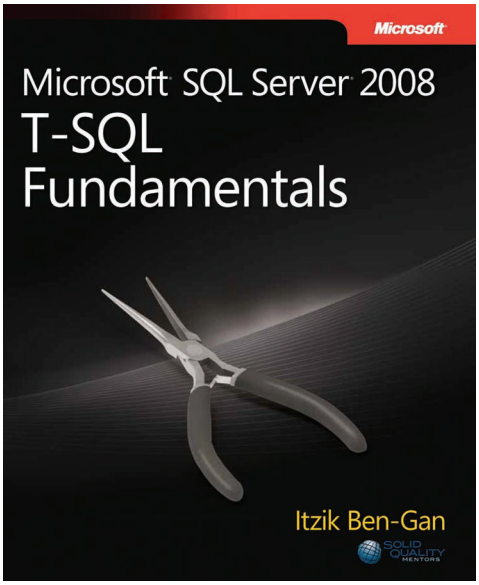

Microsoft® SQL Server® 2008 T-SQL Fundamentals

Itzik Ben-Gan
(Solid Quality Mentors)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12806.aspx

9780735626010

Microsoft Press

Table of Contents

Chapter 5

Table Expressions

In this chapter:

Table expressions are named query expressions that represent a valid relational table. You can use them in data manipulation statements similar to other tables. Microsoft SQL Server supports four types of table expressions: derived tables, common table expressions (CTEs), views, and inline table-valued functions (inline TVFs), each of which I will describe in detail in this chapter. The focus of this chapter is SELECT queries against table expressions; Chapter 8, "Data Modification," covers modifications against table expressions.

Table expressions are not physically materialized anywhere—they are virtual. A query against a table expression is internally translated to a query against the underlying objects. The benefits of using table expressions are typically related to logical aspects of your code and not to performance. For example, table expressions help you simplify your solutions by using a modular approach. Table expressions also help you circumvent certain restrictions in the language, such as the inability to refer to column aliases assigned in the SELECT clause in query clauses that are logically processed prior to the SELECT clause.

This chapter also introduces the APPLY table operator used in conjunction with a table expression. I will explain how to use this operator to apply a table expression to each row of another table.

Derived Tables

Derived tables (also known as table subqueries) are defined in the FROM clause of an outer query. Their scope of existence is the outer query. As soon as the outer query is finished, the derived table is gone.

You specify the query defining the derived table within parentheses, followed by the AS clause and the derived table name. For example, the following code defines a derived table called USACusts based on a query that returns all customers from the United States, and the outer query selects all rows from the derived table:

```
USE TSQLFundamentals2008;

SELECT *
FROM (SELECT custid, companyname
      FROM Sales.Customers
      WHERE country = N'USA') AS USACusts;
```

In this particular case, which is a simple example of the basic syntax, a derived table is not needed because the outer query doesn't apply any manipulation.

The code in this basic example returns the following output:

```
custid      companyname
----------- ---------------
32          Customer YSIQX
36          Customer LVJSO
43          Customer UISOJ
45          Customer QXPPT
48          Customer DVFMB
55          Customer KZQZT
65          Customer NYUHS
71          Customer LCOUJ
75          Customer XOJYP
77          Customer LCYBZ
78          Customer NLTYP
82          Customer EYHKM
89          Customer YBQTI
```

A query must meet three requirements to be valid to define a table expression of any kind:

1. **Order is not guaranteed.** A table expression is supposed to represent a relational table, and the rows in a relational table have no guaranteed order. Recall that this aspect of a relation stems from set theory. For this reason, ANSI SQL disallows an ORDER BY clause in queries that are used to define table expressions. T-SQL follows this restriction for the most part, with one exception—when TOP is also specified. In the context of a query with the TOP option, the ORDER BY clause serves a logical purpose: defining for the TOP option which rows to filter. If you use a query with TOP and ORDER BY to define a table expression, ORDER BY is only guaranteed to serve the logical filtering purpose for the TOP option and not the usual presentation purpose. If the outer query against the table expression does not have a presentation ORDER BY, the output is not guaranteed to be returned in any particular order. The section "Views and the ORDER BY Clause," later in this chapter, provides more detail on this item.
2. **All columns must have names.** All columns in a table must have names; therefore, you must assign column aliases to all expressions in the SELECT list of the query that is used to define a table expression.

3. **All column names must be unique.** All column names in a table must be unique; therefore, a table expression that has multiple columns with the same name is invalid. This might happen when the query defining the table expression joins two tables, and both tables have a column with the same name. If you need to incorporate both columns in your table expression, they must have different column names. You can resolve this by assigning the two columns with different column aliases.

Assigning Column Aliases

One of the benefits of using table expressions is that in any clause of the outer query you can refer to column aliases that were assigned in the SELECT clause of the inner query. This helps you get around the fact that you can't refer to column aliases assigned in the SELECT clause in query clauses that are logically processed prior to the SELECT clause (for example, WHERE or GROUP BY).

For example, suppose that you need to write a query against the Sales.Orders table and return the number of distinct customers handled in each order year. The following attempt is invalid because the GROUP BY clause refers to a column alias that was assigned in the SELECT clause, and the GROUP BY clause is logically processed prior to the SELECT clause:

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

You could solve the problem by referring to the expression YEAR(orderdate) in both the GROUP BY and the SELECT clauses, but this is an example with a short expression. What if the expression were much longer? Maintaining two copies of the same expression might hurt code readability and maintainability and is more prone to errors. To solve the problem in a way that requires only one copy of the expression, you can use a table expression like so:

LISTING 5-1 Query with a Derived Table Using Inline Aliasing Form

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

This query returns the following output:

```
orderyear   numcusts
----------- -----------
2006        67
2007        86
2008        81
```

This code defines a derived table called D based on a query against the Orders table that returns the order year and customer ID from all rows. The SELECT list of the inner query uses inline aliasing format to assign the alias orderyear to the expression YEAR(orderdate). The outer query can refer to the orderyear column alias in both the GROUP BY and SELECT clauses, because as far as the outer query is concerned, it queries a table called D with columns called orderyear and custid.

As I mentioned earlier, SQL Server expands the definition of the table expression and accesses the underlying objects directly. After expansion, the query in Listing 5-1 looks like this:

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

This is just to emphasize that you use table expressions for logical (not performance-related) reasons. Generally speaking, table expressions have neither positive nor negative performance impact.

The code in Listing 5-1 uses the inline aliasing format to assign column aliases to expressions. The syntax for inline aliasing is <expression> [AS] <alias>. Note that the word AS is optional in the syntax for inline aliasing; however, I find that it helps the readability of the code and recommend using it.

In some cases, you might prefer to use a second supported form for assigning column aliases, which you can think of as an external form. With this form you do not assign column aliases following the expressions in the SELECT list—you specify all target column names in parentheses following the table expression's name like so:

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

It is generally recommended that you use the inline form for a couple of reasons. If you need to debug the code when using the inline form, when you highlight the query defining the table expression and run it, the columns in the result appear with the aliases you assigned. With the external form, you cannot include the target column names when you highlight the table expression query, so the result appears with no column names in the case of the unnamed expressions. Also, when the table expression query is lengthy, using the external form it can be quite difficult to figure out which column alias belongs to which expression.

Even though it's a best practice to use the inline aliasing form, in some cases you may find the external form more convenient to work with. For example, when the query defining the table expression isn't going to undergo any further revisions and you want to treat it like a "black box"—you want to focus your attention on the table expression name followed by the target column list when you look at the outer query.

Using Arguments

In the query defining a derived table, you can refer to arguments. The arguments can be local variables and input parameters to a routine such as a stored procedure or function. For example, the following code declares and initializes a local variable called *@empid*, and the query in the code that is used to define the derived table D refers to the local variable in the WHERE clause:

```
DECLARE @empid AS INT = 3;

/*
-- Prior to SQL Server 2008 use separate DECLARE and SET statements:
DECLARE @empid AS INT;
SET @empid = 3;
*/

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

This query returns the number of distinct customers per year that handled the orders of the input employee (the employee whose ID is stored in the variable *@empid*). Here's the output of this query:

```
orderyear   numcusts
----------- -----------
2006        16
2007        46
2008        30
```

Nesting

If you need to define a derived table using a query that by itself refers to a derived table, you end up nesting derived tables. Nesting of derived tables is a result of the fact that a derived table is defined in the FROM clause of the outer query and not separately. Nesting is a problematic aspect of programming in general as it tends to complicate the code and reduce its readability.

For example, the code in Listing 5-2 returns order years and the number of customers handled in each year only for years in which more than 70 customers were handled:

LISTING 5-2 Query with Nested Derived Tables

```
SELECT orderyear, numcusts
FROM (SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
      FROM (SELECT YEAR(orderdate) AS orderyear, custid
            FROM Sales.Orders) AS D1
      GROUP BY orderyear) AS D2
WHERE numcusts > 70;
```

This code returns the following output:

```
orderyear   numcusts
----------- -----------
2007        86
2008        81
```

The purpose of the innermost derived table, D1, is to assign the column alias orderyear to the expression YEAR(orderdate). The query against D1 refers to orderyear in both the GROUP BY and SELECT clauses, and assigns the column alias numcusts to the expression COUNT(DISTINCT custid). The query against D1 is used to define the derived table D2. The query against D2 refers to numcusts in the WHERE clause to filter order years in which more than 70 customers were handled.

The whole purpose of using table expressions in this example was to simplify the solution by reusing column aliases instead of repeating expressions. However, with the complexity added by the nesting aspect of derived tables, I'm not sure that the solution is simpler than the alternative, which does not make any use of derived tables but instead repeats expressions:

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR(orderdate)
HAVING COUNT(DISTINCT custid) > 70;
```

In short, nesting is a problematic aspect of derived tables.

Multiple References

Another problematic aspect of derived tables stems from the fact that derived tables are defined in the FROM clause of the outer query and not prior to the outer query. As far as the FROM clause of the outer query is concerned, the derived table doesn't exist yet; therefore, if you need to refer to multiple instances of the derived table, you can't. Instead, you have to define multiple derived tables based on the same query. The query in Listing 5-3 provides an example:

LISTING 5-3 Multiple Derived Tables Based on the Same Query

```
SELECT Cur.orderyear,
  Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
  Cur.numcusts - Prv.numcusts AS growth
FROM (SELECT YEAR(orderdate) AS orderyear,
        COUNT(DISTINCT custid) AS numcusts
      FROM Sales.Orders
      GROUP BY YEAR(orderdate)) AS Cur
  LEFT OUTER JOIN
     (SELECT YEAR(orderdate) AS orderyear,
        COUNT(DISTINCT custid) AS numcusts
      FROM Sales.Orders
      GROUP BY YEAR(orderdate)) AS Prv
    ON Cur.orderyear = Prv.orderyear + 1;
```

This query joins two instances of a table expression to create two derived tables: the first derived table, Cur, represents current years, and the second derived table, Prv, represents previous years. The join condition Cur.orderyear = Prv.orderyear + 1 ensures that each row from the first derived table matches with the previous year of the second. By making it a LEFT outer join, the first year that has no previous year is also returned from the Cur table. The SELECT clause of the outer query calculates the difference between the number of customers handled in the current and previous years.

The code in Listing 5-3 produces the following output:

```
orderyear   curnumcusts prvnumcusts growth
----------- ----------- ----------- -----------
2006        67          NULL        NULL
2007        86          67          19
2008        81          86          -5
```

The fact that you cannot refer to multiple instances of the same derived table forces you to maintain multiple copies of the same query definition. This leads to lengthy code that is hard to maintain and is prone to errors.

Common Table Expressions

Common table expressions (CTEs) are another form of table expression very similar to derived tables, yet with a couple of important advantages. CTEs were introduced in SQL Server 2005 and are part of ANSI SQL:1999 and later standards.

CTEs are defined using a *WITH* statement and have the following general form:

```
WITH <CTE_Name>[(<target_column_list>)]
AS
(
  <inner_query_defining_CTE>
)
<outer_query_against_CTE>;
```

The inner query defining the CTE must follow all requirements mentioned earlier to be valid to define a table expression. As a simple example, the following code defines a CTE called USACusts based on a query that returns all customers from the United States, and the outer query selects all rows from the CTE:

```
WITH USACusts AS
(
  SELECT custid, companyname
  FROM Sales.Customers
  WHERE country = N'USA'
)
SELECT * FROM USACusts;
```

As with derived tables, as soon as the outer query finishes, the CTE gets out of scope.

Note The WITH clause is used in T-SQL for several different purposes. To avoid ambiguity, when the WITH clause is used to define a CTE, the preceding statement in the same batch—if one exists—must be terminated with a semicolon. And oddly enough, the semicolon for the entire CTE is not required, though I still recommend specifying it.

Assigning Column Aliases

CTEs also support two forms of column aliasing—inline and external. For the inline form, specify <expression> AS <column_alias>; for the external form, specify the target column list in parentheses immediately after the CTE name.

Here's an example of the inline form:

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

And here's an example of the external form:

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

The motivations for using one form or the other are similar to those described in the context of derived tables.

Using Arguments

As with derived tables, you can also use arguments in the query used to define a CTE. Here's an example:

```
DECLARE @empid AS INT = 3;

/*
-- Prior to SQL Server 2008 use separate DECLARE and SET statements:
DECLARE @empid AS INT;
SET @empid = 3;
*/
```

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

Defining Multiple CTEs

On the surface, the difference between derived tables and CTEs might seem to be merely semantic. However, the fact that you first define a CTE and then use it gives it several important advantages over derived tables. One of those advantages is that if you need to refer to one CTE from another, you don't end up nesting them like derived tables. Instead, you simply define multiple CTEs separated by commas under the same *WITH* statement. Each CTE can refer to all previously defined CTEs, and the outer query can refer to all CTEs. For example, the following code is the CTE alternative to the nested derived tables approach in Listing 5-2:

```
WITH C1 AS
(
  SELECT YEAR(orderdate) AS orderyear, custid
  FROM Sales.Orders
),
C2 AS
(
  SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
  FROM C1
  GROUP BY orderyear
)
SELECT orderyear, numcusts
FROM C2
WHERE numcusts > 70;
```

Because you define a CTE before you use it, you don't end up nesting CTEs. Each CTE appears separately in the code in a modular manner. This modular approach substantially improves the readability and maintainability of the code compared to the nested derived table approach.

Technically you cannot nest CTEs, nor can you define a CTE within the parentheses of a derived table. However, nesting is a problematic practice; therefore, think of these restrictions as aids to code clarity rather than obstacles.

Multiple References

The fact that a CTE is defined first and then queried has another advantage: As far as the FROM clause of the outer query is concerned, the CTE already exists; therefore, you

can refer to multiple instances of the same CTE. For example, the following code is the logical equivalent of the code shown earlier in Listing 5-3, using CTEs instead of derived tables:

```
WITH YearlyCount AS
(
  SELECT YEAR(orderdate) AS orderyear,
    COUNT(DISTINCT custid) AS numcusts
  FROM Sales.Orders
  GROUP BY YEAR(orderdate)
)
SELECT Cur.orderyear,
  Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
  Cur.numcusts - Prv.numcusts AS growth
FROM YearlyCount AS Cur
  LEFT OUTER JOIN YearlyCount AS Prv
    ON Cur.orderyear = Prv.orderyear + 1;
```

As you can see, the CTE YearlyCount is defined once and accessed twice in the FROM clause of the outer query—once as Cur and once as Prv. You need to maintain only one copy of the CTE query and not multiple copies as you would with derived tables.

If you're curious about performance, recall that earlier I mentioned that typically table expressions have no performance impact because they are not physically materialized anywhere. Both references to the CTE here are going to be expanded. Internally, this query has a self join between two instances of the Orders table, each of which involves scanning the table data and aggregating it before the join—the same physical processing that takes place with the derived table approach.

Recursive CTEs

This section is optional because it covers subjects that are beyond the fundamentals.

CTEs are unique among table expressions because they have recursive capabilities. A recursive CTE is defined by at least two queries (more are possible)—at least one query known as the *anchor member* and at least one query known as the *recursive member*. The general form of a basic recursive CTE looks like this:

```
WITH <CTE_Name>[(<target_column_list>)]
AS
(
  <anchor_member>
  UNION ALL
  <recursive_member>
)
<outer_query_against_CTE>;
```

The anchor member is a query that returns a valid relational result table—like a query that is used to define a nonrecursive table expression. The anchor member query is invoked only once.

The recursive member is a query that has a reference to the CTE name. The reference to the CTE name represents what is logically the previous result set in a sequence of executions. The first time that the recursive member is invoked, the previous result set represents whatever the anchor member returned. In each subsequent invocation of the recursive member, the reference to the CTE name represents the result set returned by the previous invocation of the recursive member. The recursive member has no explicit recursion termination check—the termination check is implicit. The recursive member is invoked repeatedly until it returns an empty set, or exceeds some limit.

Both queries must be compatible in terms of the number of columns they return and the data types of the corresponding columns.

The reference to the CTE name in the outer query represents the unified result sets of the invocation of the anchor member and all invocations of the recursive member.

If this is your first encounter with recursive CTEs, you might find this explanation hard to understand. They are best explained with an example. The following code demonstrates how to use a recursive CTE to return information about an employee (Don Funk, employee ID 2) and all of the employee's subordinates in all levels (direct or indirect):

```
WITH EmpsCTE AS
(
  SELECT empid, mgrid, firstname, lastname
  FROM HR.Employees
  WHERE empid = 2

  UNION ALL

  SELECT C.empid, C.mgrid, C.firstname, C.lastname
  FROM EmpsCTE AS P
    JOIN HR.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

The anchor member queries the HR.Employees table and simply returns the row for employee 2:

```
  SELECT empid, mgrid, firstname, lastname
  FROM HR.Employees
  WHERE empid = 2
```

The recursive member joins the CTE—representing the previous result set—with the Employees table to return the direct subordinates of the employees returned in the previous result set:

```
  SELECT C.empid, C.mgrid, C.firstname, C.lastname
  FROM EmpsCTE AS P
    JOIN HR.Employees AS C
      ON C.mgrid = P.empid
```

In other words, the recursive member is invoked repeatedly, and in each invocation it returns the next level of subordinates. The first time the recursive member is invoked it returns the direct subordinates of employee 2—employees 3 and 5. The second time the recursive member is invoked, it returns the direct subordinates of employees 3 and 5—employees 4, 6, 7, 8, and 9. The third time the recursive member is invoked, there are no more subordinates; the recursive member returns an empty set and therefore recursion stops.

The reference to the CTE name in the outer query represents the unified result sets; in other words, employee 2 and all of the employee's subordinates.

Here's the output of this code:

```
empid       mgrid       firstname  lastname
----------- ----------- ---------- --------------------
2           1           Don        Funk
3           2           Judy       Lew
5           2           Sven       Buck
6           5           Paul       Suurs
7           5           Russell    King
9           5           Zoya       Dolgopyatova
4           3           Yael       Peled
8           3           Maria      Cameron
```

In the event of a logical error in the join predicate in the recursive member, or problems with the data resulting in cycles, the recursive member can potentially be invoked an infinite number of times. As a safety measure, by default SQL Server restricts the number of times that the recursive member can be invoked to 100. The code will fail upon the 101st invocation of the recursive member. You can change the default maximum recursion limit by specifying the hint OPTION(MAXRECURSION *n*) at the end of the outer query, where *n* is an integer in the range 0 through 32,767 representing the maximum recursion limit you want to set. If you want to remove the restriction altogether, specify MAXRECURSION 0. Note that SQL Server stores the intermediate result sets returned by the anchor and recursive members in a work table in tempdb; if you remove the restriction and have a runaway query, the work table will quickly get very large. If tempdb can't grow anymore—for example, when you run out of disk space—the query will fail.

Views

The two types of table expressions discussed so far—derived tables and CTEs—have a very limited scope, which is the single statement scope. As soon as the outer query against those table expressions is finished, they are gone. This means that derived tables and CTEs are not reusable.

Views and inline table-valued functions (inline TVFs) are two reusable types of table expressions; their definition is stored as a database object. Once created, those objects are permanent parts of the database and are only removed from the database if explicitly dropped.

In most other respects, views and inline TVFs are treated like derived tables and CTEs. For example, when querying a view or an inline TVF, SQL Server expands the definition of the table expression and queries the underlying objects directly, as with derived tables and CTEs.

In this section, I'll describe views; in the next section, I'll describe inline TVFs. As I mentioned earlier, a view is a reusable table expression whose definition is stored in the database. For example, the following code creates a view called USACusts in the Sales schema in the TSQLFundamentals2008 database, representing all customers from the United States:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Sales.USACusts') IS NOT NULL
  DROP VIEW Sales.USACusts;
GO
CREATE VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Note that just as with derived tables and CTEs, instead of using inline column aliasing as shown in the preceding code, you can use external column aliasing by specifying the target column names in parentheses immediately after the view name.

Once you create this view, you can query it much like you query other tables in the database:

```
SELECT custid, companyname
FROM Sales.USACusts;
```

Because a view is an object in the database, you can control access to the view with permissions just like other objects that can be queried (for example, SELECT, INSERT, UPDATE, and DELETE permissions). For example, you can deny direct access to the underlying objects while granting access to the view.

Note that the general recommendation to avoid using SELECT * has specific relevance in the context of views. The columns are enumerated in the compiled form of the view and new table columns will not be automatically added to the view. For example, suppose you define a view based on the query SELECT * FROM dbo.T1, and at the view creation time the table T1 has the columns col1 and col2. SQL Server stores information only on those two columns in the view's metadata. If you alter the definition of the table adding new columns, those new columns will not be added to the view. You can refresh the view's metadata using a stored procedure called sp_refreshview, but to avoid confusion, the best practice is to explicitly list the column names that you need in the definition of the view. If columns are added to the underlying tables and you need them in the view, use the *ALTER VIEW* statement to revise the view definition accordingly.

Views and the ORDER BY Clause

The query that you use to define a view must meet all requirements mentioned earlier with respect to table expressions in the context of derived tables. The view should not guarantee any order to the rows, all view columns must have names, and all column names must be unique. In this section, I'll elaborate a bit about the ordering issue, which is a fundamental point that is crucial to understand.

Remember that a presentation ORDER BY clause is not allowed in the query defining a table expression because there's no order among the rows of a relational table. An attempt to create an ordered view is absurd because it violates fundamental properties of a relation as defined by the relational model. If you need to return rows from a view sorted for presentation purposes, you shouldn't try to make the view something it shouldn't be. Instead, you should specify a presentation ORDER BY clause in the outer query against the view, like so:

```
SELECT custid, companyname, region
FROM Sales.USACusts
ORDER BY region;
```

Try running the following code to create a view with a presentation ORDER BY clause:

```
ALTER VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

This attempt fails and you get the following error:

```
Msg 1033, Level 15, State 1, Procedure USACusts, Line 9
The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and
common table expressions, unless TOP or FOR XML is also specified.
```

The error message indicates that SQL Server allows the ORDER BY clause in two exceptional cases—when the TOP or FOR XML options are used. Neither case follows the SQL standard, and in both cases the ORDER BY clause serves a purpose beyond the usual presentation purpose.

Because T-SQL allows an ORDER BY clause in a view when TOP is also specified, some people think that they can create "ordered views" by using TOP (100) PERCENT like so:

```
ALTER VIEW Sales.USACusts
AS

SELECT TOP (100)
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
```

```
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

Even though the code is technically valid and the view is created, you should be aware that because the query is used to define a table expression, the ORDER BY clause here is only guaranteed to serve the logical filtering purpose for the TOP option. If you query the view and don't specify an ORDER BY clause in the outer query, presentation order is not guaranteed.

For example, run the following query against the view:

```
SELECT custid, companyname, region
FROM Sales.USACusts;
```

Here is the output from one of my executions showing that the rows are not sorted by region:

```
custid      companyname              region
----------- ------------------------ ---------------
32          Customer YSIQX           OR
36          Customer LVJSO           OR
43          Customer UISOJ           WA
45          Customer QXPPT           CA
48          Customer DVFMB           OR
55          Customer KZQZT           AK
65          Customer NYUHS           NM
71          Customer LCOUJ           ID
75          Customer XOJYP           WY
77          Customer LCYBZ           OR
78          Customer NLTYP           MT
82          Customer EYHKM           WA
89          Customer YBQTI           WA
```

In some cases a query that is used to define a table expression has the TOP option with an ORDER BY clause, and the query against the table expression doesn't have an ORDER BY clause. In those cases, therefore, the output might or might not be returned in the specified order. If the results happen to be ordered, it may be due to optimization reasons, especially when you use values other than TOP (100) PERCENT. The point I'm trying to make is that any order of the rows in the output is considered valid, and no specific order is guaranteed; therefore, when querying a table expression, you should not assume any order unless you specify an ORDER BY clause in the outer query.

Do not confuse the behavior of a query that is used to define a table expression with a query that isn't. A query with TOP and ORDER BY does not guarantee presentation order only in the context of a table expression. In the context of a query that is not used to define a table expression, the ORDER BY clause serves both the logical filtering purpose for the TOP option and the presentation purpose.

View Options

When you create or alter a view, you can specify view attributes and options as part of the view definition. In the header of the view under the WITH clause you can specify attributes such as ENCRYPTION and SCHEMABINDING, and at the end of the query you can specify WITH CHECK OPTION. The following sections describe the purpose of these options.

The ENCRYPTION Option

The ENCRYPTION option is available when you create or alter views, stored procedures, triggers, and user-defined functions (UDFs). The ENCRYPTION option indicates that SQL Server will internally store the text with the definition of the object in an obfuscated format. The obfuscated text is not directly visible to users through any of the catalog objects—only to privileged users through special means.

Before you look at the ENCRYPTION option, run the following code to alter the definition of the USACusts view to its original version:

```
ALTER VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

To get the definition of the view, invoke the *OBJECT_DEFINITION* function like so:

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

The text with the definition of the view is available because the view was created without the ENCRYPTION option. You get the following output:

```
CREATE VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
```

Next, alter the view definition—only this time, include the ENCRYPTION option:

```
ALTER VIEW Sales.USACusts WITH ENCRYPTION
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
```

```
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Try again to get the text with the definition of the view:

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

This time you get a NULL back.

As an alternative to the *OBJECT_DEFINITION* function, you can use the sp_helptext stored procedure to get object definitions. The *OBJECT_DEFINITION* function was added in SQL Server 2005 while sp_helptext was also available in earlier versions. For example, the following code requests the object definition of the USACusts view:

```
EXEC sp_helptext 'Sales.USACusts';
```

Because in our case the view was created with the ENCRYPTION option, you will not get the object definition back, but the following message:

```
The text for object 'Sales.USACusts' is encrypted.
```

The SCHEMABINDING Option

The SCHEMABINDING option is available to views and UDFs, and it binds the schema of referenced objects and columns to the schema of the referencing object. It indicates that referenced objects cannot be dropped and that referenced columns cannot be dropped or altered.

For example, alter the USACusts view with the SCHEMABINDING option:

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Now try to drop the Address column from the Customers table:

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

You get the following error:

```
Msg 5074, Level 16, State 1, Line 1
The object 'USACusts' is dependent on column 'address'.
Msg 4922, Level 16, State 9, Line 1
ALTER TABLE DROP COLUMN address failed because one or more objects access this column.
```

Without the SCHEMABINDING option, such a schema change would have been allowed, as well as dropping the Customers table altogether. This can lead to errors at run time when

you try to query the view, and referenced objects or columns that do not exist. If you create the view with the SCHEMABINDING option, you can avoid these errors.

The object definition must meet a couple of technical requirements to support the SCHEMABINDING option. The query is not allowed to use * in the SELECT clause; instead, you have to explicitly list column names. Also, you must use schema-qualified two-part names when referring to objects. Both requirements are actually good practices in general.

As you can imagine, creating your objects with the SCHEMABINDING option is a good practice.

The Option CHECK OPTION

The purpose of CHECK OPTION is to prevent modifications through the view that conflict with the view's filter—assuming that one exists in the query defining the view.

The query defining the view USACusts filters customers where the country attribute is equal to N'USA'. The view is currently defined without CHECK OPTION. This means that you can currently insert rows through the view with customers from countries other than the United States, and you can update existing customers through the view, changing their country to one other than the United States. For example, the following code successfully inserts a customer with company name Customer ABCDE from the United Kingdom through the view:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
 VALUES(
  N'Customer ABCDE', N'Contact ABCDE', N'Title ABCDE', N'Address ABCDE',
  N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

The row was inserted through the view into the Customers table. However, because the view filters only customers from the United States, if you query the view looking for the new customer you get an empty set back:

```
SELECT custid, companyname, country
FROM Sales.USACusts
WHERE companyname = N'Customer ABCDE';
```

Query the Customers table directly looking for the new customer:

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE companyname = N'Customer ABCDE';
```

You get the customer information in the output, because the new row made it to the Customers table:

```
custid      companyname        country
----------- ------------------ ---------------
92          Customer ABCDE     UK
```

Similarly, if you update a customer row through the view, changing the country attribute to a country other than the United States, the update makes it to the table. But that customer doesn't show up anymore in the view because it doesn't qualify to the view's query filter.

If you want to prevent modifications that conflict with the view's filter, add WITH CHECK OPTION at the end of the query defining the view:

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
WITH CHECK OPTION;
GO
```

Now try to insert a row that conflicts with the view's filter:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
 VALUES(
  N'Customer FGHIJ', N'Contact FGHIJ', N'Title FGHIJ', N'Address FGHIJ',
  N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

You get the following error:

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH CHECK
OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from
the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

When you're done, run the following code for cleanup:

```
DELETE FROM Sales.Customers
WHERE custid > 91;

DBCC CHECKIDENT('Sales.Customers', RESEED, 91);

IF OBJECT_ID('Sales.USACusts') IS NOT NULL DROP VIEW Sales.USACusts;
```

Inline Table-Valued Functions

Inline TVFs are reusable table expressions that support input parameters. In all respects except for the support for input parameters, inline TVFs are similar to views. For this reason, I like to think of inline TVFs as parameterized views, even though they are not called this formally.

For example, the following code creates an inline TVF called *fn_GetCustOrders* in the TSQLFundamentals2008 database:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('dbo.fn_GetCustOrders') IS NOT NULL
  DROP FUNCTION dbo.fn_GetCustOrders;
GO
CREATE FUNCTION dbo.fn_GetCustOrders
  (@cid AS INT) RETURNS TABLE
AS
RETURN
  SELECT orderid, custid, empid, orderdate, requireddate,
    shippeddate, shipperid, freight, shipname, shipaddress, shipcity,
    shipregion, shippostalcode, shipcountry
  FROM Sales.Orders
  WHERE custid = @cid;
GO
```

This inline TVF accepts an input parameter called *@cid* representing a customer ID, and returns all orders that were placed by the input customer. You query inline TVFs like you query other tables with DML statements. If the function accepts input parameters, you specify those in parentheses following the function's name. Also, make sure you provide an alias to the table expression. Providing a table expression with an alias is not always a requirement but is a good practice because it makes your code more readable and less prone to errors. For example, the following code queries the function requesting all orders that were placed by customer 1:

```
SELECT orderid, custid
FROM dbo.fn_GetCustOrders(1) AS CO;
```

This code returns the following output:

```
orderid     custid
----------- -----------
10643       1
10692       1
10702       1
10835       1
10952       1
11011       1
```

As with other tables, you can refer to an inline TVF as part of a join. For example, the following query joins the inline TVF returning customer 1's orders with the Sales.OrderDetails table, matching customer 1's orders with the related order lines:

```
SELECT CO.orderid, CO.custid, OD.productid, OD.qty
FROM dbo.fn_GetCustOrders(1) AS CO
  JOIN Sales.OrderDetails AS OD
    ON CO.orderid = OD.orderid;
```

This code returns the following output:

```
orderid     custid      productid   qty
----------- ----------- ----------- ------
10643       1           28          15
10643       1           39          21
10643       1           46           2
10692       1           63          20
10702       1            3           6
10702       1           76          15
10835       1           59          15
10835       1           77           2
10952       1            6          16
10952       1           28           2
11011       1           58          40
11011       1           71          20
```

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.fn_GetCustOrders') IS NOT NULL
  DROP FUNCTION dbo.fn_GetCustOrders;
```

The APPLY Operator

The APPLY operator is a nonstandard table operator that was introduced in SQL Server 2005. This operator is used in the FROM clause of a query like all table operators. The two supported types of the APPLY operator are CROSS APPLY and OUTER APPLY. CROSS APPLY implements only one logical query processing phase, while OUTER APPLY implements two.

The APPLY operator operates on two input tables, the second of which may be a table expression; I'll refer to them as the left and right tables. The right table is usually a derived table or an inline TVF. The CROSS APPLY operator implements one logical query processing phase—it applies the right table expression to each row from the left table, and produces a result table with the unified result sets.

So far it might sound like the CROSS APPLY operator is very similar to a cross join, and in a sense that's true. For example, the following two queries return the same result sets:

```
SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
  CROSS JOIN HR.Employees AS E;

SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
  CROSS APPLY HR.Employees AS E;
```

However, with the CROSS APPLY operator the right table expression can represent a different set of rows per each row from the left table, unlike in a join. You can achieve this when you use a derived table in the right side, and in the derived table query refer to attributes from the left side. Or when you use an inline TVF, you can pass attributes from the left side as input arguments.

For example, the following code uses the CROSS APPLY operator to return the three most recent orders for each customer:

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
  CROSS APPLY
    (SELECT TOP(3) orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS A;
```

You can think of the table expression A as a correlated table subquery. In terms of logical query processing, the right table expression (derived table in our case) is applied to each row from the Customers table. Notice the reference to the attribute C.custid from the left table in the derived table's query filter. The derived table returns the three most recent orders for the customer from the current left row. Because the derived table is applied to each row from the left side, the CROSS APPLY operator returns the three most recent orders for each customer.

Here's the output of this query, shown here in abbreviated form:

```
custid      orderid     orderdate
----------- ----------- -----------------------
1           11011       2008-04-09 00:00:00.000
1           10952       2008-03-16 00:00:00.000
1           10835       2008-01-15 00:00:00.000
2           10926       2008-03-04 00:00:00.000
2           10759       2007-11-28 00:00:00.000
2           10625       2007-08-08 00:00:00.000
3           10856       2008-01-28 00:00:00.000
3           10682       2007-09-25 00:00:00.000
3           10677       2007-09-22 00:00:00.000
...

(263 row(s) affected)
```

If the right table expression returns an empty set, the CROSS APPLY operator does not return the corresponding left row. For example, customers 22 and 57 did not place orders. In both cases the derived table is an empty set; therefore, those customers are not returned in the output. If you want to return rows from the left table for which the right table expression returns an empty set, use the OUTER APPLY operator instead of CROSS APPLY. The OUTER APPLY operator adds a second logical phase that identifies rows from the left side for which the right table expression returns an empty set, and adds those rows to the result table as outer rows with NULLs in the right side's attributes as place holders. In a sense, this phase is similar to the phase that adds outer rows in a left outer join.

For example, run the following code to return the three most recent orders for each customer, and include in the output customers with no orders as well:

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
  OUTER APPLY
```

```
    (SELECT TOP(3) orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS A;
```

This time, customers 22 and 57, who did not place orders, are included in the output, which is shown here in abbreviated form:

```
custid      orderid     orderdate
----------- ----------- -----------------------
1           11011       2008-04-09 00:00:00.000
1           10952       2008-03-16 00:00:00.000
1           10835       2008-01-15 00:00:00.000
2           10926       2008-03-04 00:00:00.000
2           10759       2007-11-28 00:00:00.000
2           10625       2007-08-08 00:00:00.000
3           10856       2008-01-28 00:00:00.000
3           10682       2007-09-25 00:00:00.000
3           10677       2007-09-22 00:00:00.000
...
22          NULL        NULL
...
57          NULL        NULL
...

(265 row(s) affected)
```

For encapsulation purposes you may find it more convenient to work with inline TVFs instead of derived tables. This way your code will be simpler to follow and maintain. For example, the following code creates an inline TVF called *fn_TopOrders* that accepts as inputs a customer ID (@*custid*) and a number (@*n*), and returns the @*n* most recent orders for customer @*custid*:

```
IF OBJECT_ID('dbo.fn_TopOrders') IS NOT NULL
  DROP FUNCTION dbo.fn_TopOrders;
GO
CREATE FUNCTION dbo.fn_TopOrders
  (@custid AS INT, @n AS INT)
  RETURNS TABLE
AS
RETURN
  SELECT TOP(@n) orderid, empid, orderdate, requireddate
  FROM Sales.Orders
  WHERE custid = @custid
  ORDER BY orderdate DESC, orderid DESC;
GO
```

You can now substitute the use of the derived table from the previous examples with the new function:

```
SELECT
  C.custid, C.companyname,
  A.orderid, A.empid, A.orderdate, A.requireddate
FROM Sales.Customers AS C
  CROSS APPLY dbo.fn_TopOrders(C.custid, 3) AS A;
```

The code is much more readable and easier to maintain. In terms of physical processing, nothing really changed because, as I stated earlier, the definition of table expressions is expanded, and SQL Server will in any case end up querying the underlying objects directly.

Conclusion

Table expressions can help you simplify your code, improve its maintainability, and encapsulate querying logic. When you need to use table expressions and are not planning to reuse their definitions, use derived tables or CTEs. CTEs have a couple of advantages over derived tables; you do not nest CTEs as you do derived tables, making CTEs more modular and easier to maintain. Also, you can refer to multiple instances of the same CTE, which you cannot do with derived tables.

When you need to define reusable table expressions, use views or inline TVFs. When you do not need to support input parameters, use views; otherwise, use inline TVFs.

Use the APPLY operator when you want to apply a table expression to each row from a source table, and unify all result sets into one result table.

Exercises

This section provides exercises to help you familiarize yourself with the subjects discussed in this chapter. All the exercises in this chapter require your session to be connected to the database TSQLFundamentals2008.

1-1

Write a query that returns the maximum order date for each employee.

Tables involved: TSQLFundamentals2008 database, Sales.Orders table.

Desired output:

```
empid       maxorderdate
----------- -----------------------
3           2008-04-30 00:00:00.000
6           2008-04-23 00:00:00.000
9           2008-04-29 00:00:00.000
7           2008-05-06 00:00:00.000
1           2008-05-06 00:00:00.000
4           2008-05-06 00:00:00.000
2           2008-05-05 00:00:00.000
5           2008-04-22 00:00:00.000
8           2008-05-06 00:00:00.000

(9 row(s) affected)
```

1-2

Encapsulate the query from Exercise 1-1 in a derived table. Write a join query between the derived table and the Orders table to return the orders with the maximum order date for each employee.

Tables involved: Sales.Orders.

Desired output:

```
empid       orderdate               orderid     custid
----------- ----------------------- ----------- -----------
9           2008-04-29 00:00:00.000 11058       6
8           2008-05-06 00:00:00.000 11075       68
7           2008-05-06 00:00:00.000 11074       73
6           2008-04-23 00:00:00.000 11045       10
5           2008-04-22 00:00:00.000 11043       74
4           2008-05-06 00:00:00.000 11076       9
3           2008-04-30 00:00:00.000 11063       37
2           2008-05-05 00:00:00.000 11073       58
2           2008-05-05 00:00:00.000 11070       44
1           2008-05-06 00:00:00.000 11077       65

(10 row(s) affected)
```

2-1

Write a query that calculates a row number for each order based on orderdate, orderid ordering.

Tables involved: Sales.Orders.

Desired output (abbreviated):

```
orderid     orderdate               custid      empid       rownum
----------- ----------------------- ----------- ----------- -------
10248       2006-07-04 00:00:00.000 85          5           1
10249       2006-07-05 00:00:00.000 79          6           2
10250       2006-07-08 00:00:00.000 34          4           3
10251       2006-07-08 00:00:00.000 84          3           4
10252       2006-07-09 00:00:00.000 76          4           5
10253       2006-07-10 00:00:00.000 34          3           6
10254       2006-07-11 00:00:00.000 14          5           7
10255       2006-07-12 00:00:00.000 68          9           8
10256       2006-07-15 00:00:00.000 88          3           9
10257       2006-07-16 00:00:00.000 35          4           10
...

(830 row(s) affected)
```

2-2

Write a query that returns rows with row numbers 11 through 20 based on the row number definition in Exercise 2-1. Use a CTE to encapsulate the code from Exercise 2-1.

Tables involved: Sales.Orders.

Desired output:

```
orderid     orderdate               custid      empid       rownum
----------- ----------------------- ----------- ----------- -------
10258       2006-07-17 00:00:00.000 20          1           11
10259       2006-07-18 00:00:00.000 13          4           12
10260       2006-07-19 00:00:00.000 56          4           13
10261       2006-07-19 00:00:00.000 61          4           14
10262       2006-07-22 00:00:00.000 65          8           15
10263       2006-07-23 00:00:00.000 20          9           16
10264       2006-07-24 00:00:00.000 24          6           17
10265       2006-07-25 00:00:00.000 7           2           18
10266       2006-07-26 00:00:00.000 87          3           19
10267       2006-07-29 00:00:00.000 25          4           20

(10 row(s) affected)
```

3

Write a solution using a recursive CTE that returns the management chain leading to Zoya Dolgopyatova (employee ID 9).

Tables involved: HR.Employees.

Desired output:

```
empid       mgrid       firstname  lastname
----------- ----------- ---------- --------------------
9           5           Zoya       Dolgopyatova
5           2           Sven       Buck
2           1           Don        Funk
1           NULL        Sara       Davis

(4 row(s) affected)
```

4-1

Create a view that returns the total quantity for each employee and year.

Tables involved: Sales.Orders and Sales.OrderDetails.

When running the following code:

```
SELECT * FROM Sales.VEmpOrders ORDER BY empid, orderyear;
```

The desired output is:

```
empid       orderyear   qty
----------- ----------- -----------
1           2006        1620
1           2007        3877
1           2008        2315
2           2006        1085
2           2007        2604
2           2008        2366
3           2006        940
3           2007        4436
3           2008        2476
4           2006        2212
4           2007        5273
4           2008        2313
5           2006        778
5           2007        1471
5           2008        787
6           2006        963
6           2007        1738
6           2008        826
7           2006        485
7           2007        2292
7           2008        1877
8           2006        923
8           2007        2843
8           2008        2147
9           2006        575
9           2007        955
9           2008        1140

(27 row(s) affected)
```

4-2 (Optional, Advanced)

Write a query against Sales.VEmpOrders that returns the running total quantity for each employee and year.

Tables involved: Sales.VEmpOrders view.

Desired output:

```
empid       orderyear   qty         runqty
----------- ----------- ----------- -----------
1           2006        1620        1620
1           2007        3877        5497
1           2008        2315        7812
2           2006        1085        1085
2           2007        2604        3689
2           2008        2366        6055
3           2006        940         940
```

```
3           2007        4436        5376
3           2008        2476        7852
4           2006        2212        2212
4           2007        5273        7485
4           2008        2313        9798
5           2006        778         778
5           2007        1471        2249
5           2008        787         3036
6           2006        963         963
6           2007        1738        2701
6           2008        826         3527
7           2006        485         485
7           2007        2292        2777
7           2008        1877        4654
8           2006        923         923
8           2007        2843        3766
8           2008        2147        5913
9           2006        575         575
9           2007        955         1530
9           2008        1140        2670

(27 row(s) affected)
```

5-1

Create an inline function that accepts as inputs a supplier ID (*@supid* AS *INT*) and a requested number of products (*@n* AS *INT*). The function should return *@n* products with the highest unit prices that are supplied by the given supplier ID.

Tables involved: Production.Products.

When issuing the following query:

```
SELECT * FROM Production.fn_TopProducts(5, 2);
```

Desired output:

```
productid   productname        unitprice
----------- ------------------ ---------------
12          Product OSFNS      38.00
11          Product QMVUN      21.00

(2 row(s) affected)
```

5-2

Using the CROSS APPLY operator and the function you created in Exercise 4-1, return, for each supplier, the two most expensive products.

Desired output:

```
supplierid  companyname       productid   productname     unitprice
----------- ----------------- ----------- --------------- ----------
8           Supplier BWGYE    20          Product QHFFP   81.00
8           Supplier BWGYE    68          Product TBTBL   12.50
20          Supplier CIYNM    43          Product ZZZHR   46.00
20          Supplier CIYNM    44          Product VJIEO   19.45
23          Supplier ELCRN    49          Product FPYPN   20.00
23          Supplier ELCRN    76          Product JYGFE   18.00
5           Supplier EQPNC    12          Product OSFNS   38.00
5           Supplier EQPNC    11          Product QMVUN   21.00
...

(55 row(s) affected)
```

Solutions

This section provides solutions to the exercises in the preceding section.

1-1

This exercise is just a preliminary step to the next exercise. This step involves writing a query that returns the maximum order date for each employee:

```
USE TSQLFundamentals2008;

SELECT empid, MAX(orderdate) AS maxorderdate
FROM Sales.Orders
GROUP BY empid;
```

1-2

This exercise requires you to use the query from the previous step to define a derived table, and join this derived table with the Orders table to return the orders with the maximum order date for each employee, like so:

```
SELECT O.empid, O.orderdate, O.orderid, O.custid
FROM Sales.Orders AS O
  JOIN (SELECT empid, MAX(orderdate) AS maxorderdate
        FROM Sales.Orders
        GROUP BY empid) AS D
    ON O.empid = D.empid
    AND O.orderdate = D.maxorderdate;
```

2-1

This exercise is a preliminary step to the next exercise. It requires you to query the Orders table and calculate row numbers based on orderdate, orderid ordering, like so:

```
SELECT orderid, orderdate, custid, empid,
  ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
FROM Sales.Orders;
```

2-2

This exercise requires you to define a CTE based on the query from the previous step, and filter only rows with row numbers in the range 11 through 20 from the CTE, like so:

```
WITH OrdersRN AS
(
  SELECT orderid, orderdate, custid, empid,
    ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
  FROM Sales.Orders
)
SELECT * FROM OrdersRN WHERE rownum BETWEEN 11 AND 20;
```

You might wonder why you need a table expression here. Remember that calculations based on the OVER clause (such as the *ROW_NUMBER* function) are only allowed in the SELECT and ORDER BY clauses of a query, and not directly in the WHERE clause. By using a table expression you can invoke the *ROW_NUMBER* function in the SELECT clause, assign an alias to the result column, and refer to the result column in the WHERE clause of the outer query.

3

You can think of this exercise as the inverse of the request to return an employee and all subordinates in all levels. Here, the anchor member is a query that returns the row for employee 9. The recursive member joins the CTE (call it C)—representing the subordinate/child from the previous level—with the Employees table (call it P)—representing the manager/parent in the next level. This way, each invocation of the recursive member returns the manager from the next level, until no next level manager is found (in the case of the CEO).

Here's the complete solution query:

```
WITH EmpsCTE AS
(
  SELECT empid, mgrid, firstname, lastname
  FROM HR.Employees
  WHERE empid = 9

  UNION ALL
```

```
  SELECT P.empid, P.mgrid, P.firstname, P.lastname
  FROM EmpsCTE AS C
    JOIN HR.Employees AS P
      ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

4-1

This exercise is a preliminary step to the next exercise. Here you are required to define a view based on a query that joins the Orders and OrderDetails tables, group the rows by employee ID and order year, and return the total quantity for each group. The view definition should look like this:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Sales.VEmpOrders') IS NOT NULL
  DROP VIEW Sales.VEmpOrders;
GO
CREATE VIEW  Sales.VEmpOrders
AS

SELECT
  empid,
  YEAR(orderdate) AS orderyear,
  SUM(qty) AS qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  empid,
  YEAR(orderdate);
GO
```

4-2

In this exercise, you query the VEmpOrders view and return the running total quantity for each employee and order year. To achieve this, you can write a query against the VEmpOrders view (call it V1) that returns from each row the employee ID, order year, and quantity. In the SELECT list you can incorporate a subquery against a second instance of VEmpOrders (call it V2), that returns the sum of all quantities from the rows where the employee ID is equal to the one in V1, and the order year is smaller than or equal to the one in V1. The complete solution query looks like this:

```
SELECT empid, orderyear, qty,
  (SELECT SUM(qty)
   FROM  Sales.VEmpOrders AS V2
   WHERE V2.empid = V1.empid
     AND V2.orderyear <= V1.orderyear) AS runqty
FROM  Sales.VEmpOrders AS V1
ORDER BY empid, orderyear;
```

5-1

This exercise requires you to define a function called *fn_TopProducts* that accepts a supplier ID (*@supid*) and a number (*@n*), and is supposed to return the *@n* most expensive products supplied by the input supplier ID. Here's how the function definition should look:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Production.fn_TopProducts') IS NOT NULL
  DROP FUNCTION Production.fn_TopProducts;
GO
CREATE FUNCTION Production.fn_TopProducts
  (@supid AS INT, @n AS INT)
  RETURNS TABLE
AS
RETURN
  SELECT TOP(@n) productid, productname, unitprice
  FROM Production.Products
  WHERE supplierid = @supid
  ORDER BY unitprice DESC;
GO
```

5-2

In this exercise, you write a query against the Production.Suppliers table, and use the CROSS APPLY operator to apply the function you defined by the previous step to each supplier. Your query is supposed to return the two most expensive products for each supplier. Here's the solution query:

```
SELECT S.supplierid, S.companyname, P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
  CROSS APPLY Production.fn_TopProducts(S.supplierid, 2) AS P;
```