Elements of the SELECT Statement

The purpose of a *SELECT* statement is to query tables, apply some logical manipulation, and return a result. In this section, I talk about the phases involved in logical query processing. I describe the logical order in which the different query clauses are processed, and what happens in each phase.

Note that by "logical query processing," I'm referring to the conceptual way ANSI SQL defines that a query should be processed and the final result achieved. Don't be alarmed if some logical processing phases that I'll describe here seem inefficient. The Microsoft SQL Server engine doesn't have to follow logical query processing to the letter; rather, it is free to physically process a query differently by rearranging processing phases, as long as the final result would be the same as dictated by logical query processing. SQL Server can—and in fact often does—make many shortcuts in the physical processing of a query.

To describe logical query processing and the various SELECT query clauses, I use the query in Listing 2-1 as an example. For now, don't worry about understanding what this query does; I'll explain the query clauses one at a time, and gradually build this query.

Listing 2-1. Sample Query

```
USE TSQLFundamentals2008;

SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid, orderyear;
```

The code starts with a *USE* statement that sets the database context of your session to the TSQLFundamentals2008 sample database. If your session is already in the context of the database you need to query, the *USE* statement is not required.

Before getting into the details of each phase of the *SELECT* statement, notice the order in which the query clauses are logically processed. In most programming languages the lines of code are processed in the order that they are written. In SQL things are different. Even though the SELECT clause appears first in the query, it is logically processed almost last. The clauses are logically processed in the following order:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

So even though syntactically our sample query in Listing 2-1 starts with a SELECT clause, logically its clauses are processed in the following order:

```
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
ORDER BY empid, orderyear
```

Or to present it in a more readable manner, here's what our statement does:

1. Queries the rows *from* the Sales.Orders table

2. Filters only orders *where* the customer ID is equal to 71

3. *Groups* the orders by employee ID and order year

4. Filters only groups (employee ID and order year) *having* more than one order

5. *Selects* (returns) for each group the employee ID, order year, and number of orders

6. *Orders* (sorts) the rows in the output *by* employee ID and order year

Unfortunately, we cannot write the query in correct logical order. We have to start with the SELECT clause as shown in Listing 2-1.

Now that you understand the order in which the query clauses are logically processed, the next sections explain the details of each phase.

When discussing logical query processing, I refer to query *clauses* and query *phases*, (the WHERE clause and the WHERE phase, for example). A query clause is a syntactical component of a query, so when discussing the syntax of a query element I usually use the term clause (for example, "In the WHERE clause you specify a predicate."). When discussing the logical manipulation taking place as part of logical query processing, I usually use the term phase (for example, "The WHERE phase returns rows for which the predicate evaluates to TRUE").

Recall my recommendation from the previous chapter regarding the use of a semicolon to terminate statements. SQL Server doesn't require you to terminate all statements with a semicolon. This is a requirement only in particular cases where the meaning of the code might otherwise be ambiguous. However, I recommend that you terminate all statements with a semicolon because it is standard, it improves the code readability, and it is likely that SQL Server will require this in more cases in the future. Currently, when a semicolon is not required, adding one doesn't interfere. Therefore I recommend that you make it a practice to terminate all statements with a semicolon.

The FROM Clause

The FROM clause is the very first query clause that is logically processed. In this clause you specify the names of the tables that you want to query and table operators that operate on those tables. This chapter doesn't get into table operators; I describe those in Chapter 5, "Table Expressions". For now, the FROM clause is simply where you specify the name of the table you want to query. The sample query in Listing 2-1 queries the Orders table in the Sales schema, finding 830 rows shown in the output below:

```
FROM Sales.Orders
```

Recall the recommendation I gave in the previous chapter to always schema-qualify object names in your code. When you don't specify the schema name explicitly, SQL Server must resolve it implicitly. This creates some minor cost, and also leaves it to SQL Server to decide which object to use in case of ambiguity. By being explicit, you ensure that you get the object that you intended to get, and that you don't pay any unnecessary penalties.

To return all rows from a table with no special manipulation, all you need is a query with a FROM clause where you specify the table you want to query, and a SELECT clause where you specify the attributes you want to return. For example, the following statement queries all rows from the Orders table in the Sales schema, selecting the attributes orderid, custid, empid, orderdate, and freight:

```
SELECT orderid, custid, empid, orderdate, freight
FROM Sales.Orders;
```

The output of this statement is shown here in abbreviated form:

```
orderid     custid      empid       orderdate                   freight
----------- ----------- ----------- --------------------------- --------------
10248       85          5           2006-07-04 00:00:00.000     32.38
10249       79          6           2006-07-05 00:00:00.000     11.61
10250       34          4           2006-07-08 00:00:00.000     65.83
10251       84          3           2006-07-08 00:00:00.000     41.34
10252       76          4           2006-07-09 00:00:00.000     51.30
```

```
10253       34          3           2006-07-10 00:00:00.000        58.17
10254       14          5           2006-07-11 00:00:00.000        22.98
10255       68          9           2006-07-12 00:00:00.000        148.33
10256       88          3           2006-07-15 00:00:00.000        13.97
10257       35          4           2006-07-16 00:00:00.000        81.91
...

(830 row(s) affected)
```

Although it might seem that the output of the query is returned in a particular order, this is not guaranteed. I'll elaborate on this point later in the chapter in the sections "The SELECT Clause" and "The ORDER BY Clause."

Delimiting Identifier Names

As long as identifiers in your query comply with rules for the format of regular identifiers, you don't need to delimit the identifier names used for schemas, tables, and columns. The rules for the format of regular identifiers can be found in SQL Server Books Online under "Identifiers." If an identifier is irregular—for example, has embedded spaces or special characters, starts with a digit, or is a reserved keyword—you have to delimit it. You can delimit identifiers in SQL Server in a couple of ways. The ANSI SQL form is to use double quotes—for example, "Order Details". The SQL Server specific form is to use square brackets—for example, [Order Details], but it also supports the standard form.

With identifiers that do comply with the rules for the format of regular identifiers, delimiting is optional. For example, a table called Order Details residing in the Sales schema can be referred to as Sales. "Order Details" or "Sales". "Order Details". My personal preference is not to use delimiters when they are not required because they tend to clutter the code. Also, when you're in charge of assigning identifiers, I recommend always using regular ones, for example, OrderDetails instead of Order Details.

The WHERE Clause

In the WHERE clause, you specify a predicate or logical expression to filter the rows returned by the FROM phase. Only rows for which the logical expression evaluates to TRUE are returned by the WHERE phase to the subsequent logical query processing phase. In the sample query in Listing 2-1, the WHERE phase filters only orders placed by customer 71:

```
FROM Sales.Orders
WHERE custid = 71
```

Out of the 830 rows returned by the FROM phase, the WHERE phase filters only the 31 rows where the customer ID is equal to 71. To see which rows you get back after applying the filter custid = 71, run the following query:

```
SELECT orderid, empid, orderdate, freight
FROM Sales.Orders
WHERE custid = 71;
```

This query generates the following output:

```
orderid     empid       orderdate                  freight
----------- ----------- -------------------------- --------------
10324       9           2006-10-08 00:00:00.000    214.27
10393       1           2006-12-25 00:00:00.000    126.56
10398       2           2006-12-30 00:00:00.000    89.16
10440       4           2007-02-10 00:00:00.000    86.53
10452       8           2007-02-20 00:00:00.000    140.26
10510       6           2007-04-18 00:00:00.000    367.63
10555       6           2007-06-02 00:00:00.000    252.49
10603       8           2007-07-18 00:00:00.000    48.77
10607       5           2007-07-22 00:00:00.000    200.24
```

```
10612       1           2007-07-28 00:00:00.000        544.08
10627       8           2007-08-11 00:00:00.000        107.46
10657       2           2007-09-04 00:00:00.000        352.69
10678       7           2007-09-23 00:00:00.000        388.98
10700       3           2007-10-10 00:00:00.000        65.10
10711       5           2007-10-21 00:00:00.000        52.41
10713       1           2007-10-22 00:00:00.000        167.05
10714       5           2007-10-22 00:00:00.000        24.49
10722       8           2007-10-29 00:00:00.000        74.58
10748       3           2007-11-20 00:00:00.000        232.55
10757       6           2007-11-27 00:00:00.000        8.19
10815       2           2008-01-05 00:00:00.000        14.62
10847       4           2008-01-22 00:00:00.000        487.57
10882       4           2008-02-11 00:00:00.000        23.10
10894       1           2008-02-18 00:00:00.000        116.13
10941       7           2008-03-11 00:00:00.000        400.81
10983       2           2008-03-27 00:00:00.000        657.54
10984       1           2008-03-30 00:00:00.000        211.22
11002       4           2008-04-06 00:00:00.000        141.16
11030       7           2008-04-17 00:00:00.000        830.75
11031       6           2008-04-17 00:00:00.000        227.22
11064       1           2008-05-01 00:00:00.000        30.09

(31 row(s) affected)
```

The WHERE clause has significance when it comes to query performance. Based on what you have in the filter expression, SQL Server evaluates the use of indexes to access the required data. By using indexes, SQL Server can sometimes get the required data with much less work compared to applying full table scans. Query filters also reduce the network traffic created by returning all possible rows to the caller and filtering on the client side.

Earlier I mentioned that only rows for which the logical expression evaluates to TRUE are returned by the WHERE phase. Always keep in mind that T-SQL uses three-valued predicate logic, where logical expressions can evaluate to TRUE, FALSE, or UNKNOWN. With three-valued logic, saying "returns TRUE" is not the same as saying "does not return FALSE." The WHERE phase returns rows for which the logical expression evaluates to TRUE, and doesn't return rows for which the logical expression evaluates to FALSE or UNKNOWN. I elaborate on this point later in this chapter in the section "NULLs."

The GROUP BY Clause

The GROUP BY phase allows you to arrange the rows returned by the previous logical query processing phase in groups. The groups are determined by the elements you specify in the GROUP BY clause. For example, the GROUP BY clause in the query in Listing 2-1 has the elements *empid* and *YEAR(orderdate)*:

```
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
```

This means that the GROUP BY phase produces a group for each unique combination of employee ID and order year values that appears in the data returned by the WHERE phase. The expression YEAR(orderdate) invokes the *YEAR* function to return only the year part from the orderdate column.

The WHERE phase returned 31 rows, within which there are 16 unique combinations of employee ID and order year values, as shown here:

```
empid       YEAR(orderdate)
----------- ---------------
1           2006
1           2007
1           2008
2           2006
2           2007
2           2008
3           2007
4           2007
4           2008
5           2007
6           2007
6           2008
7           2007
```

```
7           2008
8           2007
9           2006
```

Thus the GROUP BY phase creates 16 groups, and associates each of the 31 rows returned from the WHERE phase with the relevant group.

If the query involves grouping, all phases subsequent to the GROUP BY phase—including HAVING, SELECT, and ORDER BY—must operate on groups as opposed to operating on individual rows. Each group is ultimately represented by a single row in the final result of the query. This implies that all expressions that you specify in clauses that are processed in subsequent phases to the GROUP BY phase are required to guarantee returning a scalar (single value) per group.

Expressions based on elements that participate in the GROUP BY list meet the requirement because by definition each group has only one unique occurrence of each GROUP BY element. For example, in the group for employee ID 8 and order year 2007, there's only one unique employee ID value and only one unique order year value. Therefore, you're allowed to refer to the expressions empid and YEAR(orderdate) in clauses that are processed in phases subsequent to the GROUP BY phase, such as the SELECT clause. The following query, for example, returns 16 rows for the 16 groups of employee ID and order year values:

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

This query returns the following output:

```
empid       orderyear
----------- -----------
1           2006
1           2007
1           2008
2           2006
2           2007
2           2008
3           2007
4           2007
4           2008
5           2007
6           2007
6           2008
7           2007
7           2008
8           2007
9           2006

(16 row(s) affected)
```

Because an aggregate function returns a single value per group, elements that do not participate in the GROUP BY list are only allowed as inputs to an aggregate function such as *COUNT*, *SUM*, *AVG*, *MIN*, or *MAX*. For example, the following query returns the total freight and number of orders per each employee and order year:

```
SELECT
  empid,
  YEAR(orderdate) AS orderyear,
  SUM(freight) AS totalfreight,
  COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

This query generates the following output:

```
empid       orderyear   totalfreight          numorders
----------- ----------- --------------------- -----------
1           2006        126.56                1
2           2006        89.16                 1
9           2006        214.27                1
```

```
1           2007        711.13              2
2           2007        352.69              1
3           2007        297.65              2
4           2007        86.53               1
5           2007        277.14              3
6           2007        628.31              3
7           2007        388.98              1
8           2007        371.07              4
1           2008        357.44              3
2           2008        672.16              2
4           2008        651.83              3
6           2008        227.22              1
7           2008        1231.56             2

(16 row(s) affected)
```

The expression SUM(freight) returns the sum of all freight values in each group, and the function *COUNT(*)* returns the count of rows in each group—which in our case means number of orders. If you try to refer to an attribute that does not participate in the GROUP BY list (such as freight) and not as an input to an aggregate function in any clause that is processed after the GROUP BY clause, you get an error—in such a case there's no guarantee that the expression will return a single value per group. For example, the following query will fail:

```
SELECT empid, YEAR(orderdate) AS orderyear, freight
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

SQL Server produces the following error:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.Orders.freight' is invalid in the select list because it is not contained in
either an aggregate function or the GROUP BY clause.
```

Note that all aggregate functions ignore NULLs with one exception—*COUNT(*)*. For example, consider a group of five rows with the values 30, 10, NULL, 10, 10 in a column called qty. The expression COUNT(*) would return 5 because there are five rows in the group, while COUNT(qty) would return 4 because there are four known values. If you want to handle only distinct occurrences of known values, specify the *DISTINCT* keyword in the parentheses of the aggregate function. For example, the expression COUNT(DISTINCT qty) would return 2 since there are two distinct known values. The *DISTINCT* keyword can be used with other functions as well. For example, while the expression SUM(qty) would return 60, the expression SUM (DISTINCT qty) would return 40. The expression AVG(qty) would return 15 while the expression AVG (DISTINCT qty) would return 20. As an example of using the DISTINCT option with an aggregate function in a complete query, the following code returns the number of distinct (different) customers handled by each employee in each order year:

```
SELECT
  empid,
  YEAR(orderdate) AS orderyear,
  COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

This query generates the following output:

```
empid       orderyear   numcusts
----------- ----------- -----------
1           2006        22
2           2006        15
3           2006        16
4           2006        26
5           2006        10
6           2006        15
```

```
7           2006        11
8           2006        19
9           2006        5
1           2007        40
2           2007        35
3           2007        46
4           2007        57
5           2007        13
6           2007        24
7           2007        30
8           2007        36
9           2007        16
1           2008        32
2           2008        34
3           2008        30
4           2008        33
5           2008        11
6           2008        17
7           2008        21
8           2008        23
9           2008        16

(27 row(s) affected)
```

The HAVING Clause

With the HAVING clause you can specify a predicate/logical expression to filter groups as opposed to filtering individual rows, which happens in the WHERE phase. Only groups for which the logical expression in the HAVING clause evaluates to TRUE are returned by the HAVING phase to the next logical query processing phase. Groups for which the logical expression evaluates to FALSE or UNKNOWN are filtered out.

Because the HAVING clause is processed after the rows have been grouped, you can refer to aggregate functions in the logical expression. For example, in the query from Listing 2-1 the HAVING clause has the logical expression COUNT(*) > 1, meaning that the HAVING phase filters only groups (employee and order year) with more than one row. The following fragment of the Listing 2-1 query shows what steps have been processed so far:

```
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
```

Recall that the GROUP BY phase created 16 groups of employee ID and order year. Seven of those groups have only one row, so after the HAVING clause is processed, nine groups remain. Run the following query to return those nine groups:

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

This query returns the following output:

```
empid       orderyear
----------- -----------
1           2007
3           2007
5           2007
6           2007
8           2007
1           2008
2           2008
4           2008
7           2008

(9 row(s) affected)
```

The SELECT Clause

The SELECT clause is where you specify the attributes (columns) that you want to return in the result table of the query. You can base the expressions in the SELECT list on attributes from the queried tables, with or without further manipulation. For example, the SELECT list in Listing 2-1 has the following expressions: empid, YEAR(orderdate), and COUNT(*). If an expression refers to an attribute with no manipulation, such as empid, the name of the target attribute is the same as the name of the source attribute. You can optionally assign your own name to the target attribute by using the AS clause—for example, empid AS employee_id. Expressions that do apply manipulation, such as YEAR(orderdate), or are not based on a source attribute, such as a call for the function *CURRENT_TIMESTAMP*, don't have a name in the result of the query if you don't alias them. T-SQL allows a query to return result columns with no names in certain cases, but the relational model doesn't. It's strongly recommended that you alias such expressions as YEAR (orderdate) AS orderyear so that all result attributes have names. In this respect, the result table returned from the query would be considered relational.

T-SQL supports, in addition to the AS clause, a couple of other forms with which you can alias expressions, but to me, the AS clause seems the most readable and intuitive form and therefore I recommend using it. I will cover the other forms for the sake of completeness and also in order to describe an elusive bug related to one of them. Besides the form <expression> AS <alias>, T-SQL also supports the forms <alias> = <expression> (alias equals expression), and <expression> <alias> (expression space alias). An example of the former is orderyear = YEAR(orderdate), and an example of the latter is YEAR(orderdate) orderyear. I find the latter form in which you specify the expression followed by a space and the alias, particularly unclear, and I strongly recommend that you avoid using it.

It is interesting to note that if by mistake you don't specify a comma between two column names in the SELECT list, your code won't fail. Rather SQL Server will assume that the second name is an alias for the first column name. As an example, suppose that you wanted to write a query that selects the orderid and orderdate columns from the Sales.Orders table, and by mistake didn't specify the comma between the column names, as follows:

```
SELECT orderid orderdate
FROM Sales.Orders;
```

This query is considered syntactically valid, as if you intended to alias the orderid column as orderdate. In the output you will get only one column holding the order IDs, with the alias orderdate:

```
orderdate
-----------
10248
10249
10250
10251
10252
...

(830 row(s) affected)
```

It can be hard to detect such a bug, so the best you can do is to be alert when writing code.

With the addition of the SELECT phase, the following query clauses from the query in Listing 2-1 have been processed so far:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
```

The SELECT clause produces the result table of the query. In the case of the query in Listing 2-1, the heading of the result table has the attributes empid, orderyear, and numorders, and the body has nine rows (one for each group). Run the following query to return those nine rows:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

This query generates the following output:

```
empid       orderyear   numorders
----------- ----------- -----------
1           2007        2
3           2007        2
5           2007        3
6           2007        3
8           2007        4
1           2008        3
2           2008        2
4           2008        3
7           2008        2

(9 row(s) affected)
```

Remember that the SELECT clause is processed after the FROM, WHERE, GROUP BY, and HAVING clauses. This means that aliases assigned to expressions in the SELECT clause do not exist as far as clauses that are processed before the SELECT clause are concerned. A very typical mistake made by programmers who are not familiar with the correct logical processing order of query clauses is to refer to expression aliases in clauses that are processed prior to the SELECT clause. Here's an example of such an invalid attempt in the WHERE clause:

```
SELECT orderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE orderyear > 2006;
```

On the surface this query might seem valid, but if you consider the fact that the column aliases are created in the SELECT phase—which is processed after the WHERE phase—you can see that the reference to the orderyear alias in the WHERE clause is invalid. And in fact, SQL Server produces the following error:

```
Msg 207, Level 16, State 1, Line 3
Invalid column name 'orderyear'.
```

One way around this problem is to repeat the expression YEAR(orderdate) in both the WHERE and the SELECT clauses:

```
SELECT orderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE YEAR(orderdate) > 2006;
```

It's interesting to note that SQL Server is capable of identifying the repeated use of the same expression—YEAR(orderdate)—in the query. It only needs to be evaluated or calculated once.

The following query is another example of an invalid reference to a column alias. The query attempts to refer to a column alias in the HAVING clause, which is also processed before the SELECT clause:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING numorders > 1;
```

This query fails with an error saying that the column name numorders is invalid. You would also need to repeat the expression COUNT(*) in both clauses:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

In the relational model, operations on relations are based on relational algebra and result in a relation (a set). In SQL, things are a bit different in the sense that a SELECT query is not guaranteed to return a true

set—namely, unique rows with no guaranteed order. To begin with, SQL doesn't require a table to qualify as a set. Without a key, uniqueness of rows is not guaranteed, in which case the table isn't a set; it's a multiset or a bag. But even if the tables you query have keys and qualify as sets, a SELECT query against the tables can still return a result with duplicate rows. The term result set is often used to describe the output of a SELECT query, but a result set doesn't necessarily qualify as a set in the mathematical sense. For example, even though the Orders table is a set because uniqueness is enforced with a key, a query against the Orders table returns duplicate rows, as shown in Listing 2-2:

Listing 2-2. Query Returning Duplicate Rows

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;
```

This query generates the following output:

```
empid       orderyear
----------- -----------
9           2006
1           2006
2           2006
4           2007
8           2007
6           2007
6           2007
8           2007
5           2007
1           2007
8           2007
2           2007
7           2007
3           2007
5           2007
1           2007
5           2007
8           2007
3           2007
6           2007
2           2008
4           2008
4           2008
1           2008
7           2008
2           2008
1           2008
4           2008
7           2008
6           2008
1           2008

(31 row(s) affected)
```

SQL provides the means to guarantee uniqueness in the result of a *SELECT* statement in the form of a DISTINCT clause that removes duplicate rows, as shown in Listing 2-3:

Listing 2-3. Query with a DISTINCT Clause

```
SELECT DISTINCT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;
```

This query generates the following output:

```
empid       orderyear
----------- -----------
1           2006
1           2007
1           2008
2           2006
2           2007
2           2008
3           2007
4           2007
4           2008
5           2007
6           2007
6           2008
7           2007
7           2008
8           2007
9           2006

(16 row(s) affected)
```

Of the 31 rows in the multiset returned by the query in Listing 2-2, 16 rows are in the set returned by the query in Listing 2-3 after removal of duplicates.

SQL supports the use of an asterisk (*) in the SELECT list to request all attributes from the queried tables instead of listing them explicitly, as in the following example:

```
SELECT *
FROM Sales.Shippers;
```

Such use of an asterisk is a bad programming practice in most cases, with very few exceptions. It is recommended that you explicitly specify the list of attributes that you need even if you need all of the attributes from the queried table. There are many reasons for this recommendation. Unlike the relational model, SQL keeps ordinal positions for columns based on the order in which the columns were specified in the *CREATE TABLE* statement. By specifying SELECT *, you're guaranteed to get the columns back in order based on their ordinal positions. Client applications can refer to columns in the result by their ordinal positions (a bad practice in its own right) instead of by name. Any schema changes applied to the table—such as adding or removing columns, rearranging their order, and so on—might result in failures in the client application, or even worse, logical bugs that will go unnoticed. By specifying the attributes that you need explicitly, you always get the right ones, as long as the columns exist in the table. If a column referenced by the query was dropped from the table, you get an error and can fix your code accordingly.

Some people wonder whether there's any performance difference between specifying an asterisk and explicitly listing column names. Some extra work may be required in resolving column names when using the asterisk, but it is usually so negligible compared to other costs involved in the query that it is unlikely to be noticed. If there is any performance difference, as minor as it may be, it is most probably in the favor of explicitly listing column names. Because that's the recommended practice anyway, it's a win-win situation.

Within the SELECT clause you are still not allowed to refer to a column alias that was created in the same SELECT clause, regardless of whether the expression that assigns the alias appears to the left or right of the expression that attempts to refer to it. For example, the following attempt is invalid:

```
SELECT orderid,
  YEAR(orderdate) AS orderyear,
  orderyear + 1 AS nextyear
FROM Sales.Orders;
```

As explained earlier in this section, one of the ways around this problem is to repeat the expression:

```
SELECT orderid,
  YEAR(orderdate) AS orderyear,
  YEAR(orderdate) + 1 AS nextyear
FROM Sales.Orders;
```

The ORDER BY Clause

The ORDER BY clause allows you to sort the rows in the output for presentation purposes. In terms of logical query processing, ORDER BY is the very last clause to be processed. The sample query shown in Listing 2-4

sorts the rows in the output by employee ID and order year:

Listing 2-4. Query Demonstrating the ORDER BY Clause

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid, orderyear;
```

This query generates the following output:

```
empid       orderyear   numorders
----------- ----------- -----------
1           2007        2
1           2008        3
2           2008        2
3           2007        2
4           2008        3
5           2007        3
6           2007        3
7           2008        2
8           2007        4

(9 row(s) affected)
```

One of the most important points to understand about SQL is that a table has no guaranteed order, because a table is supposed to represent a set (or multiset if it has duplicates), and a set has no order. This means that when you query a table without specifying an ORDER BY clause, the query returns a table result, and SQL Server is free to return the rows in the output in any order. The only way for you to guarantee that the rows in the result are sorted is to explicitly specify an ORDER BY clause. However, if you do specify an ORDER BY clause, the result cannot qualify as a table because the order of the rows in the result is guaranteed. A query with an ORDER BY clause results in what ANSI calls a cursor—a nonrelational result with order guaranteed among rows. You're probably wondering why it matters whether a query returns a table result or a cursor. Some language elements and operations in SQL expect to work with table results of queries and not with cursors; examples include table expressions and set operations, which I cover in detail later in the book.

Notice that the ORDER BY clause refers to the column alias orderyear, which was created in the SELECT phase. The ORDER BY phase is in fact the only phase in which you can refer to column aliases created in the SELECT phase, because it is the only phase that is processed after the SELECT phase.

When you want to sort by an expression in ascending order, you either specify ASC right after the expression, such as orderyear ASC, or don't specify anything after the expression because ASC is the default. If you want to sort in descending order, you need to specify DESC after the expression, such as orderyear DESC.

SQL and T-SQL both allow you to specify in the ORDER BY clause ordinal positions of columns based on the order in which the columns appear in the SELECT list. For example, in the query in Listing 2-4, instead of using:

```
ORDER BY empid, orderyear
```

You could use:

```
ORDER BY 1, 2
```

However, this is considered bad programming practice for a couple of reasons. First, in the relational model attributes don't have ordinal positions and need to be referred to by name. Second, when you make revisions to the SELECT clause, you might forget to make the corresponding revisions in the ORDER BY clause. When you use column names, your code is safe from this type of mistake.

T-SQL allows you to specify elements in the ORDER BY clause that do not appear in the SELECT clause,

meaning that you can sort by something that you don't necessarily want to return in the output. For example, the following query sorts the employee rows by hire date without returning the hiredate attribute:

```
SELECT empid, firstname, lastname, country
FROM HR.Employees
ORDER BY hiredate;
```

However, when DISTINCT is specified, you are restricted in the ORDER BY list only to elements that appear in the SELECT list. The reasoning behind this restriction is that when DISTINCT is specified, a single result row might represent multiple source rows; therefore, it might not be clear which of the multiple possible values in the ORDER BY expression should be used. Consider the following invalid query:

```
SELECT DISTINCT country
FROM HR.Employees
ORDER BY empid;
```

There are nine employees in the Employees table—five from the USA and four from the UK. If you omit the invalid ORDER BY clause from this query, you get two rows back—one for each distinct country. Because each country appears in multiple rows in the source table, and each such row has a different employee ID, the meaning of ORDER BY empid is not really defined.

The TOP Option

The TOP option is a proprietary T-SQL feature that allows you to limit the number or percentage of rows that your query returns. When an ORDER BY clause is specified in the query, the TOP option relies on it to define the logical precedence among rows. For example, to return from the Orders table the five most recent orders, you would specify TOP (5) in the SELECT clause and orderdate DESC in the ORDER BY clause, as shown in Listing 2-5:

Listing 2-5. Query Demonstrating the TOP Option

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

This query returns the following output:

```
orderid     orderdate                   custid      empid
----------- --------------------------- ----------- -----------
11077       2008-05-06 00:00:00.000     65          1
11076       2008-05-06 00:00:00.000     9           4
11075       2008-05-06 00:00:00.000     68          8
11074       2008-05-06 00:00:00.000     73          7
11073       2008-05-05 00:00:00.000     58          2

(5 row(s) affected)
```

In terms of logical query processing, the TOP option is processed as part of the SELECT phase, right after the DISTINCT clause is processed (if one exists). Note that when TOP is specified in a query the ORDER BY clause serves a dual purpose. That is, as part of the SELECT phase the TOP option relies on the ORDER BY clause to determine logical precedence among rows, and based on this precedence filters as many as were requested. Later, as part of the ORDER BY phase that follows the SELECT phase, the very same ORDER BY clause is used to sort the rows in the output for presentation purposes. For example, the query in Listing 2-5 returns the five rows with the highest orderdate values, and sorts the rows in the output by orderdate DESC for presentation purposes.

If you're confused about whether a TOP query returns a table result or a cursor, you have every reason to be. When TOP is used, the same ORDER BY clause serves both the purpose of determining logical precedence for TOP, and the normal meaning—presentation—that changes the nature of the result from a table to a cursor with guaranteed order. For example, you can't specify in the same query that you want the logical precedence among rows to be determined by one ORDER BY list for the TOP option, while you want to sort the rows in the output for presentation purposes by another, or not at all. To achieve this, you have to use a table expression, but I'll save the discussion of table expressions for Chapter 5. All I want to say for now is that if the design of the TOP option seems confusing, there's a good reason. In other words, it's not you—it's

the feature's design.

You can use the TOP option with the *PERCENT* keyword, in which case SQL Server calculates the number of rows to return based on a percentage of the number of qualifying rows, rounded up. For example, the following query requests the top one percent of the most recent orders:

```
SELECT TOP (1) PERCENT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

This query generates the following output:

```
orderid     orderdate                   custid      empid
----------- --------------------------- ----------- -----------
11074       2008-05-06 00:00:00.000     73          7
11075       2008-05-06 00:00:00.000     68          8
11076       2008-05-06 00:00:00.000     9           4
11077       2008-05-06 00:00:00.000     65          1
11070       2008-05-05 00:00:00.000     44          2
11071       2008-05-05 00:00:00.000     46          1
11072       2008-05-05 00:00:00.000     20          4
11073       2008-05-05 00:00:00.000     58          2
11067       2008-05-04 00:00:00.000     17          1

(9 row(s) affected)
```

The query returns 9 rows because the Orders table has 830 rows, and 1 percent of 830, rounded up, is 9.

In the query in Listing 2-5, you might have noticed that the ORDER BY list is not unique because no primary key or unique constraint is defined on the orderdate column. Multiple rows can have the same order date. In a case where no tiebreaker is specified, precedence among rows in case of ties (rows with the same order date) is undefined. This fact makes the query nondeterministic—more than one result can be considered correct. In case of ties, SQL Server chooses rows based on whichever row it physically happens to access first.

Notice in the output for the query in Listing 2-5 that the minimum order date out of the rows returned is May 5, 2008, and one row in the output has that date. Other rows in the table may have the same order date, and with the existing non-unique ORDER BY list, there is no guarantee which of those will be returned.

If you want the query to be deterministic, you need to make the ORDER BY list unique; in other words, add a tiebreaker. For example, you can add orderid DESC to the ORDER BY list as shown in Listing 2-6 so that in case of ties, precedence is determined by order ID descending:

Listing 2-6. Query Demonstrating TOP with Unique ORDER BY List

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
```

This query returns the following output:

```
orderid     orderdate                   custid      empid
----------- --------------------------- ----------- -----------
11077       2008-05-06 00:00:00.000     65          1
11076       2008-05-06 00:00:00.000     9           4
11075       2008-05-06 00:00:00.000     68          8
11074       2008-05-06 00:00:00.000     73          7
11073       2008-05-05 00:00:00.000     58          2

(5 row(s) affected)
```

If you examine the results of the queries from Listing 2-5 and Listing 2-6, you'll notice that they seem to be the same. The important difference is that the result shown in the query output for Listing 2-5 is one of several possible valid results for this query, while the result shown in the query output for Listing 2-6 is the only possible valid result.

Instead of adding a tiebreaker to the ORDER BY list, you can request to return all ties. For example, besides the five rows that you get back from the query in Listing 2-5, you can ask to return all other rows from the table that have the same sort value (order date in our case) as the last one found (May 5, 2008 in our case). You achieve this by adding the *WITH TIES* option as shown in the following query:

```
SELECT TOP (5) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

This query returns the following output:

```
orderid     orderdate                   custid      empid
----------- --------------------------- ----------- -----------
11077       2008-05-06 00:00:00.000     65          1
11076       2008-05-06 00:00:00.000     9           4
11075       2008-05-06 00:00:00.000     68          8
11074       2008-05-06 00:00:00.000     73          7
11073       2008-05-05 00:00:00.000     58          2
11072       2008-05-05 00:00:00.000     20          4
11071       2008-05-05 00:00:00.000     46          1
11070       2008-05-05 00:00:00.000     44          2

(8 row(s) affected)
```

Notice that the output has eight rows even though you specified TOP (5). SQL Server first returned the TOP (5) rows based on orderdate DESC precedence, and also all other rows from the table that had the same *orderdate* value as in the last of the five rows that was accessed.

The OVER Clause

The OVER clause exposes a window of rows to certain kinds of calculations. Think of a window of rows simply as a certain set of rows that the calculation operates on. Aggregate and ranking functions, for example, are the types of calculations that support the OVER clause. Because the OVER clause exposes a window of rows to those functions, they are known as window functions.

Because the whole point of an aggregate function is to aggregate a set of values, aggregate functions traditionally operate in the context of GROUP BY queries. Recall from earlier discussions in the section "The GROUP BY Clause" that once you group data, the query returns only one row for each group; therefore, all your expressions are restricted to returning a single value per group.

An aggregate window function operates against a set of values in a window of rows that you expose to it using the OVER clause, and not in the context of a GROUP BY query. Therefore, you don't have to group the data, and you can return base row attributes and aggregates in the same row.

To understand the OVER clause, think of the Sales.OrderValues view. I will discuss views in Chapter 10, "Programmable Objects," but for now, simply think of a view as if it were a table. The Sales.OrderValues view has a row for each order, with the order ID (orderid), customer ID (custid), employee ID (empid), shipper ID (shipperid), order date (orderdate) and order value (val).

An OVER clause with empty parentheses exposes all rows to the calculation. The phrase "all rows" doesn't necessarily mean all rows from the table that appears in the FROM clause; rather the rows exposed are those available after the FROM, WHERE, GROUP BY and HAVING phases are completed. Note that the OVER clause is allowed only in the SELECT and ORDER BY phases. In order not to overwhelm you with too much information at this early stage, I'll focus on using the OVER clause in the SELECT phase. So, for example, if you specify the expression SUM(val) OVER() in the SELECT clause of a query against the OrderValues view, the function calculates the total value out of all rows that the SELECT phase operates on. If the query doesn't filter data or apply any other logical phases before the SELECT phase, the expression returns the total value out of all OrderValues rows.

If you want to restrict or partition the rows, you can use the PARTITION BY clause. For example, if instead of returning the total value of all OrderValues rows, you want to return the total value of the current customer (out of all rows with the same custid as in the current row), specify SUM(val) OVER(PARTITION BY custid).

To demonstrate both nonpartitioned and partitioned expressions, the following query returns all OrderValues rows. Additionally, in each row except for the base attributes, the query returns the grand total value and the customer total value:

```
SELECT orderid, custid, val,
```

```
  SUM(val) OVER() AS totalvalue,
  SUM(val) OVER(PARTITION BY custid) AS custtotalvalue
FROM Sales.OrderValues;
```

This query returns the following output:

```
orderid     custid      val          totalvalue       custtotalvalue
----------- ----------- ------------ ---------------- ---------------
10643       1           814.50       1265793.22       4273.00
10692       1           878.00       1265793.22       4273.00
10702       1           330.00       1265793.22       4273.00
10835       1           845.80       1265793.22       4273.00
10952       1           471.20       1265793.22       4273.00
11011       1           933.50       1265793.22       4273.00
10926       2           514.40       1265793.22       1402.95
10759       2           320.00       1265793.22       1402.95
10625       2           479.75       1265793.22       1402.95
10308       2           88.80        1265793.22       1402.95
10365       3           403.20       1265793.22       7023.98
...

(830 row(s) affected)
```

The totalvalue column has, in all result rows, the total value out of all rows. The custtotalvalue column has the total value out of all rows that have the same custid value as in the current row.

One benefit of the OVER clause is that by enabling you to return base row attributes and aggregate them in the same row, it also enables you to write expressions that mix the two. For example, the following query calculates for each OrderValues row the percentage of the current value out of the grand total, and also the percentage of the current value out of the customer total:

```
SELECT orderid, custid, val,
  100. * val / SUM(val) OVER() AS pctall,
  100. * val / SUM(val) OVER(PARTITION BY custid) AS pctcust
FROM Sales.OrderValues;
```

Note that the reason that I specified the decimal value 100. (one hundred dot) in the expressions instead of the integer 100 is in order to cause implicit conversion of the integer values val and SUM(val) to decimal values. Otherwise, the division would have been an integer division and the fractional part would have been truncated.

This query returns the following output:

```
orderid      custid val         pctall                          pctcust
------------ ------ ----------- ------------------------------- ------------------------------
10643        1      814.50      0.0643470029014691672941        19.0615492628130119354083
10692        1      878.00      0.0693636200705830925528        20.5476246197051252047741
10702        1      330.00      0.0260706089103558320528        7.7229113035338169904048
10835        1      845.80      0.0668197606556938265161        19.7940556985724315469225
10952        1      471.20      0.0372256694501808123130        11.0273812309852562602387
11011        1      933.50      0.0737482224782338461253        21.8464778843903580622513
10926        2      514.40      0.0406385491620819394181        36.6655974910011048148544
10759        2      320.00      0.0252805904585268674452        22.8090808653195053280587
10625        2      479.75      0.0379011352264945770526        34.1958017035532271285505
10308        2      88.80       0.0070153638522412057160        6.3295199401261627285362
10365        3      403.20      0.0318535439777438529809        5.7403352515240647040566
...

(830 row(s) affected)
```

The OVER clause is also supported with four ranking functions: *ROW_NUMBER*, *RANK*, *DENSE_RANK*, and *NTILE*. The following query demonstrates the use of these functions:

```
SELECT orderid, custid, val,
  ROW_NUMBER() OVER(ORDER BY val) AS rownum,
```

```
  RANK()       OVER(ORDER BY val) AS rank,
  DENSE_RANK() OVER(ORDER BY val) AS dense_rank,
  NTILE(100)   OVER(ORDER BY val) AS ntile
FROM Sales.OrderValues
ORDER BY val;
```

This query generates the following output:

```
orderid     custid      val       rownum  rank    dense_rank ntile
----------- ----------- --------- ------- ------- ---------- -----
10782       12          12.50     1       1       1          1
10807       27          18.40     2       2       2          1
10586       66          23.80     3       3       3          1
10767       76          28.00     4       4       4          1
10898       54          30.00     5       5       5          1
10900       88          33.75     6       6       6          1
10883       48          36.00     7       7       7          1
11051       41          36.00     8       7       7          1
10815       71          40.00     9       9       8          1
10674       38          45.00     10      10      9          1
...
10691       63          10164.80  821     821     786        10
10540       63          10191.70  822     822     787        10
10479       65          10495.60  823     823     788        10
10897       37          10835.24  824     824     789        10
10817       39          10952.85  825     825     790        10
10417       73          11188.40  826     826     791        10
10889       65          11380.00  827     827     792        10
11030       71          12615.05  828     828     793        10
10981       34          15810.00  829     829     794        10
10865       63          16387.50  830     830     795        10

(830 row(s) affected)
```

The *ROW_NUMBER* function assigns incrementing sequential integers to the rows in the result set of a query, based on logical order that is specified in the ORDER BY subclause of the OVER clause. In our sample query, the logical order is based on the val column; therefore, you can see in the output that when the value increases the row number increases as well. However, even when the ordering value doesn't increase, the row number still must increase. Therefore, if the *ROW_NUMBER* function's ORDER BY list is non-unique, as in the preceding example, the query is nondeterministic. That is, more than one correct result is possible. For example, observe that two rows with the value 36.00 got the row numbers 7 and 8. Any arrangement of these row numbers would have been considered correct. If you want to make a row number calculation deterministic, you need to add elements to the ORDER BY list to make it unique; meaning that the list of elements in the ORDER BY clause would uniquely identify rows. For example, you can add the orderid column as a tiebreaker to the ORDER BY list to make the row number calculation deterministic.

As I mentioned earlier, the *ROW_NUMBER* function must produce unique values even when there are ties in the ordering values. If you want to treat ties in the ordering values the same way, you will probably want to use the *RANK* or *DENSE_RANK* function instead. Both are similar to the *ROW_NUMBER* function, but they produce the same ranking value in all rows that have the same logical ordering value. The difference between *RANK* and *DENSE_RANK* is that *RANK* indicates how many rows have a lower ordering value, while *DENSE_RANK* indicates how many distinct ordering values are lower. For example, in our sample query, a rank 9 indicates 8 rows with lower values. A dense rank 9 indicates 8 distinct lower values.

The *NTILE* function allows you to associate the rows in the result with tiles (equally sized groups of rows) by assigning a tile number to each row. You specify as input to the function how many tiles you are after, and in the OVER clause you specify the logical ordering. Our sample query has 830 rows and the request was for 10 tiles; therefore, the tile size is 83 (830 divided by 10). Logical ordering is based on the val column. This means that the 83 rows with the lowest values are assigned with tile number 1, the next 83 with tile number 2, the next 83 with tile number 3, and so on. The *NTILE* function is logically related to the *ROW_NUMBER* function. It's as if you assigned row numbers to the rows based on val ordering, and based on the calculated tile size 83 assigned tile number 1 to rows 1 through 83, tile number 2 to rows 84 through 166, and so on. If the number of rows doesn't divide evenly by the number of tiles, from the remainder an extra row is added to each of the first tiles. For example, if there had been 102 rows and 5 tiles were requested, the first two tiles would have had 21 rows instead of 20.

Like aggregate window functions, ranking functions also support a PARTITION BY clause in the OVER clause. It's probably easy to understand the meaning of the PARTITION BY clause in the context of ranking

calculations; think of it as making the calculation independent for each partition, or window. For example, the expression ROW_NUMBER() OVER(PARTITION BY custid ORDER BY val) assigns row numbers for each subset of rows with the same custid independently, as opposed to assigning those across the whole set. Here's the expression in a query:

```
SELECT orderid, custid, val,
  ROW_NUMBER() OVER(PARTITION BY custid
                    ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

This query generates the following output:

```
orderid     custid      val          rownum
----------- ----------- ------------ -------
10702       1           330.00       1
10952       1           471.20       2
10643       1           814.50       3
10835       1           845.80       4
10692       1           878.00       5
11011       1           933.50       6
10308       2           88.80        1
10759       2           320.00       2
10625       2           479.75       3
10926       2           514.40       4
10682       3           375.50       1
...

(830 row(s) affected)
```

As you can see in the output, the row numbers are calculated independently for each customer, as though the calculation were reset for each customer.

Note that the logical ORDER BY specified in the OVER clause has nothing to do with presentation, and does not change the nature of the result from being a table. If you do not specify a presentation ORDER BY in the query, as explained earlier, you don't have any guarantees in terms of the order of the rows in the output. If you need to guarantee presentation order, you have to add a presentation ORDER BY clause, as I did in the last two queries demonstrating the use of ranking functions.

If specified in the SELECT phase, window calculations are processed before the DISTINCT clause (if one exists).

To put it all together, the following list presents the logical order in which all clauses discussed so far are processed:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
    - OVER
    - DISTINCT
    - TOP
- ORDER BY

Are you wondering why it matters that the DISTINCT clause is processed after window calculations that appear in the SELECT clause are processed, and not before? I'll explain with an example. Currently the OrderValues view has 830 rows with 795 distinct values. Consider the following query and its output:

```
SELECT DISTINCT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues;
```

```
val        rownum
---------- -------
12.50      1
18.40      2
23.80      3
28.00      4
30.00      5
33.75      6
36.00      7
36.00      8
40.00      9
45.00      10
...
12615.05   828
15810.00   829
16387.50   830

(830 row(s) affected)
```

The *ROW_NUMBER* function is processed before the DISTINCT clause. First, unique row numbers are assigned to the 830 rows from the OrderValues view. Then the DISTINCT clause is processed—therefore, no duplicate rows to remove. You can consider it a best practice not to specify both DISTINCT and ROW_NUMBER in the same SELECT clause as the DISTINCT clause has no effect in such a case. If you want to assign row numbers to the 795 unique values, you need to come up with a different solution. For example, because the GROUP BY phase is processed before the SELECT phase, you could use the following query:

```
SELECT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues
GROUP BY val;
```

This query generates the following output:

```
val       rownum
--------- -------
12.50     1
18.40     2
23.80     3
28.00     4
30.00     5
33.75     6
36.00     7
40.00     8
45.00     9
48.00     10
...
12615.05  793
15810.00  794
16387.50  795

(795 row(s) affected)
```

Here, the GROUP BY phase produces 795 groups for the 795 distinct values, and then the SELECT phase produces a row for each group with the value and a row number based on val order.