. That means an odd number of parameters 5 and above make a DECODE function into a switch without a default condition. Likewise, an even number of parameters 6 and above make a DECODE function into a switch with a default condition.

NOTE
The mnemonic for an if-then-elseif block is five or any greater odd number of call parameters, and the mnemonic for an if-then-elseif-else block is six or any greater even number of call parameters to the DECODE function.

This behavior allows us to test more than just two cases. The downside is that quickly it becomes verbose. You can now count the single, divorced, and married rows, like so:

```
SQL> WITH inline AS
  2  (SELECT 'Single' AS marital_status FROM dual
  3   UNION ALL
  4   SELECT 'Single' AS marital_status FROM dual
  5   UNION ALL
  6   SELECT 'Divorced' AS marital_status FROM dual
  7   UNION ALL
  8   SELECT 'Annulled' AS marital_status FROM dual
  9   UNION ALL
 10   SELECT 'Married' AS marital_status FROM dual)
 11  SELECT COUNT(DECODE(inline.marital_status,'Single',1)) AS Single
 12  ,      COUNT(DECODE(inline.marital_status,'Divorced',1)) AS Divorced
 13  ,      COUNT(DECODE(inline.marital_status,'Married',1)) AS Married
 14  FROM   inline;
```

The data set now has four classifications in the in-line view but only three evaluations in the SELECT list. Any person whose marital status is annulled is excluded from your SQL report. The results would be the following:

```
    SINGLE   DIVORCED    MARRIED
---------- ---------- ----------
         2          1          1
```

You could capture the annulled marriages by some math operation, such as the following, with the DECODE function:

```
 11  SELECT COUNT(DECODE(inline.marital_status,'Single',1)) AS Single
 12  ,      COUNT(DECODE(inline.marital_status,'Divorced',1)) AS Divorced
 13  ,      COUNT(DECODE(inline.marital_status,'Married',1)) AS Married
 14  ,      COUNT(inline.marital_status) -
 15           (COUNT(DECODE(inline.marital_status,'Single',1)) +
             COUNT(DECODE(inline.marital_status,'Divorced',1)) +
             COUNT(DECODE(inline.marital_status,'Married',1))) AS Other
    FROM   inline
```

It would now give you this:

```
    SINGLE   DIVORCED    MARRIED      OTHER
---------- ---------- ---------- ----------
         2          1          1          1
```

This is the limit of what you can do with the DECODE function. Next, you'll examine the MySQL IF function, which doesn't include a case element.

MySQL Proprietary IF Function

The IF function allows you to perform an if-then-else logic based on equality, inequality, range, and element in set comparisons. As conditional functions go, the MySQL IF function is superior to the Oracle DECODE function, because the IF function isn't limited to equality comparisons only.

Here's the prototype for the IF function:

```
IF(comparison_expression, expression_when_true, expression_when_false)
```

The IF function evaluates whether a comparison expression is true or false. The IF function processes and returns the second expression when the comparison expression is true and the third expression when it isn't true. You can nest IF functions in all IF function expressions. The expressions returned for true and false must be returned as strings, floating point, or integer numbers.

The following subsections qualify the most frequent type of use cases with the IF function.

Numeric Comparisons Numeric comparisons are the easiest place to start. Here's a query that tests the most likely numeric comparisons you would use:

```
SELECT IF((1 = 1),"E: (1 = 1)","NE:(1 = 1)") AS comparison
UNION ALL
SELECT IF((1 = 1.0),"E: (1 = 1.0)","NE:(1 = 1.0)") AS comparison
UNION ALL
SELECT IF((1 = 1.1),"E: (1 = 1.1)","NE:(1 = 1.1)") AS comparison
UNION ALL
SELECT IF((1 > 1),"E: (1 > 1)","NE:(1 > 1)") AS comparison
UNION ALL
SELECT IF((1 <> 2),"E: (1 <> 2)","NE:(1 <> 2)") AS comparison;
```

The output of these comparisons is in the following result set, where *E* means equal and *NE* means not equal:

```
+--------------+
| comparison   |
+--------------+
| E: (1 = 1)   |
| E: (1 = 1.0) |
| NE:(1 = 1.1) |
| NE:(1 > 1)   |
| E: (1 <> 2)  |
+--------------+
```

The comparison of two integers of equal values is a true scenario. The comparison of an integer and decimal is true when the decimal casts as an integer of equal value. The comparison of an integer to a decimal is false when the decimal integer isn't equal to the decimal value. The inequality comparison is false because the integers hold the same values. The not equal comparison confirms that 1 is not equal to 2.

String Comparisons String comparisons are straightforward if you come from a MySQL background, but they can be a bit confusing when you come from an Oracle background. That's because MySQL uses case-insensitive comparisons by default, while Oracle uses a case-sensitive comparison.

The following `SELECT` statement illustrates how you can perform case-insensitive and case-sensitive comparisons:

```
SELECT   IF(('One' = 'one')
           ,"E: ('One' = 'one')"
           ,"NE:('One' = 'one')") AS string_comparison
       , IF((BINARY 'One' = 'one')
            ,"E: (BINARY 'One' = 'one')"
            ,"NE:(BINARY 'One' = 'one')") AS binary_comparison; comparisons:
```

The first column performs a case-insensitive comparison while the second column performs a binary comparison. Binary comparisons are case-sensitive. Here are the results of the query:

```
+--------------------+---------------------------+
| string_comparison  | binary_comparison         |
+--------------------+---------------------------+
| E: ('One' = 'one') | NE:(BINARY 'One' = 'one') |
+--------------------+---------------------------+
```

NOTE
A string of "One" is less than a string of "one," because uppercase letters have a lower ASCII binary value than their lowercase equivalents—the opposite of what you might expect.

Range Comparisons Range comparisons work with the `BETWEEN` operator. You can perform range comparisons against dates, numbers, or strings. The first column of the range comparison example compares whether the `UTC_DATE` function value is between two literal values (true at the time of writing). The second column uses a `NOT BETWEEN` operator to demonstrate a false result.

```
SELECT
 IF((UTC_DATE() BETWEEN '2011-05-05' AND '2011-07-04')
   ,"T:(UTC_DATE() BETWEEN date1 AND date2)"
   ,"F:(UTC_DATE() NOT BETWEEN date1 AND date2)") AS in_range
,IF((UTC_DATE() NOT BETWEEN '2011-05-05' AND '2011-07-04')
   ,"T:(UTC_DATE() NOT BETWEEN date1 AND date2)"
   ,"F:(UTC_DATE() BETWEEN date1 AND date2)") AS out_of_range\G
```

Notice the odd formatting? It's been changed here to fit on the page. The `\G` execution puts column labels to the left and results to the right. Here are the results of this query:

```
************************* 1. row *************************
    in_range: T: (UTC_DATE() BETWEEN date1 AND date2)
out_of_range: F: (UTC_DATE() BETWEEN date1 AND date2)
```

The `UTC_DATE` function returned must either be inside or outside the range. You always evaluate true first and false second. The first column returns true because the value is found within the range and the second column returns false because the value is not outside of the range. More or less, the first column's truth is whether the value is between the others and the second column's truth is whether it isn't.

In-set Comparisons Determining whether a value is within a set is also possible with the `IF` function. The following demonstrates an in-set and not in-set comparison operation by performing one against strings and the other against binary strings:

```
SELECT
  IF(('bat' IN ('BAT','BALL'))
    ,"T: ('bat' IN ('BAT','BALL'))"
    ,"F: ('bat' NOT IN ('BAT','BALL'))") AS in_set
, IF((BINARY 'bat' NOT IN ('BAT','BALL'))
    ,"T: (BINARY 'bat' NOT IN ('BAT','BALL'))"
    ,"F: (BINARY 'bat' IN ('BAT','BALL'))") AS not_in_set\G
```

The results confirm what we would suspect—the case-insensitive string is found in the set while the case-sensitive (binary) string isn't:

```
************************* 1. row *************************
    in_set: T: ('bat' IN ('BAT','BALL'))
not_in_set: T: (BINARY 'bat' NOT IN ('BAT','BALL'))
```

The only problem with this type of comparison is that you can make it only against literal values or other columns returned in the same row. MySQL offers another possibility. The following query returns In stock if the subquery finds a match, and Out of stock when it doesn't:

```
SELECT
 IF('Star Wars VII' IN (SELECT item_title FROM item)
   ,'In stock','Out of stock') AS data_inlieu_no_row_found;
```

returns the following:

```
+--------------------------+
| data_inlieu_no_row_found |
+--------------------------+
| Out of stock             |
+--------------------------+
```

You would get an empty set if you rewrote the query using a `WHERE` clause that checks whether the literal value equals any value in the `item_title` column. The foregoing query gives you a descriptive result whether or not it's found.

ANSI SQL CASE Operator

The `CASE` operator is the most portable, and it allows for equality and inequality evaluation, range comparisons, and in-set comparisons. It also supports multiple `CASE` statements, such as a switch statement without fall-through characteristics. You can likewise use comparisons against subqueries and correlated subqueries.

In an Oracle database, the following query matches case-insensitive strings from the in-line view against string literals for the primary colors on the color wheel:

```
SQL> SELECT   inline.color_name
  2  ,        CASE
  3             WHEN UPPER(inline.color_name) = 'BLUE' THEN
  4               'Primary Color'
  5             WHEN UPPER(inline.color_name) = 'RED' THEN
  6               'Primary Color'
  7             WHEN UPPER(inline.color_name) = 'YELLOW' THEN
  8               'Primary Color'
  9             ELSE
 10               'Not Primary Color'
 11           END AS color_type
 12  FROM    (SELECT 'Red' AS color_name FROM dual
 13           UNION ALL
 14           SELECT 'Blue' AS color_name FROM dual
 15           UNION ALL
 16           SELECT 'Purple' AS color_name FROM dual
 17           UNION ALL
 18           SELECT 'Green' AS color_name FROM dual
 19           UNION ALL
 20           SELECT 'Yellow' AS color_name FROM dual) inline
 21  ORDER BY 2 DESC, 1 ASC
```

This query works in MySQL whether you leave it as is or modify it by removing the `UPPER` function around the `inline.color_name` column. The `CASE` operator includes several `WHEN` clauses that evaluate conditions and an `ELSE` clause that acts as the default catchall for the `CASE` operator. Note that `END` by itself terminates a `CASE` operator. If you were to put `END CASE`, the word *`CASE`* would become the column alias.

Although the sample evaluates only a single logical condition, each `WHEN` clause supports any number of `AND` or `OR` logical operators. Any comparison phrase can use the standard equality and inequality comparison operators; the `IN`, `=ANY`, `=SOME`, and `=ALL` lookup operators; and scalar, single-row, multiple-row, and correlated subqueries.

You would get the following results from the preceding query in Oracle—at least you would when you format the `color_name` column to an alphanumeric 10 character string in SQL*Plus (check Chapter 2 for syntax):

```
COLOR_NAME COLOR_TYPE
---------- -----------------
Blue       Primary Color
Red        Primary Color
Yellow     Primary Color
Green      Not Primary Color
Purple     Not Primary Color
```

There's a lot of power in using the CASE operator, but you need to understand the basics and experiment. For example, you saw how to get an in-stock or out-of-stock answer with the IF function in MySQL. You can rewrite that query with a CASE operator that runs in an Oracle or MySQL database, and it would look like this:

```
SELECT CASE
         WHEN 'Star Wars VII' IN (SELECT item_title FROM item)
         THEN 'In-stock'
         ELSE 'Out-of-stock'
       END AS yes_no_answer
FROM   dual;
```

The foregoing query is limited, but it mirrors the MySQL query that used the IF function. A more useful approach that works in Oracle or MySQL databases is the following query with the CASE operator. It uses an in-line view to fabricate a data set that's used as the decision-making parameter to the WHEN clause in the CASE operator.

```
SELECT inline.query_string
,      CASE
         WHEN inline.query_string IN (SELECT item_title FROM item)
         THEN 'In-stock'
         ELSE 'Out-of-stock'
       END AS yes_no_answer
FROM  (SELECT 'Star Wars II' AS query_string FROM dual
       UNION ALL
       SELECT 'Star Wars VII' AS query_string FROM dual) inline;
```

It returns the following output from a MySQL database:

```
+---------------+---------------+
| query_string  | yes_no_answer |
+---------------+---------------+
| Star Wars II  | In-stock      |
| Star Wars VII | Out-of-stock  |
+---------------+---------------+
```

The CASE operator also allows you to validate complex math or date math. Date math isn't very transferable between an Oracle and MySQL database, because of the differences between their implementations. The following subsections explain Oracle and MySQL date math and provide a CASE statement that leverages date math in each database.

Oracle Date Math Oracle's date math is very straightforward. You simply add or subtract numbers from a date to get a date in the future or past, respectively. The only twist in the model is that the DATE data type is a date-time not a date. You can shave off the hours and minutes of any day with the TRUNC function and make the date-time equivalent to midnight the morning of a date. This is the closest you have to a true DATE data type in an Oracle database.

The following example looks at yesterday, today, and tomorrow:

```
SQL> SELECT   SYSDATE - 1 AS yesterday
  2  ,        SYSDATE     AS today
  3  ,        SYSDATE + 1 AS tomorrow
  4  FROM     dual;
```

The results are deceiving, because Oracle automatically prints them as dates, like so:

```
YESTERDAY TODAY     TOMORROW
--------- --------- ---------
19-JUN-11 20-JUN-11 21-JUN-11
```

If we convert the date-time values to strings with formatting instructions down to the second, you would see the full date-time stamp. This query uses the Oracle proprietary `TO_CHAR` function to do that:

```
SQL> SELECT   TO_CHAR(SYSDATE - 1,'DD-MON-YYYY HH24:MI:SS') AS Yesterday
  2  ,        TO_CHAR(SYSDATE    ,'DD-MON-YYYY HH24:MI:SS') AS Today
  3  ,        TO_CHAR(SYSDATE + 1,'DD-MON-YYYY HH24:MI:SS') AS Tomorrow
  4  FROM     dual;
```

It yields the following:

```
 YESTERDAY            TODAY                TOMORROW
-------------------- -------------------- --------------------
19-JUN-2011 22:59:45 20-JUN-2011 22:59:45 21-JUN-2011 22:59:45
```

You could use the `TRUNC` function to shave the decimal portion of time, which would give you 12 midnight in the morning of each day. The following query truncates the time from the `SYSDATE` value:

```
SQL> SELECT   TO_CHAR(TRUNC(SYSDATE)-1,'DD-MON-YYYY HH24:MI') AS Yesterday
  2  ,        TO_CHAR(TRUNC(SYSDATE)  ,'DD-MON-YYYY HH24:MI:SS') AS Today
  3  ,        TO_CHAR(TRUNC(SYSDATE)+1,'DD-MON-YYYY HH24:MI:SS') AS Tomorrow
  4  FROM     dual;
```

The results show that everything is now 12 midnight the morning of each day:

```
YESTERDAY         TODAY                TOMORROW
----------------- -------------------- --------------------
19-JUN-2011 00:00 20-JUN-2011 00:00:00 21-JUN-2011 00:00:00
```

This means any date plus an integer of 1 yields a day that is 24 hours in the future and any date minus an integer of 1 yields a day that is 24 hours behind the current date-time value. The `TRUNC` function also lets you get the first day of a month or the first day of a year. It works like this:

```
SQL> SELECT   TRUNC(SYSDATE,'MM') AS first_day_of_month
  2  ,        TRUNC(SYSDATE,'YY') AS first_day_of_year
  3  FROM     dual;
```

Here are the results for any day in June:

```
FIRST_DAY_OF_MONTH FIRST_DAY_OF_YEAR
------------------ ------------------
01-JUN-11          01-JAN-11
```

If you subtract two days, you get the number of days between them, like so:

```
SQL> SELECT TO_DATE('30-MAY-2011') - TO_DATE('14-FEB-2011') AS days
  2  FROM dual;
```

This query would tell us the number of days between Valentine's Day and Memorial Day, as shown here:

```
      DAYS
----------
       105
```

Although you can subtract days, you can't add them. If you tried to add dates, the following error would be raised:

```
SELECT TO_DATE('30-MAY-2011') + TO_DATE('14-FEB-2011')
                              *
ERROR at line 1:
ORA-00975: date + date not allowed
```

Table 11-1 provides additional built-in functions that can help when you're performing date math on an Oracle database. Although the table's not inclusive of timestamp functions, it covers those functions that work with dates. You can check the *Oracle Database SQL Reference* for more information on the timestamp built-ins.

Date Function	Description

`ADD_MONTHS`	Lets you add or subtract months, like so:
	`SELECT ADD_MONTHS(SYSDATE, 3)`
	`FROM dual;`

`CAST`	Lets you convert a string that uses the Oracle default date format masks of `DD-MON-RR` or `DD-MON-YYYY` to a `DATE` data type. The example uses an `INSERT` statement to show the conversion:
	`INSERT INTO` *`some_table`*
	`VALUES`
	`(CAST('15-APR-11' AS DATE));`

`CURRENT_DATE`	Finds the current system date: `SELECT CURRENT_DATE FROM dual;`

`GREATEST`	Finds the most forward date in a set of dates. It works like this to find tomorrow:
	`SELECT GREATEST(SYSDATE,SYSDATE + 1)`
	`FROM dual;`

`EXTRACT`	Lets you extract the integer that represents a year, month, day, hour, minute, or second from a `DATE` data type. The following prototypes show you how to grab the day, month, or year from a `DATE` data type:
	`SELECT EXTRACT(DAY FROM SYSDATE) AS dd`
	`,      EXTRACT(MONTH FROM SYSDATE) AS mm`
	`,      EXTRACT(YEAR FROM SYSDATE) AS yy`
	`FROM dual;`

`LEAST`	Finds the most forward date in a set of dates. It works like this to find yesterday:
	`SELECT LEAST(SYSDATE -1,SYSDATE)`
	`FROM dual;`

TABLE 11-1. *Oracle Built-in Date Functions*

Date Function	Description

`LAST_DAY`	Lets you find the last date of the month for any date, like so:
	`SELECT last_day(SYSDATE)`
	`FROM dual;`

`LEAST`	Finds the most forward date in a set of dates. It works like this to find yesterday:
	`SELECT LEAST(SYSDATE -1,SYSDATE)`
	`FROM dual;`

`MONTHS_BETWEEN`	Lets you find the decimal value between two dates. The function returns a positive number when the greater date is the first call parameter and a negative number when it's the second call parameter. Here's an example:
	`SELECT MONTHS_BETWEEN('25-DEC-11',SYSDATE)`
	`FROM dual;`

`NEXT_DAY`	Lets you find the date of the next day of the week, like so:
	`SELECT next_day(SYSDATE,'FRIDAY')`
	`FROM dual;`

`ROUND`	Shaves off the decimal portion of a `DATE` when the current date-time is before noon. Alternatively, it adds the complement of the decimal to make the day midnight of the next day. Use it like this: `SELECT ROUND(SYSDATE) FROM dual;`
`SYSDATE`	Finds the current system date: `SELECT SYSDATE FROM dual;`

`TO_CHAR`	Lets you apply a format to a date, like so:
	`SELECT`
	`  TO_CHAR(SYSDATE,'DD-MON-YYYY HH24:MI:SS')`
	`FROM dual;`
	Supports the following format syntax:
	`DD` – Two-digit day
	`MM` – Two-digit month
	`MON` – Three-character month, based on `NLS_LANG` value
	`YY` – Two-digit year
	`YYYY` – Two-digit absolute year
	`RR` – Two-digit relative year
	`HH` – Two-digit hour, values 1 to 12
	`HH24` – Two-digit hour, values 0 to 23
	`MI` – Two-digit minutes, values 0 to 59
	`SS` – Two-digit seconds, values 0 to 59

`TO_DATE`	Converts a string to a `DATE` data type. Lets you to convert nonstandard `DATE` format masks, which come from external import sources. The default format masks of `DD-MON-RR` and `DD-MON-YYYY` also work with the `TO_DATE` function but can be cast to a `DATE` with the `CAST` function. The `TO_DATE` function would convert a MySQL default format date string to a `DATE` with this syntax: `SELECT TO_DATE('2011-07-14','YYYY-MM-DD') FROM dual;`

TABLE 11-1. *Oracle Built-in Date Functions* (continued)

The EXTRACT function works in both Oracle and MySQL databases. Blending the conditional CASE operator with the EXTRACT date function, you can write a statement that finds transaction_amount values for a given month. The following is such a statement:

```
SQL> SELECT  CASE
  2            WHEN EXTRACT(MONTH FROM transaction_date) = 1 AND
  3                 EXTRACT(YEAR FROM transaction_date) = 2011 THEN
  4              transaction_amount
  5          END AS "January"
  6  FROM    transaction;
```

Lines 2 and 3 identify transaction amounts from a month and year, and only transaction_amount values for January 2011 would be returned by the query. However, it would also return null value rows for every other month and year present in the table. You need to filter the query with a WHERE clause to restrict it to the data of interest. The CASE operator in a SELECT list works in tandem with the WHERE clause of a query.

Another approach could have you return a Y as an active_flag value when the item_release_date values are less than or equal to 30 days before today, and an N when they're greater than 30 days. You would write that CASE operator like this:

```
SQL> SELECT CASE
  2           WHEN (SYSDATE - i.release_date) <= 30 THEN 'Y'
  3           WHEN (SYSDATE - i.release_date) >  30 THEN 'N'
  4         END AS active_flag
  5  FROM   item i;
```

In conclusion, the CASE operator is more flexible than the DECODE or IF function. The best practice is to write portable code with the CASE operator.

MySQL Date Math MySQL's date math is very different from Oracle's approach. You can't add or subtract numbers from dates. You must use designated functions. The fact that some functions have aliases makes the list of functions longer. My recommendation is that you choose the function or its alias and use it consistently, such as the DATE_ADD function or its synonym ADDDATE.

The DATE_ADD function has the following prototype:

```
DATE_ADD(date, INTERVAL number date_interval)
```

The UTC_DATE function returns the current date in MySQL, and you can add 6 days to it like this:

```
SELECT DATE_ADD(UTC_DATE(), INTERVAL 6 DAY);
```

Alternatively, you can add a week by using a different interval data type, like so:

```
SELECT DATE_ADD(UTC_DATE(), INTERVAL 1 WEEK);
```

Subtracting units from dates works the same way as adding them. For example, you would subtract 6 days like this:

```
SELECT DATE_SUB(UTC_DATE(), INTERVAL 6 DAY);
```

And you could subtract a week like this:

```
SELECT DATE_SUB(UTC_DATE(), INTERVAL 1 WEEK);
```

Table 11-2 shows the available intervals and formats for adding to or subtracting from dates. Some intervals in the table are integers and others are strings. The strings are enclosed in single quotes. For clarity, a data type column qualifies which are integers or strings. The format masks support the function's internal parser.

The difference between two dates is numeric, as in Oracle. You use the `DATEDIFF` function to calculate it. Here's an example that returns the number of days between Memorial Day and Valentine's Day:

```
SELECT DATEDIFF('2011-05-30','2011-02-14');
```

It returns 105, which is the same as would be returned by Oracle's calculation. Date math functions work solidly in both implementations. Table 11-3 qualifies the date functions you will most likely use, and you can find the date-time function in the *MySQL 5.5 Reference Manual* in Chapter 11.7, "Date and Time Functions."

Interval	**Data Type**	**Format Mask**
DAY	Integer	Dd
DAY_HOUR	String	'dd hh'
DAY_MICROSECOND	String	'dd.nn'
DAY_MINUTE	String	'dd hh:mm'
DAY_SECOND	String	'dd hh:mm:ss.nn'
HOUR	Integer	Hh
HOUR_MICROSECOND	String	'dd hh'
HOUR_MINUTE	String	'dd nn'
HOUR_SECOND	String	'dd hh:mm'
MICROSECOND	Integer	Nn
MINUTE	Integer	Mm
MINUTE_MICROSECOND	String	'mm.nn'
MINUTE_SECOND	String	'mm:ss'
MONTH	Integer	Mm
QUARTER	Integer	Qq
SECOND	Integer	Ss
SECOND_MICROSECOND	String	'ss.nn'
WEEK	Integer	Ww
YEAR	Integer	Yy
YEAR_MONTH	String	'yy-mm'

TABLE 11-2. *DATE_ADD and DATE_SUB Interval Values*

Date Function	Description
`ADDDATE`	An alias for the `DATE_ADD` function that lets you add an interval to a date and returns a new date, like so: `SELECT ADDDATE(UTC_DATE(), INTERVAL 1 DAY);`
`CAST`	Lets you convert a string that uses the Oracle default date format masks of `YYYY-MM-DD` to a `DATE` data type. This example uses an `INSERT` statement to show the conversion: `INSERT INTO some_table` `VALUES` `(CAST('2011-04-15' AS DATE));`
`CURDATE`	Finds the current system date and is synonymous with the `CURRENT_DATE` function: `SELECT CURDATE();`
`CURRENT_DATE`	Finds the current system date and is synonymous with the `CURDATE` function: `SELECT CURRENT_DATE();`
`DATE`	Creates a `DATE` from a default date or date-time string. The defaults are `YYYY-MM-DD` for a date and `YYYY-MM-DD HH:MI:SS` for a date-time. It works like this with a date format: `SELECT DATE('2011-07-04');` It works like this with a date-time format: `SELECT DATE('2011-07-04 06:00:01');`
`DATE_ADD`	The function called when you use the `ADDDATE` synonym, which lets you add an interval to a date and returns a new date. It uses the intervals qualified in Table 11-2, like this: `SELECT DATE_ADD(UTC_DATE(), INTERVAL 1 DAY);`
`DATE_FORMAT`	Lets you apply a format to a date, like so: `SELECT DATE_FORMAT(UTC_DATE(),'%D %M %Y');` It would return `+------------------------------------+` `| DATE_FORMAT(UTC_DATE(),'%D %M %Y') |` `+------------------------------------+` `| 21st June 2011                     |` `+------------------------------------+` This function supports the following format syntax: `%b` – Abbreviated month name; `%c` – Two-digit month; `%d` – Two-digit day, where a zero precedes a single-digit day; `%D` – One- or two-digit day with English suffix: 1st, 2nd; `%e` – One- or two-digit day; `%h` – Two-digit hour, values 1 to 12; `%H` – Two-digit hour, values 0 to 23; `%i` – Two-digit minutes, values 0 to 59; `%I` – Two-digit hour, values 1 to 12; `%j` – Three-digit day of the year, values 001 to 366; `%k` – Two-digit hour, values 0 to 23; `%l` – Two-digit hour, values 1 to 12; `%m` – Two-digit month, values 1 to 12; `%M` – Full month name; `%p` – A.M. or P.M. designation; `%s` – Two-digit seconds, values 0 to 59; `%S` – Two-digit seconds, values 0 to 59; `%y` – Two-digit year; `%Y` – Four-digit year

TABLE 11-3. *MySQL Built-in Date Functions*

Date Function	Description
`DATE_SUB`	Lets you subtract an interval from a date and returns a new date. It uses the intervals qualified in Table 11-2, like so: `SELECT DATE_SUB(UTC_DATE(), INTERVAL 1 DAY);`
`DATEDIFF`	Lets you subtract one date from another. When the first date is greater than the second date, it returns a positive number. When the first date is less than the second date, it returns a negative number. `SELECT DATEDIFF(`*`date1`*`, `*`date2`*`);`
`DAY`	Synonymous with the `DAYOFMONTH` function and returns the two-digit value for the current day: `SELECT DAY(UTC_DATE());`
`DAYNAME`	Returns the language representation for the day of the week, such as Monday, Tuesday, and so on: `SELECT DAYNAME(UTC_DATE());`
`DAYOFMONTH`	Synonymous with the `DAY` function and returns the two-digit value for the current day: `SELECT DAY(UTC_DATE());`
`DAYOFWEEK`	The `DAYOFWEEK` function returns the one digit value for the current day of the week, values are 1 to 7. `SELECT DAYOFWEEK(UTC_DATE());`
`DAYOFYEAR`	Returns the one- to three-digit value for the current day of the year; values are 1 to 366. `SELECT DAYOFYEAR(UTC_DATE());`

`EXTRACT`	Lets you extract the integer that represents a year, month, day, hour, minute, or second from a `DATE` data type. The following prototypes show how to grab the day, month, or year from a `DATE` data type:
	`SELECT EXTRACT(DAY FROM UTC_DATE()) AS dd`
	`,      EXTRACT(MONTH FROM UTC_DATE()) AS mm`
	`,      EXTRACT(YEAR FROM UTC_DATE()) AS yy`
	`FROM dual;`

`LAST_DAY`	Lets you find the last date of the month for any date, like so: `SELECT last_day(UTC_DATE());`
`MONTHNAME`	Returns the name of the month: `SELECT MONTHNAME(UTC_DATE());`
`NOW`	Lets you find the current date-time stamp and is equivalent to the Oracle `SYSDATE` function. `SELECT NOW();`

`STR_TO_DATE`	Lets you apply a format to a date string, like so:
	`SELECT DATE_FORMAT('11-07-04','%D %M %Y');`
	Returns this:
	`+-----------------------------------+`
	`| DATE_FORMAT('11-07-04','%D %M %Y') |`
	`+-----------------------------------+`
	`| 4th July 2011                      |`
	`+-----------------------------------+`
	The function uses the same formatting commands as the `DATE_FORMAT` function.

`SUBDATE`	Lets you subtract an interval from a date and returns a new date. It uses the intervals qualified in Table 11-2, like so: `SELECT SUBDATE(UTC_DATE(), INTERVAL 1 DAY);`
`SYSDATE`	Finds the current system date: `SELECT SYSDATE FROM dual;`

TABLE 11-3. *MySQL Built-in Date Functions* (continued)

As mentioned, the EXTRACT function is portable and works the same way in MySQL and Oracle. Other date math must change to reflect the product-specific date math function. For example, in the Oracle section, you saw a CASE operator that calculated when to select an active or inactive flag value. That example compared the number of days between two dates, where the one date was subtracted from the other.

In MySQL, that would require the DATEDIFF function, like this:

```
SQL> SELECT CASE
              WHEN DATEDIFF(UTC_DATE(),i.release_date) <= 30 THEN 'Y'
              WHEN DATEDIFF(UTC_DATE(),i.release_date) >  30 THEN 'N'
            END AS active_flag
     FROM   item i;
```

The Oracle and MySQL date math sections prepare you to see how selective aggregation works. They also provide you with some immediate capabilities in your queries.

Selective Aggregation

Selective aggregation uses conditional logic, similar to the DECODE function, IF function, or CASE operator inside aggregation functions. The conditional decision-making of these if-then-else functions lets you filter what you count, sum, average, or take the maximum or minimum value of. Understanding this is the first step toward transforming data into useful data sets for accountants and other professional data analysts. The second step lets you transform aggregated rows into column values, which you accomplish through column aliases.

The next example demonstrates transforming rows of data into a financial report where the columns represents months, quarters, and year-to-date values and the rows represent account numbers charged for various expenses.

Although the example uses the EXTRACT function, which works the same way both in Oracle and MySQL, the formatting for a report differs. For convenience, they're covered in separate subsections.

Oracle Selective Aggregation The following query returns a column of data for the month of January 2011. Since all numbers are stored as positive values, a nested CASE operator evaluates the transaction_type column in the same row to determine whether to add or subtract the value. Debits are added and credits are subtracted, because this works with an asset account.

```
SQL> SELECT   t.transaction_account AS "Transaction"
               LPAD(TO_CHAR
              (SUM
                (CASE
                   WHEN EXTRACT(MONTH FROM transaction_date) = 1 AND
                        EXTRACT(YEAR FROM transaction_date) = 2011 THEN
                     CASE
                       WHEN t.transaction_type = 'DEBIT' THEN
                         t.transaction_amount
                       ELSE
                         t.transaction_amount * -1
                     END
                 END),'99,999.00'),10,' ') AS "JAN"
     FROM     transaction t
     GROUP BY t.transaction_account;
```

You would see something like this when you expand beyond a single column, which has been limited to the first three months of the year and two rows of data:

```
Transaction     Jan        Feb        Mar
--------------- ---------- ---------- ----------
10-12-551234      2,671.20   4,270.74   5,371.02
10-14-551234       -690.06  -1,055.76  -1,405.56
```

The value passed to the SUM function is all rows that meet the selectivity criteria. The criteria are the result of a CASE and nested CASE operator. The business result is the total expense grouped by the transaction account number. The LPAD function right-aligns the string returned by the TO_CHAR function, which formats the number always to have two decimal places, even when it's zero cents.

Line 5 could be replaced by the following to capture a first quarter result:

```
  5  WHEN EXTRACT(MONTH FROM transaction_date) IN (1,2,3) AND
```

Moreover, you could generate a year-to-date or year-end report for a completed year by eliminating the validation criterion for the month.

MySQL Selective Aggregation The following query returns the same data as the preceding Oracle equivalent. The differences are in how the values are converted to strings and formatted for output.

```
mysql> SELECT   t.transaction_account AS "Transaction"
  2 ->           LPAD(FORMAT
  3 ->          (SUM
  4 ->            (CASE
  5 ->               WHEN EXTRACT(MONTH FROM transaction_date) = 1 AND
  6 ->                    EXTRACT(YEAR FROM transaction_date) = 2011 THEN
  7 ->                 CASE
  8 ->                   WHEN t.transaction_type = 'DEBIT' THEN
  9 ->                     t.transaction_amount
 10 ->                   ELSE
 11 ->                     t.transaction_amount * -1
 12 ->                 END
 13 ->             END),2),10,' ') AS "JAN"
 14 -> FROM     transaction t
 15 -> GROUP BY t.transaction_account;
```

The output for the first three months would look like this:

```
+--------------+------------+------------+------------+
| Transaction  | Jan        | Feb        | Mar        |
+--------------+------------+------------+------------+
| 10-12-551234 |   2,957.40 |   4,022.70 |   5,654.04 |
| 10-14-551234 |    -750.48 |    -992.16 |  -1,437.36 |
+--------------+------------+------------+------------+
```

Like the Oracle example, the selectivity occurs in the value returned to the SUM function. While the LPAD function works the same way in MySQL, the FORMAT function replaces the TO_CHAR function. The balance of the FORMAT and LPAD functions are displayed in ordinary text after the boldface text on line 13.

Join Results

You can join tables, logical structures that store physical data, or views, logical structures that store directions on how to find data stored in other views or tables, into larger result sets. Views can be subsets of tables, filtered to show only some columns and rows, as covered in the "Query Results" section at the beginning of the chapter. Views can also be the result sets combined from two or more tables or views. You combine tables through join operations.

Although joins in procedural programming languages would involve an outer and inner loop to read two sets into memory, SQL doesn't have loops. As a set-based declarative language, SQL operates like an automatic transmission that hides the clutch and gear-changing process. Imperative languages, on the other hand, are like standard transmissions. Imperative languages require the developer to master the nature of manually switching gears. Managing the clutch and gears would be equivalent to writing outer and inner loops and conditional logic to join the results from two sets into one. SQL joins hide the complexity by letting the developer state what he or she wants without specifying the implementation details.

You can perform several types of joins in SQL. They can be organized into an abstract Unified Modeling Language (UML) inheritance diagram, which is a hierarchy, as shown in Figure 11-6.

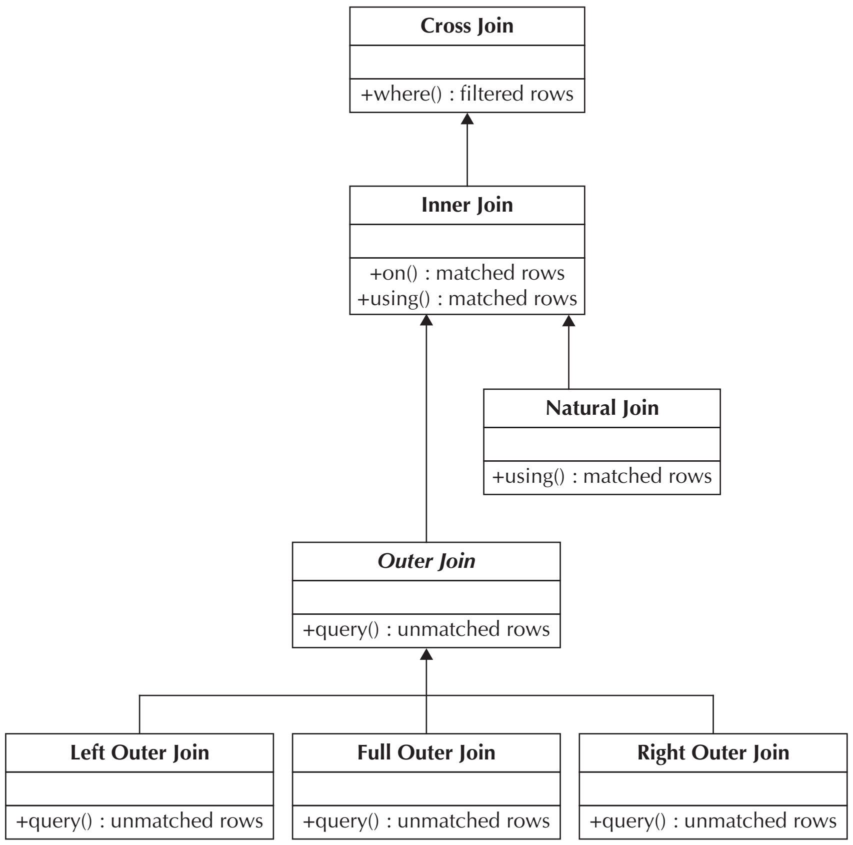


FIGURE 11-6. *Join inheritance tree*

This type of hierarchy is also known as an inverted tree. The top of the hierarchy is the root node and the bottom nodes are the leaf nodes. The root node is the parent node and is not derivative of any other node. Nodes below the root (or parent) node are its child nodes. Leaf nodes aren't parents yet but may become so later; they are child nodes. Nodes between the root and leaf nodes are both parent and child nodes.

Inheritance trees indicate that the most generalized behaviors are in the root node, and most specialized behaviors are in leaf nodes. The inheritance tree tells us the following:

- A *cross join* is the most generalized behavior in joins, and it inherits nothing but implements the base behaviors for join operations in a set-based declarative language.
- An *inner join* is the *only* child of a cross join, and it inherits the behaviors of a cross join and provides some additional features (specialized behaviors) by extending the parent class's behaviors.
- A *natural join* is a child of an inner join, and it inherits and extends the parent class's behaviors. The natural join is also a leaf node, which means no other class extends its behaviors.
- An *outer join* is an abstract class, which means its implementations are available only to subclasses, but it does extend the behavior of the parent class. Through the parent class, it also extends the behavior of the root node or grandparent class.
- A *left join* is a concrete class that extends its abstract parent class and all other classes preceding its parent in the tree. It implements a left outer join behavior beyond the inherited inner join operations.
- A *full join* is a concrete class that extends its abstract parent class and all other classes preceding its parent in the tree. It implements a full outer join behavior beyond the inherited inner join operations.
- A *right join* is a concrete class that extends its abstract parent class and all other classes preceding its parent in the tree. It implements a right outer join behavior beyond the inherited inner join operations.

The subsections qualify what the various join types do. Each section uses Venn diagrams to depict the relationship of sets. Unlike basic sets with a list of elements, the rows in tables are vectors, or record structures. A vector is often visualized as a line, and the line is made up of points. A record structure is like a line when it's labeled as a vector but isn't composed of points. The points in a record data structure are the elements, and each element of a row belongs to a column in a table or view.

There are two types of joins between tables: one splices rows from one table together with rows from another table, and the other splices like collections together. The spliced rows become larger record structures. Splicing together two collections requires that both original collections have the same record structure or table definition. The cross, inner, natural, and outer joins work with splicing rows together. You use set operators to splice together collections of the same record structure.

The next two subsections cover joins that splice rows together and joins that splice collections together. They should be read sequentially because there are some dependencies between the two sections.

Joins that Splice Together Rows

Tables should contain all the information that defines a single subject when the tables are properly normalized. This section assumes you're working with normalized tables. You'll see the differences between ANSI SQL-89 and ANSI SQL-92 joins throughout the subsequent sections.

TIP
You should learn both ANSI SQL-89 and ANSI SQL-92 join semantics.

As a refresher, normalized tables typically contain a set of columns that uniquely defines every row in the table, and collectively these columns are the natural key of the table. Although possible but rare, a single column can define the natural key. The best example of that use case is a `vehicle` table with a `vin` (vehicle identification number) column. Tables also contain surrogate key columns, which are artificial numbers generated through sequences. Surrogate key columns don't define anything about the data, but they uniquely define rows of a table and should have a one-to-one relationship to the natural key of a table. Although the surrogate and natural keys are both candidates to become the primary key of a table, you should always select the surrogate key.

The primary key represents rows of the table externally, and you can copy it to another table as a foreign key. Matches between the primary and foreign key columns let you join tables into multiple-subject record structures. Joins between primary and foreign key values are known as *equijoins* or joins based on an equality match between columns. Joins that don't match based on the equality of columns are *non-equijoins*, and they're filtered searches of a Cartesian product. The "Cross Join" section that follows explains these types of joins.

The natural key differs from a surrogate key because it represents the columns that you use to find a unique row in a table, and it contains descriptive values that define the unique subject of a normalized table. When you write a query against a single table, you use the natural key columns in the `WHERE` clause to find unique rows.

Cross joins are the most generalized form; outer joins are the most specialized.

Cross Join

A cross join in SQL produces a Cartesian product, which is the rows of one table matched against all the rows of another table. It is equivalent to a for loop reading one row from one collection and then a nested for loop appending all rows from another collection to copies of that one row. The operation is repeated for every row in the outer for loop.

A Venn diagram for a cross join is shown in Figure 11-7. The discrete math represents that one set is multiplied by the other. For example, a cross join between a `customer` and `address` table would return 32 rows when the `customer` table holds 4 rows and the `address` table holds 8 rows.

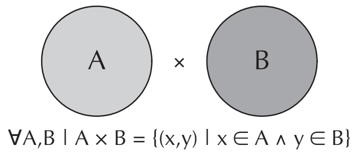


FIGURE 11-7. *Cross join Venn diagram*

Cross joins are useful when you want to perform a non-equijoin match, such as looking up transaction amounts based on their transaction dates within a calendar month. In this case, you'd be filtering a Cartesian product to see when one column in a table holds a value between two columns in the other table. This is also known as a "filtered cross join" statement.

The difference between ANSI SQL-89 and ANSI SQL-92 syntax for a cross join is that the tables are comma-delimited in the `FROM` clause for ANSI SQL-89, whereas, they're bridged by a `CROSS JOIN` operator in ANSI SQL-92.

ANSI SQL-89 Cross Join The following shows a filtered cross join between the `transaction` and `calendar` tables:

```
SQL> SELECT   c.month_short_name
  2  ,        t.transaction_amount
  3  FROM     calendar c, transaction t
  4  WHERE    t.transaction_date BETWEEN c.start_date AND c.end_date
  5  ORDER BY EXTRACT(MONTH FROM t.transaction_date);
```

Notice that the `FROM` clause on line 3 lists comma-delimited tables, and the `WHERE` clause on line 4 doesn't have an equality based join between the primary and foreign key columns.

The query would display the following type of result set:

```
Month
Name  Amount
----- ------
JAN    32.87
JAN    38.99
MAR     9.99
APR    43.19
```

A `GROUP BY` clause on a line 5 combined with a `SUM` aggregation formula on line 2 would return three rows of aggregated data, one row for each distinct month:

```
SQL> SELECT   c.month_short_name
  2  ,        SUM(t.transaction_amount)
  3  FROM     calendar c, transaction t
  4  WHERE    t.transaction_date BETWEEN c.start_date AND c.end_date
  5  GROUP BY c.month_short_name
  6  ORDER BY EXTRACT(MONTH FROM t.transaction_date);
```

This type of cross join logic is extremely useful when you're creating financial statements with SQL queries. It filters results based on whether the value of one table's column is between the values of two columns of another table.

ANSI SQL-92 Cross Join Like the prior syntax example, this example shows a filtered cross join between the `transaction` and `calendar` tables:

```
SQL> SELECT   c.month_short_name
  2  ,        t.transaction_amount
  3  FROM     calendar c CROSS JOIN transaction t
  4  WHERE    t.transaction_date BETWEEN c.start_date AND c.end_date
  5  ORDER BY EXTRACT(MONTH FROM t.transaction_date);
```

Cartesian Product

A Cartesian product is the result of a cross join in SQL. The Cartesian product is named after René Descartes, who is known for penning the phrase, "*Cogito ergo sum*." Translated, it means, "I think therefore I am." He was also a 17th century mathematician who developed the theory of analytical geometry, which lays the foundation for calculus and many aspects of set theory. Set theory is a major foundational element for relational calculus and databases.

This query would return the same result set from the sample data set found in the video store database (see Introduction for details). Notice that in lieu of the comma-delimited tables on line 3, two tables are separated by the `CROSS JOIN` operator.

You also can refactor this query to use the following syntax:

```
SELECT   c.month_short_name
,        t.transaction_amount
FROM     calendar c INNER JOIN transaction t
ON      (t.transaction_date BETWEEN c.start_date AND c.end_date)
ORDER BY EXTRACT(MONTH FROM t.transaction_date);
```

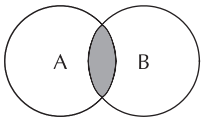
Although this appears to be an inner join operation, it actually runs as a filtered cross join. The `ON` subclause performs a range comparison that determines whether the `transaction_date` column value is found between the dates of two columns from the `calendar` table. Why is the last syntax important if it's misleading and does the same thing? Because Oracle has recently included this on the certification test.

Inner Join

An inner join in SQL produces an intersection between two tables. It lets you splice rows into one large row. It is equivalent to a for loop reading one row from one collection, then a nested for loop reading another row, and finally a conditional if statement checking whether they match. When they match, you have an intersection, and only those rows are returned by an `INNER JOIN` operator.

The Venn diagram for an inner join is shown in Figure 11-8. The discrete math represents that one set intersects the other. For example, when you match a table with a primary and foreign key value, you get only those rows that match both the primary and foreign key values.

There are two key differences between ANSI SQL-89 and ANSI SQL-92 syntax for an inner join. One is that tables are comma-delimited in the `FROM` clause for ANSI SQL-89, and they're



$\forall A,B \mid A \cap B = \{x \in A \cap B \mid x \in A \vee x \in B\}$

FIGURE 11-8. *Inner join Venn diagram*

separated by the `INNER JOIN` operator in ANSI SQL-92. The other is that the join condition is in the `WHERE` clause in ANSI SQL-89 and the `FROM` clause in ANSI SQL-92. You use a `USING` clause when the primary and foreign key column(s) share the same column name and the `ON` clause when they don't. However, you can use the `ON` clause when the column names are the same.

ANSI SQL-89 Inner Join The following shows you how to join the `member` and `contact` tables on their respective primary and foreign key columns. Notice that the tables are comma-delimited and that the join is performed in the `WHERE` clause.

```
SQL> SELECT   COUNT(*)
  2  FROM     member m, contact c
  3  WHERE    m.member_id = c.member_id;
```

The query returns the number of rows where the values in the two *`member_id`* columns match exactly. This is an equijoin (or equality value match) relationship.

For outer join subsections, the parent table will always be on the left side of the operator and the child table will be on the right side. If you swap their locations, the results would likewise invert, because the left join of parent to child is the same as the right join of child to parent.

ANSI SQL-92 Inner Join Like the preceding example, this example shows you how to join the `member` and `contact` tables on their respective primary and foreign key columns. The `INNER JOIN` operator replaces the comma-delimited notation of the older syntax pattern.

Two subclause notations are available for joins: the `USING` subclause and the `ON` subclause. The `USING` subclause works when the column or columns have the same name. These operators are required in inner and outer joins but not with the `NATURAL JOIN` operator.

Here's the `USING` prototype:

```
FROM   table [alias] {LEFT | INNER | RIGHT | FULL} JOIN table [alias]
USING ( column [, column [, … ]])
```

You can provide as many columns as you need in the `USING` subclause. You should enter them as a comma-separated list. They are processed as though they were connected through a series of logical `AND` statements.

The `ON` prototype works when the columns have the same or different names. Here's its prototype:

```
FROM   table [alias] {LEFT | INNER | RIGHT | FULL} JOIN table [alias]
   ON {table | alias}.column = {table | alias}.column
[ AND {table | alias}.column = {table | alias}.column
[ AND ... ]
```

Notice that the following tables aren't comma-delimited. The `INNER JOIN` operator replaces the comma between tables. This join uses the `USING` subclause to match primary and foreign key columns that share the same name.

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m INNER JOIN contact c USING(member_id);
```

You must use the `ON` subclause when the column name or column names aren't the same, but you can also use them when they're the same. Here is an example of the `ON` subclause syntax:

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m INNER JOIN contact c ON m.member_id = c.member_id;
```

Both of these queries return the number of rows where the values in the two `member_id` columns match exactly. Like the older syntax, that means they return the intersection between the two tables and a result set that potentially includes data from both tables.

Natural Join A natural join doesn't exist in the ANSI SQL-89 standard. It exists only in the ANSI SQL-92 standard. The intent of a natural join is to provide the intersection between two sets. It's called a natural join because you don't have to provide the names of the primary or foreign key columns. The natural join checks the data catalog for matching column names with the same data type, and then it creates the join condition implicitly for you.

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m NATURAL JOIN contact c;
```

The only problem with a natural join occurs when columns share the same column name and data type but aren't the primary or foreign keys. A natural join will include them in the join condition. Attempting to match non-key columns that make up the who-audit columns excludes rows that should be returned. The who-audit is composed of the `created_by`, `creation_date`, `last_updated_by`, and `last_update_date` columns (or like column names).

Outer Joins

Outer joins allow you to return result sets that are found both in the intersection and outside the intersection. Applying the paradigm of a parent table that holds a primary key and child table that holds a foreign key, three relationships are possible when foreign key integrity is enforced by the API rather than database constraints:

- **Scenario 1** A row in the parent table matches one or more rows in the child table.
- **Scenario 2** A row in the parent table doesn't match any row in the child table.
- **Scenario 3** A row in the child table doesn't match any row in the parent table, which makes the row in the child table an orphan.

Any or all of the three scenarios can occur. Inner joins help us find the results for scenario 2, but outer joins help us find rows that meet the criteria of scenarios 1 and 3.

NOTE
The ANSI SQL-89 syntax has no provision for outer joins.

Oracle provided outer join syntax before it was defined by the ANSI SQL-92 definition. Oracle's syntax works with joins in the `WHERE` clause. You append a `(+)` on the column of a table in the join, and it indicates that you want the relative complement of that table. A relative complement contains everything not found in the original set.

For example, the following `SELECT` statement uses an Oracle proprietary outer join, and you would get all `account_number` values from the `member` table that had a contact or didn't have a contact. That's because any `member_id` values found in the `member` table that aren't found in the `contact` table are returned with the inner join result set.

```
SQL> SELECT   m.account_number
  2  FROM     member m, contact c
  3  WHERE    m.member_id = c.member_id(+);
```

If you change the `SELECT` list and switch the `(+)` from the contact table's column to the member table's `member_id` column, you would get any orphaned rows from the `contact` table. Here's the syntax to get orphaned contacts' names:

```
SQL> SELECT   c.last_name || ', ' || c.first_name AS customer_name
  2  FROM     member m, contact c
  3  WHERE    m.member_id(+) = c.member_id;
```

Line 3 has the change of the (+) from one side of the join to the other. As mentioned, the `(+)` symbol is included on a column of one table in a join and effectively points to its relative complement in the other table. Having positioned the `member` table on the left and `contact` table on the right in the previous examples, when the `(+)` is pointing from a `contact` table's column, you get the equivalent of a left join. Switch the `(+)` to point from a column in the `member` table and you get the equivalent of a right join.

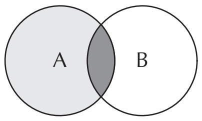
Oracle's proprietary syntax isn't portable to any other platform. It also doesn't support full outer join behavior unless you combine results with a `UNION ALL` set operator, which is covered in the "Union" subsection later in this chapter.

Left Outer Join A left join in SQL extends an inner join because it returns the intersection between two tables plus the right relative complement of the join. The right relative complement is everything in the table to the left of the join operation that's not found in the table on the right. The Venn diagram for a left join is shown in Figure 11-9.

A left join splices several rows into one large row, and it puts null values in the columns from the table on the right when nothing is found in those columns. Left joins use join conditions like those in the inner join section. The ANSI SQL-92 syntax supports a left join operation. That means `JOIN` statements are in the `FROM` clause not the `WHERE` clause, and you use either the `ON` or `USING` subclauses to qualify the joining columns.

The following lets you find any `account_number` values in the `member` table, whether or not they have valid customer information in the contact table:

```
SQL> SELECT   m.account_number
  2  FROM     member m LEFT OUTER JOIN contact c
  3  ON       m.member_id = c.member_id;
```



$$\forall A,B \mid A \cap B + A \setminus B = \{x \in A \wedge x \in B\} + \{y \in A \wedge y \neg\in B\}$$

FIGURE 11-9. *Left join Venn diagram*

The `LEFT OUTER JOIN` phrase is the fully qualified syntax, but the `OUTER` keyword is optional in Oracle and MySQL databases. If you were to reverse the relative positions of the `member` and `contact` tables, only the `account_number` values that meet the conditions of an `INNER JOIN` operator are returned. That's because you would get the right relative complement of the `member` table, not the `contact` table. The right relative complement of the `member` table returns the data rows in the `contact` table that have foreign key values that don't resolve to primary key values in the `member` table. Naturally, this type of data set is possible only when you're not maintaining database-level foreign key constraints.

The more frequent use of a `LEFT JOIN` would be to look for orphaned customers, in order to delete them from the database. In that use case, you'd write the statement like this:

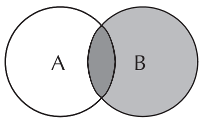
```
SQL> SELECT   c.last_name || ', ' || c.first_name AS customer_name
  2  FROM     contact c LEFT JOIN member m
  3  ON       m.member_id = c.member_id;
```

Notice that the values in the `SELECT` list should come from the table that holds data not from the table that may not contain data. Otherwise, you get a bunch of null values. The preceding query returns rows from the `contact` table that don't point back to a row in the `member` table, and such rows would be orphans. They are referred to as orphans because their foreign key column values aren't found in the list of primary key column values. This can occur when there aren't foreign key constraints in the database.

Right Outer Join Like a left join, a right join in SQL extends an inner join. A right join returns the intersection between two tables, plus the left relative complement of the join. The left relative complement is everything in the table to the right of the join operation that's not found in the table on the left. This makes the right join a mirror image of the left join, as shown in Figure 11-10.

A right join splices rows into one large row like the inner and left join. It puts null values in the columns from the table on the left when they don't match columns that exist in the table on the right. Right joins use join conditions like those in the inner and left join operations. That means they adhere to the ANSI SQL-92 syntax rules, and `JOIN` statements are in the `FROM` clause not the `WHERE` clause. You can choose to use the `ON` or `USING` subclauses to qualify joining columns.

The first example, shown next, lets you find all customer names in the `contact` table—those tables that have a valid foreign key value that matches a valid primary key value, and those that have an invalid foreign key value. Invalid foreign key values can exist only when you opt not to enforce database-level foreign key constraints. Rows holding invalid foreign key values are known as *orphaned* rows because the row with a valid primary key doesn't exist. This follows the paradigm that the parent holds the primary key and the child holds the foreign key (copy of the primary key).



$$\forall A,B \mid A \cap B + B \setminus A = \{x \in A \wedge x \in B\} + \{y \in B \wedge y \neg\in A\}$$

FIGURE 11-10. *Right join Venn diagram*

This example returns all customer names from the `contact` table and really doesn't require a join operation at all:

```
SQL> SELECT   c.last_name || ', ' || c.first_name AS customer_name
  2  FROM     member m RIGHT JOIN contact c
  3  ON       m.member_id = c.member_id;
```

A more meaningful result would exclude all rows where the primary key value is missing. That's easy to do by adding a single filtering `WHERE` clause statement that says "return rows only where there's no valid primary key in the `member` table." Here's the example:

```
SQL> SELECT   c.last_name || ', ' || c.first_name AS customer_name
  2  FROM     member m RIGHT JOIN contact c
  3  ON       m.member_id = c.member_id
  4  WHERE    m.member_id IS NULL;
```

Line 4 filters the return set so that it returns only the orphaned customer names. You would use that type of statement to delete orphan records, typically as a subquery in a `DELETE FROM` statement. Here's an example of such a statement:

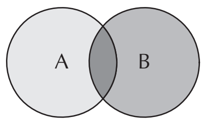
```
SQL> DELETE FROM contact
  2  WHERE  contact_id IN (SELECT   c.contact_id
  3                        FROM     member m RIGHT JOIN contact c
  4                        ON       m.member_id = c.member_id
  5                        WHERE    m.member_id IS NULL);
```

The right and left join semantics are very useful for cleaning up data when foreign key values have lost their matching primary key values. They help you find the relative complements of outer joins when you subtract the inner join rows.

Full Outer Join A full outer join provides the inner join results with the right and left relative complements of left and right joins, respectively. The combination of the two relative complements without the intersection is known as the *symmetric difference*. The Venn diagram for a full outer join is shown in Figure 11-11.

Like the left and right joins, a full outer join splices rows into one large row, like the inner and left join. It puts null values in the columns from the table on the left and right. Assume the following `FULL JOIN` syntax (by the way, `OUTER` is an optional keyword and seldom used):

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m FULL JOIN contact c
  4  ON       m.member_id = c.member_id;
```



$\forall A,B \mid A \cap B + (A \setminus B + B \setminus A)$

FIGURE 11-11. *Full join Venn diagram*

This query's result set would return null values for the `account_number` when there isn't a foreign key using a primary key in the member table, and the `customer_name` when a foreign key value isn't found in the primary key list. This type of analysis is done when database-level foreign key constraints aren't maintained and the API fails to synchronize primary and foreign key values properly. You need to figure out which customers should be associated with a member, because a member row would never be written unless a contact row was also written. This kind of problem occurs more often than you might expect.

You would filter out the inner join by using the following syntax or the `MINUS` set operator (shown later):

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m FULL JOIN contact c
  4  ON       m.member_id = c.member_id
  5  WHERE    m.member_id IS NOT NULL
  6  AND      c.member_id IS NOT NULL;
```

This wraps up joins. Next you'll see how to work with set operators. The next section builds on the examples in this section.

Joins that Splice Collections

Set operations often combine or filter row sets. That means they act as the glue that binds together two queries. The queries must return the same `SELECT` list, which means the column names and data types must match.

The basic prototype glues the top query to the bottom query. Both top and bottom queries can have their own `GROUP BY` or `HAVING` clauses, but only one `ORDER BY` clause can appear at the end. You can splice more than two queries by using other set operators in sequence. The value operations are performed top-down unless you use parentheses to group set operations. The default order of precedence splices the first query result set with the second, and they become a master set that in turn is spliced by another set operator with a subsequent query. Here's a prototype with only a single table in the `FROM` clause:

```
SELECT column_list
FROM some_table
[WHERE some_condition [{AND | OR } some_condition [ ...]]]
[GROUP BY column_list]
[HAVING aggregation_function]
{INTERSECT | UNION | UNION ALL | MINUS}
SELECT column_list
FROM some_table
[WHERE some_condition [{AND | OR } some_condition [ ...]]]
[GROUP BY column_list]
[HAVING aggregation_function]
[ORDER BY column_list];
```

As qualified in the prototype, there are four set operators in SQL: `INTERSECT`, `UNION`, `UNION ALL`, and `MINUS`. The `INTERSECT` operator finds the intersection of two sets and returns a set of unique rows. The `UNION` set operator finds the unique set of rows, and returns them. The `UNION ALL` set operator finds and returns an unsorted merge of all rows from both queries, which results in two copies of any like rows. The MINUS set operator removes the rows in the second query from the rows of the first query where they match.

The following sections discuss the set operators in more depth and provide examples and use cases for them. They're organized in what is the general frequency of use.

Union

The UNION set operator acts like a union in set math and returns the unique things from two sets. This is a two step process: first it gathers the rows into one set and then it sorts them and returns the unique set. Figure 11-12 shows the Venn diagram for a UNION set operation, which looks exactly like a full outer join Venn diagram. The difference between the two is that a full outer join returns all the columns from two tables into one new and larger row, while the UNION merges one set of rows with another (matching set) uniquely.

The UNION set operator lets Oracle achieve a full outer join with its proprietary syntax. One query (A) gets the left join and the other query (B) gets the right join, and the UNION set operator sorts the non-unique row set and returns a unique set.

The code for an Oracle proprietary full outer join would look like this:

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m, contact c
  4  WHERE    m.member_id = c.member_id(+)
  5  UNION
  6  SELECT   m.account_number
  7  ,        c.last_name || ', ' || c.first_name AS customer_name
  8  FROM     member m, contact c
  9  WHERE    m.member_id(+) = c.member_id;
```

The first query returns a left join, which is the inner join between the columns and the right relative complement (those things in the left table not found in the right table). The second query returns the right join. The right join holds a copy of the left relative complement and a second copy of the inner join result set. The UNION set operator sorts the non-unique set and discards the second copy of the inner join.

This is more or less the use case for the UNION set operator. You use it when you can't guarantee that the queries return exclusive sets of rows.

Union All

The UNION ALL set operator differs from the UNION set operator in one key way: it doesn't sort and eliminate duplicate rows. That's a benefit when you can guarantee that two queries return exclusive row sets, because a sorting operation requires more computing resources. The Venn

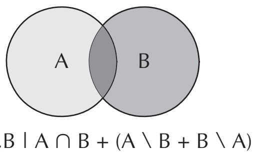


FIGURE 11-12. *Venn diagram for a UNION set operator*

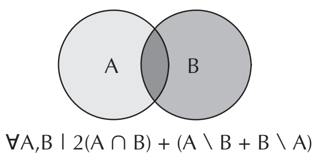


FIGURE 11-13. *Venn diagram for `UNION ALL` operator*

diagram for a `UNION ALL` shown in Figure 11-13 looks remarkably like the one for a `UNION`. The difference is seen in the discrete math below the figure that indicates that it holds two copies of the intersection between the row sets.

The `UNION ALL` set operator is the preferred solution when you can guarantee that queries one and two return unique rows, which means there's no intersection. A symmetrical difference between two sets is the easiest way to demonstrate this set operator, because it is the combined results of the left and right relative complements. This means it doesn't include the intersection between two row sets.

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m LEFT JOIN contact c
  4  ON       m.member_id = c.member_id
  5  WHERE    m.member_id IS NOT NULL
  6  UNION ALL
  7  SELECT   m.account_number
  8  ,        c.last_name || ', ' || c.first_name AS customer_name
  9  FROM     member m RIGHT JOIN contact c
 10  ON       m.member_id = c.member_id
 11  WHERE    c.member_id IS NOT NULL;
```

The first query (A) is a left join that excludes the inner join set, and the second query (B) is a right join that excludes the inner join set. It returns only the symmetric difference between the two row sets.

Intersect

The `INTERSECT` set operator returns only the unique rows found in two queries. It's useful when you want to find rows that meet the criteria of two different queries. The Venn diagram for the `INTERSECT` set operator shown in Figure 11-14 is exactly like that for the inner join. The only difference is one joins row sets and the other joins column sets.

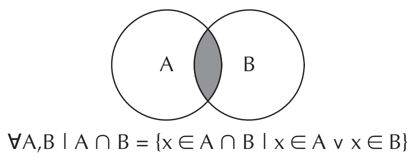


FIGURE 11-14. *Venn diagram for `INTERSECT` operator*

While it might look like a lot of work to get the unique set of rows with an `INTERSECT` set operator, you can verify that the inner join between the `member` and `contact` table is the unique intersection of rows. The `INTERSECT` operator returns the rows that match between query one (A) and query two (B), which is the `INNER JOIN` between the two tables.

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m LEFT JOIN contact c
  4  ON       m.member_id = c.member_id
  5  INTERSECT
  6  SELECT   m.account_number
  7  ,        c.last_name || ', ' || c.first_name AS customer_name
  8  FROM     member m RIGHT JOIN contact c
  9  ON       m.member_id = c.member_id;
```

This query returns the same set of information as an `INNER JOIN` between the *`member`* and *`contact`* table using the `member_id` column. The two relative complements are discarded because the collective values of all columns in the matching rows differ.

Minus

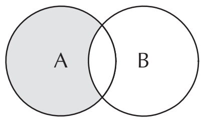
The `MINUS` set operator lets you subtract the matching rows of the second query from the first query. It allows you to find the symmetric difference between two sets, or the relative complement of two sets. Although you can accomplish both of these tasks without set operators by checking whether the joining columns aren't null, sometimes it's a better fit to use set operators to solve this type of problem.

The Venn diagram for the `MINUS` set operator shown in Figure 11-15 is different from those for `JOIN` statements because it excludes the intersection area. I was tempted to provide two examples in this section: one that subtracts the inner join from a full join and another that subtracts the cross join from the same full join. The result would be the same row set, because the only row matches between a full and inner join are the intersection rows, which is the same for a full and cross join. That's because the possible nonjoins in a Cartesian product aren't found in a full join result set.

Here's the full join minus the cross join:

```
SQL> SELECT   m.account_number
  2  ,        c.last_name || ', ' || c.first_name AS customer_name
  3  FROM     member m FULL JOIN contact c
  4  ON       m.member_id = c.member_id
  5  WHERE    m.member_id IS NOT NULL
  6  MINUS
  7  SELECT   m.account_number
  8  ,        c.last_name || ', ' || c.first_name AS customer_name
  9  FROM     member m CROSS JOIN contact c;
```

You always subtract a cross join from a full join when you want the symmetrical difference because it is less expensive than subtracting an inner join.



∀A,B | A \ B = {y ∈ A ∧ y ¬∈ B}

FIGURE 11-15. *Venn diagram for `MINUS` operator*

Views: Stored Queries

Views are stored queries. They can return a subset of one table or a superset of several tables. They can include columns from tables or columns that are the result of expressions, such as string operations, numeric or date math results, or calls to built-in or user-defined functions. Some views are view only (read only) and others are updateable (read-write).

You create views much as you do other things in the database. The next two sections qualify how you create views in Oracle and MySQL. As you'll see, there are some syntax differences.

Creating Oracle Views

Oracle supports ordinary views that are read-only or read-writeable. Here's the basic prototype for creating a view in Oracle:

```
CREATE [OR REPLACE] [[NO] FORCE] VIEW view_name
[( column_name [inline_constraint] [, ...]
[,CONSTRAINT constraint_name
  UNIQUE (column_name) RELY DISABLE NONVALIDATE]
[,CONSTRAINT constraint_name
  PRIMARY KEY (column_list) RELY DISABLE NONVALIDATE]
AS select_statement
[WITH {READ ONLY | CHECK OPTION}]
```

The `OR REPLACE` clause is very helpful because you don't have to drop the previous view before re-creating it with a new definition. Although `NO FORCE` is the default, `FORCE` tells the DBMS to create the view even when the query references things that aren't in the database. This is handy when you're doing a major upgrade, because you can run views concurrently with scripts that change the definition of tables.

You can query the data catalog to find invalid views. This query returns a list of all invalid views in a user's schema:

```
SQL> SELECT   object_name
  2  ,        object_type
  3  ,        status
  4  FROM     user_objects
  5  WHERE    status = 'INVALID';
```

A read-only view occurs when the query (`SELECT` statement) contains a subquery, collection expression, selectivity operator (`DECODE` function or `CASE` operator), or `DISTINCT` operator in the `SELECT` list; or when the query uses a join, set operator, aggregate function, or `GROUP BY`, ORDER BY, MODEL, CONNECT BY, or START WITH clause. The last two clauses are hybrid Oracle clauses that support recursive queries. You can't write to base rows when views perform these types of operations. However, you can deploy INSTEAD OF triggers in Oracle that unwind the logic of the query and let you write to the base rows. Chapter 15 shows a simple example of an INSTEAD OF trigger.

The following is an example of a read-only view:

```
SQL> CREATE OR REPLACE VIEW employee_view
  2  ( employee_id
  3  , employee_name
  4  , employee_status
  5  , CONSTRAINT pk_employee
  6    PRIMARY KEY (employee_id) RELY DISABLE NOVALIDATE)
  7  AS
  8  SELECT   c.contact_id AS employee_id
  9  ,        c.last_name || ', ' || c.first_name ||
 10           CASE
 11             WHEN c.middle_name IS NULL
 12             THEN ' '
 13             ELSE ' '||c.middle_name
 14           END AS employee_name
 15  ,        CASE
 16             WHEN c.contact_type = common_lookup_id
 17             THEN 'Active'
 18             ELSE 'Inactive'
 19           END AS employee_status
 20  FROM     contact c INNER JOIN common_lookup cl
 21  ON       common_lookup_table = 'CONTACT'
 22  AND      common_lookup_column = 'CONTACT_TYPE'
 23  AND      common_lookup_type = 'EMPLOYEE';
```

Line 6 has a primary key constraint on employee_id, which maps inside the subquery to the primary key contact_id column for the contact table. The employee_name column is selectively concatenated from other columns in the contact table, and the employee_status column is fabricated through the combination of a JOIN and CASE operator. Since this is a read-only view, there would be no entry in the user_updatable_columns conceptual view.

A read-write trigger can contain derived columns or expressions, but you cannot write to those specific columns. The following creates a view with two derived columns:

```
SQL> CREATE OR REPLACE VIEW transaction_view AS
  2  SELECT   t.transaction_id AS id
  3  ,        t.transaction_account AS account
  4  ,        t.transaction_type AS purchase_type
  5  ,        t.transaction_date AS purchase_date
  6  ,        t.transaction_amount AS retail_amount
  7  ,        t.transaction_amount * .0875 AS sales_tax
  8  ,        t.transaction_amount * 1.0875 AS total
  9  ,        t.rental_id AS rental_id
 10  ,        t.payment_method_type AS payment_type
 11  ,        t.payment_account_number AS account_number
 12  ,        t.created_by
 13  ,        t.creation_date
 14  ,        t.last_updated_by
 15  ,        t.last_update_date
 16  FROM     transaction t;
```

Lines 7 and 8 contain results from a calculation based on column values multiplied by numeric constants. You can write to the base table through this view by providing an override signature that excludes the two derived columns. Chapter 8 discusses the mechanics of using override signatures in `INSERT` statements.

You can see which columns are updatable with the following query (formatted with SQL*Plus commands):

```
COLUMN column_name FORMAT A20
COLUMN updatable   FORMAT A9
SQL> SELECT   column_name
  2  ,        updatable
  3  FROM     user_updatable_columns
  4  WHERE    table_name = 'TRANSACTION_VIEW'
  5  AND      updatable = 'NO';
```

It would return the two derived columns, where the column names are the aliases assigned in the query:

```
COLUMN_NAME          UPDATABLE
-------------------- ---------
SALES_TAX            NO
TOTAL                NO
```

The `WITH CHECK OPTION` clause limits the rows that you can change with an `UPDATE` statement or `DELETE FROM` the table, by applying a rule that you can touch only those rows that you can see in the view. For example, login protocols often set the `CLIENT_INFO` value when they validate a user's privileges in a web application. Adding a `WHERE` clause on line 17 could limit the visible rows to those that match the value of the session's `CLIENT_INFO` value, like this:

```
17  WHERE TO_NUMBER(NVL(SYS_CONTEXT('userenv','client_info'),-1)) = 0;
```

If a view becomes invalid, you can query the `USER_DEPENDENCIES` concept view. It shows you which dependencies are missing. Replacing the dependencies allows the next DML against the view to work.

The following query finds dependencies for a view:

```
SQL> SELECT   name
  2  ,        referenced_owner||'.'||referenced_name AS reference
  3  ,        referenced_type
  4  FROM     user_dependencies
  5  WHERE    type = 'VIEW'
  6  AND      name = 'view_name';
```

Views are powerful and complex. Although the views you've seen only store queries and rules, Oracle also supports materialized views, which are query results stored in the database.

The results in materialized views are often not current with the moment and can be hours or a day or more old. Materialized views are created when the cost of returning a result set is very high and places an inordinate load on the server during normal operational windows. Materialized views are often used in data marts and warehouses.

Creating MySQL Views

MySQL, like Oracle, supports ordinary views that are read-only or read-writeable. Here's the basic prototype for creating a view:

```
CREATE [OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = { user | CURRENT_USER }]
[SQL SECURITY { DEFINER | INVOKER }]
 VIEW view_name
 [(column_list)]
 AS select_statement
 [WITH [{CASCADED | LOCAL}] CHECK OPTION]
```

Again, the `OR REPLACE` clause is helpful because you don't have to drop the previous view before re-creating it with a new definition. The `OR REPLACE` clause was added in MySQL 5.0.1. Several algorithm and security features are available with views. The `ALGORITHM` clause instructs the MySQL engine how to process the view; it's a SQL extension and `UNDEFINED` is the default option. The `MERGE` algorithm can't be used when a subquery exists in the `SELECT` list, an aggregate function is in the query, or a `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT` clause, or a `UNION` or `UNION ALL` set operator exists.

You can set the definer value of the view as a named user or as the current user, at least if you have super user privileges. `CURRENT_USER` is the default value for the `CREATE VIEW` statement. You can also define a view in a `DEFINER` or `INVOKER` mode of operations. The `DEFINER` mode means that the view runs with the privileges of the user who defined the view, and the `INVOKER` mode means that the view runs with the privileges of the user who calls the view. The `WITH CHECK OPTION` works as it does in Oracle, restricting updatable views only to those rows a user can see.

MySQL imposes several restrictions on views:

- There can be no subqueries in the `SELECT` list.
- The `SELECT` statement cannot refer to system or user variables—session variables.
- The `SELECT` statement cannot refer to prepared statement variables. (Prepared statements are covered in Chapter 14 and they are dynamic SQL statements.)
- The `SELECT` statement cannot refer to temporary tables or tables that haven't been created yet.
- You cannot associate a trigger with a view, which means there is no equivalent to an Oracle `INSTEAD OF` trigger.
- As of MySQL 5.0.52, column aliases have a maximum length of 64, not 256 like tables.

TIP
The `ORDER BY` inside a view is overridden when you put an `ORDER BY` clause in a query that calls an ordered view. As a rule, avoid ordering inside views.

A view can become obsolete when it refers to a table that was removed. The tables referenced in the `FROM` clause of a `SELECT` statement are dependencies for the view. When a DML statement against a view fails, you can use the `CHECK TABLE` statement to find out what dependency is missing. Here's the syntax:

```
CHECK TABLE current_rental;
```

A dependency table was dropped from the database to populate failure messages. Since the screen dump doesn't fit nicely on a page, the following list summarizes the three messages that appear when the `studentdb` database is missing the `rental_item` table:

- 'studentdb.rental_item' doesn't exist
- 'studentdb.current_rental' references invalid table(s) or column(s) or function(s) or definer/invoker of view lack rights to use them
- Corrupt

The first two explain that the table is missing, and because it's missing, the columns cannot be found. The last message says the view is corrupt; Oracle would report it as invalid.

The read-only view requires some modifications to run in MySQL, because it doesn't support constraints in a view and uses piped concatenation. Here's the modified view statement:

```
CREATE OR REPLACE VIEW employee_view
( employee_id
, employee_name
, employee_status )
AS
SELECT   c.contact_id AS employee_id
,        CONCAT(c.last_name,', ',c.first_name,
           CASE
             WHEN c.middle_name IS NULL
               THEN ' '
               ELSE CONCAT(' ',c.middle_name)
           END) AS employee_name
,        CASE
           WHEN c.contact_type = common_lookup_id
           THEN 'Active'
           ELSE 'Inactive'
         END AS employee_status
FROM     contact c INNER JOIN common_lookup cl
ON       common_lookup_table = 'CONTACT'
AND      common_lookup_column = 'CONTACT_TYPE'
AND      common_lookup_type = 'EMPLOYEE';
```

The read-write view works without any changes.

NOTE
Be careful when you incorporate user-defined functions in views, because they can invalidate views when the functions have different privileges than the views.

Summary

This chapter covered how you query results from tables and examined string concatenation, numeric, and date mathematics. It explored the various clauses of `SELECT` statements and the ways to filter result sets in the `WHERE` clause. The chapter also explored built-in Oracle and MySQL date functions; subqueries were also covered.

The chapter also covered how you perform joins between tables by splicing columns together into new record structures, and how you perform set operations between sets of rows. You should have learned the difference between cross, inner, left outer, right outer, and full outer joins, as well as the difference between the `UNION, UNION ALL, INTERSECT`, and `MINUS` set operators.

Mastery Check

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

1. **True** ☐ **False** ☐ A `SELECT` statement can concatenate strings.
2. **True** ☐ **False** ☐ A `SELECT` statement can add, subtract, multiply, and divide numbers to return derived results in the `SELECT` list.
3. **True** ☐ **False** ☐ Date mathematics work the same way in both Oracle and MySQL databases, because you can get yesterday's date by subtracting one from the current date.
4. **True** ☐ **False** ☐ A scalar subquery returns one to many columns and only one row of data.
5. **True** ☐ **False** ☐ An `INNER JOIN` can splice a set of columns from one table with a set of columns from another table based on a column that shares the same value.
6. **True** ☐ **False** ☐ A `LEFT JOIN` returns the `INNER JOIN` results plus everything in the left table of the join not found in the right table.
7. **True** ☐ **False** ☐ In MySQL, you can use the `IF` function to perform conditional operations.
8. **True** ☐ **False** ☐ In Oracle, you can use the `DECODE` function to perform conditional operations.
9. **True** ☐ **False** ☐ You use the `UNION ALL` set operator performs the exact same thing as a `UNION` set operator.
10. **True** ☐ **False** ☐ You can't filter results in the `WHERE` clause on negation of a comparison.
11. Which of the following returns a null and works as a subquery?
    A. A scalar subquery
    B. A single-row subquery
    C. A multiple-row subquery
    D. A correlated subquery
    E. None of the above

12. How many columns does a scalar subquery return?

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

13. What is the maximum number of rows you can update with a correlated subquery?

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

14. The `MINUS` set operator yields the symmetric difference in which of the following examples?

- **A.** The left join minus the right join
- **B.** The left join minus the inner join
- **C.** The right join minus the inner join
- **D.** The full join minus the cross join
- **E.** The full join minus the right join

15. Which clause holds the join between tables in ANSI SQL-92?

- **A.** The `WHERE` clause
- **B.** The `GROUP BY` clause
- **C.** The `FROM` clause
- **D.** The `ORDER BY` clause
- **E.** None, because the natural join principal always manages the process