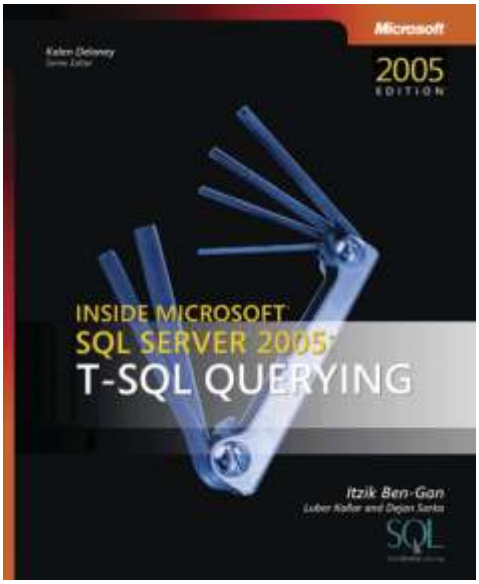

Inside Microsoft® SQL Server™ 2005: T-SQL Querying

Itzik Ben-Gan (Solid Quality Learning), Lubor Kollar, Dejan Sarka

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/mspress/books/9615.aspx

9780735623132
Publication Date: March 2006

Microsoft Press

Table of Contents

Chapter 1

Logical Query Processing

In this chapter:

Observing true experts in different fields, you will find a common practice that they all share—mastering the basics. One way or another, all professions deal with problem solving. All solutions to problems, complex as they may be, involve applying a mix of key techniques. If you want to master a profession, you need to build your knowledge upon strong foundations. Put a lot of effort in perfecting your techniques; master the basics, and you will be able to solve any problem.

This book is about Transact-SQL (T-SQL) querying—learning key techniques and applying them to solve problems. I can't think of a better way to start the book than with a chapter on fundamentals of logical query processing. I find this chapter the most important in the book—not just because it covers the essentials of query processing, but also because SQL programming is conceptually very different than any other sort of programming.

The Microsoft SQL Server dialect of SQL—Transact-SQL—follows the ANSI standard. Microsoft SQL Server 2000 conforms to the ANSI SQL:1992 standard at the Entry SQL level, and Microsoft SQL Server 2005 implements some important ANSI SQL:1999 and ANSI SQL:2003 features.

Throughout the book, I will interchangeably use the terms *SQL* and *T-SQL*. When discussing aspects of the language that originated from ANSI SQL and are relevant to most dialects, I will typically use the term *SQL*. When discussing aspects of the language with the implementation of SQL Server in mind, I'll typically use the term *T-SQL*. Note that the formal language name is *Transact-SQL*, although it is commonly called *T-SQL*. Most programmers, including myself, feel more comfortable calling it T-SQL, so I made a conscious choice of using the term *T-SQL* throughout the book.

Origin of SQL Pronunciation

Many English-speaking database professionals pronounce *SQL* as *sequel*, although the correct pronunciation of the language is *S-Q-L* ("ess kyoo ell"). One can make educated guesses about the reasoning behind the incorrect pronunciation. My guess is that there are both historical reasons and linguistic ones.

As for historical reasons, in the 1970s IBM developed a language called SEQUEL, which was an acronym for Structured English QUEry Language. The language was designed to manipulate data stored in a database system called System R, which was based on Dr. Edgar F. Codd's model for Relational Database Management Systems (RDBMS). Later on, the acronym SEQUEL was shortened to SQL because of a trademark dispute. ANSI adopted SQL as a standard in 1986, and ISO did so in 1987. ANSI declared that the official pronunciation of the language is "ess kyoo ell," but it seems that this fact is not common knowledge.

As for linguistic reasons, the sequel pronunciation is simply more fluent, mainly for English speakers. I have to say that I often use it myself for this reason.

You can sometimes guess which pronunciation people use by inspecting their writings. Someone writing "an SQL Server" probably uses the correct pronunciation, while someone writing "a SQL Server" probably uses the incorrect one.

More Info I urge you to read about the history of SQL and its pronunciation, which I find fascinating, at *http://www.wikimirror.com/SQL*. The coverage of SQL history on the Wikimirror site and in this chapter is based on an article from Wikipedia, the free encyclopedia.

There are many unique aspects of SQL programming, such as thinking in sets, the logical processing order of query elements, and three-valued logic. Trying to program in SQL without this knowledge is a straight path to lengthy, poor-performing code that is hard to maintain. This chapter's purpose is to help you understand SQL the way its designers envisioned it. You need to create strong roots upon which all the rest will be built. Where relevant, I'll explicitly indicate elements that are T-SQL specific.

Throughout the book, I will cover complex problems and advanced techniques. But in this chapter, as mentioned, I will deal only with the fundamentals of querying. Throughout the book, I also will put a lot of focus on performance. But in this chapter, I will deal only with the logical aspects of query processing. I ask you to make an effort while reading this chapter to not think about performance at all. There will be plenty of performance coverage later in the book. Some of the logical query processing phases that I'll describe in this chapter might seem very inefficient. But keep in mind that in practice, the actual physical processing of a query might be very different than the logical one.

The component in SQL Server in charge of generating the actual work plan (execution plan) for a query is the query optimizer. The optimizer determines in which order to access the tables, which access methods and indexes to use, which join algorithms to apply, and so on. The optimizer generates multiple valid execution plans and chooses the one with the lowest cost. The phases in the logical processing of a query have a very specific order. On the other hand, the optimizer can often make shortcuts in the physical execution plan that it generates. Of course, it will make shortcuts only if the result set is guaranteed to be the correct one—in other words, the same result set you would get by following the logical processing phases. For example, to use an index, the optimizer can decide to apply a filter much sooner than dictated by logical processing.

For the aforementioned reasons, it's important to make a clear distinction between logical and physical processing of a query.

Without further ado, let's delve into logical query processing phases.

Logical Query Processing Phases

This section introduces the phases involved in the logical processing of a query. I will first briefly describe each step. Then, in the following sections, I'll describe the steps in much more detail and apply them to a sample query. You can use this section as a quick reference whenever you need to recall the order and general meaning of the different phases.

Listing 1-1 contains a general form of a query, along with step numbers assigned according to the order in which the different clauses are logically processed.

Listing 1-1 Logical query processing step numbers

```
(8)  SELECT (9) DISTINCT (11) <TOP_specification> <select_list>
(1)  FROM <left_table>
(3)    <join_type> JOIN <right_table>
(2)      ON <join_condition>
(4)  WHERE <where_condition>
(5)  GROUP BY <group_by_list>
(6)  WITH {CUBE | ROLLUP}
(7)  HAVING <having_condition>
(10) ORDER BY <order_by_list>
```

The first noticeable aspect of SQL that is different than other programming languages is the order in which the code is processed. In most programming languages, the code is processed in the order in which it is written. In SQL, the first clause that is processed is the FROM clause, while the SELECT clause, which appears first, is processed almost last.

Each step generates a virtual table that is used as the input to the following step. These virtual tables are not available to the caller (client application or outer query). Only the table generated by the final step is returned to the caller. If a certain clause is not specified in a query, the

corresponding step is simply skipped. Following is a brief description of the different logical steps applied in both SQL Server 2000 and SQL Server 2005. Later in the chapter, I will discuss separately the steps that were added in SQL Server 2005.

Brief Description of Logical Query Processing Phases

Don't worry too much if the description of the steps doesn't seem to make much sense for now. These are provided as a reference. Sections that come after the scenario example will cover the steps in much more detail.

1. **FROM:** A Cartesian product (cross join) is performed between the first two tables in the FROM clause, and as a result, virtual table VT1 is generated.
2. **ON:** The ON filter is applied to VT1. Only rows for which the <*join_condition*> is TRUE are inserted to VT2.
3. **OUTER (join):** If an OUTER JOIN is specified (as opposed to a CROSS JOIN or an INNER JOIN), rows from the preserved table or tables for which a match was not found are added to the rows from VT2 as outer rows, generating VT3. If more than two tables appear in the FROM clause, steps 1 through 3 are applied repeatedly between the result of the last join and the next table in the FROM clause until all tables are processed.
4. **WHERE:** The WHERE filter is applied to VT3. Only rows for which the <*where_condition*> is TRUE are inserted to VT4.
5. **GROUP BY:** The rows from VT4 are arranged in groups based on the column list specified in the GROUP BY clause. VT5 is generated.
6. **CUBE | ROLLUP:** Supergroups (groups of groups) are added to the rows from VT5, generating VT6.
7. **HAVING:** The HAVING filter is applied to VT6. Only groups for which the <*having_condition*> is TRUE are inserted to VT7.
8. **SELECT:** The SELECT list is processed, generating VT8.
9. **DISTINCT:** Duplicate rows are removed from VT8. VT9 is generated.
10. **ORDER BY:** The rows from VT9 are sorted according to the column list specified in the ORDER BY clause. A cursor is generated (VC10).
11. **TOP:** The specified number or percentage of rows is selected from the beginning of VC10. Table VT11 is generated and returned to the caller.

Sample Query Based on Customers/Orders Scenario

To describe the logical processing phases in detail, I'll walk you through a sample query. First run the code in Listing 1-2 to create the Customers and Orders tables and populate them with sample data. Tables 1-1 and 1-2 show the contents of Customers and Orders.

Listing 1-2 Data definition language (DDL) and sample data for Customers and Orders

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
GO
IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
GO
CREATE TABLE dbo.Customers
(
  customerid  CHAR(5)     NOT NULL PRIMARY KEY,
  city        VARCHAR(10) NOT NULL
);

INSERT INTO dbo.Customers(customerid, city) VALUES('FISSA', 'Madrid');
INSERT INTO dbo.Customers(customerid, city) VALUES('FRNDO', 'Madrid');
INSERT INTO dbo.Customers(customerid, city) VALUES('KRLOS', 'Madrid');
INSERT INTO dbo.Customers(customerid, city) VALUES('MRPHS', 'Zion');

CREATE TABLE dbo.Orders
(
  orderid    INT         NOT NULL PRIMARY KEY,
  customerid CHAR(5)     NULL     REFERENCES Customers(customerid)
);

INSERT INTO dbo.Orders(orderid, customerid) VALUES(1, 'FRNDO');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(2, 'FRNDO');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(3, 'KRLOS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(4, 'KRLOS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(5, 'KRLOS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(6, 'MRPHS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(7, NULL);
```

Table 1-1 Contents of Customers Table

customerid	*city*
FISSA	Madrid
FRNDO	Madrid
KRLOS	Madrid
MRPHS	Zion

Table 1-2 Contents of Orders Table

orderid	*customerid*
1	FRNDO
2	FRNDO
3	KRLOS

Table 1-2 Contents of Orders Table

orderid	*customerid*
4	KRLOS
5	KRLOS
6	MRPHS
7	NULL

I will use the query shown in Listing 1-3 as my example. The query returns customers from Madrid that made fewer than three orders (including zero orders), along with their order counts. The result is sorted by order count, from smallest to largest. The output of this query is shown in Table 1-3.

Listing 1-3 Query: Madrid customers with fewer than three orders

```
SELECT C.customerid, COUNT(O.orderid) AS numorders
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.customerid = O.customerid
WHERE C.city = 'Madrid'
GROUP BY C.customerid
HAVING COUNT(O.orderid) < 3
ORDER BY numorders;
```

Table 1-3 Output: Madrid Customers with Fewer than Three Orders

customerid	*numorders*
FISSA	0
FRNDO	2

Both FISSA and FRNDO are customers from Madrid who made fewer than three orders. Examine the query, and try to read it while following the steps and phases described in Listing 1-1 and the section "Brief Description of Logical Query Processing Phases." If this is the first time you're thinking of a query in such terms, it's probably confusing for you. The following section should help you understand the nitty-gritty details.

Logical Query Processing Phase Details

This section describes the logical query processing phases in detail by applying them to the given sample query.

Step 1: Performing a Cartesian Product (Cross Join)

A Cartesian product (a cross join, or an unrestricted join) is performed between the first two tables that appear in the FROM clause, and as a result, virtual table VT1 is generated. VT1 contains one row for every possible combination of a row from the left table and a row from the

right table. If the left table contains *n* rows and the right table contains *m* rows, VT1 will contain *n* × *m* rows. The columns in VT1 are qualified (prefixed) with their source table names (or table aliases, if you specified ones in the query). In the subsequent steps (step 2 and on), a reference to a column name that is ambiguous (appears in more than one input table) must be table-qualified (for example, *C.customerid*). Specifying the table qualifier for column names that appear in only one of the inputs is optional (for example, *O.orderid* or just *orderid*).

Apply step 1 to the sample query (shown in Listing 1-3):

```
FROM Customers AS C ... JOIN Orders AS O
```

As a result, you get the virtual table VT1 shown in Table 1-4 with 28 rows (4×7).

Table 1-4 Virtual Table VT1 Returned from Step 1

C.customerid	*C.city*	*O.orderid*	*O.customerid*
FISSA	Madrid	1	FRNDO
FISSA	Madrid	2	FRNDO
FISSA	Madrid	3	KRLOS
FISSA	Madrid	4	KRLOS
FISSA	Madrid	5	KRLOS
FISSA	Madrid	6	MRPHS
FISSA	Madrid	7	NULL
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
FRNDO	Madrid	3	KRLOS
FRNDO	Madrid	4	KRLOS
FRNDO	Madrid	5	KRLOS
FRNDO	Madrid	6	MRPHS
FRNDO	Madrid	7	NULL
KRLOS	Madrid	1	FRNDO
KRLOS	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
KRLOS	Madrid	6	MRPHS
KRLOS	Madrid	7	NULL
MRPHS	Zion	1	FRNDO
MRPHS	Zion	2	FRNDO
MRPHS	Zion	3	KRLOS
MRPHS	Zion	4	KRLOS
MRPHS	Zion	5	KRLOS
MRPHS	Zion	6	MRPHS
MRPHS	Zion	7	NULL

Step 2: Applying the ON Filter (Join Condition)

The ON filter is the first of three possible filters (ON, WHERE, and HAVING) that can be specified in a query. The logical expression in the ON filter is applied to all rows in the virtual table returned by the previous step (VT1). Only rows for which the <*join_condition*> is TRUE become part of the virtual table returned by this step (VT2).

Three-Valued Logic

Allow me to digress a bit to cover important aspects of SQL related to this step. The possible values of a logical expression in SQL are TRUE, FALSE, and UNKNOWN. This is referred to as three-valued logic. Three-valued logic is unique to SQL. Logical expressions in most programming languages can be only TRUE or FALSE. The UNKNOWN logical value in SQL typically occurs in a logical expression that involves a NULL (for example, the logical value of each of these three expressions is UNKNOWN: *NULL* > *42*; *NULL* = *NULL*; *X* + *NULL* > *Y*). The special value NULL typically represents a missing or irrelevant value. When comparing a missing value to another value (even another NULL), the logical result is always UNKNOWN.

Dealing with UNKNOWN logical results and NULLs can be very confusing. While NOT TRUE is FALSE, and NOT FALSE is TRUE, the opposite of UNKNOWN (NOT UNKNOWN) is still UNKNOWN.

UNKNOWN logical results and NULLs are treated inconsistently in different elements of the language. For example, all query filters (ON, WHERE, and HAVING) treat UNKNOWN in the same way as FALSE. A row for which a filter is UNKNOWN is eliminated from the result set. On the other hand, an UNKNOWN value in a CHECK constraint is actually treated like TRUE. Suppose you have a CHECK constraint in a table to require that the salary column be greater than zero. A row entered into the table with a NULL salary is accepted, because (*NULL* > *0*) is UNKNOWN and treated like TRUE in the CHECK constraint.

A comparison between two NULLs in filters yields an UNKNOWN, which as I mentioned earlier, is treated like FALSE—as if one NULL is different than another.

On the other hand, UNIQUE and PRIMARY KEY constraints, sorting, and grouping treat NULLs as equal:

- You cannot insert into a table two rows with a NULL in a column that has a UNIQUE or PRIMARY KEY constraint defined on it.
- A GROUP BY clause groups all NULLs into one group.
- An ORDER BY clause sorts all NULLs together.

In short, it's a good idea to be aware of the way UNKNOWN logical results and NULLs are treated in the different elements of the language to spare you grief.

Apply step 2 to the sample query:

```
ON C.customerid = O.customerid
```

Table 1-5 shows the value of the logical expression in the ON filter for the rows from VT1.

Table 1-5 Logical Results of ON Filter Applied to Rows from VT1

Match?	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
FALSE	FISSA	Madrid	1	FRNDO
FALSE	FISSA	Madrid	2	FRNDO
FALSE	FISSA	Madrid	3	KRLOS
FALSE	FISSA	Madrid	4	KRLOS
FALSE	FISSA	Madrid	5	KRLOS
FALSE	FISSA	Madrid	6	MRPHS
UNKNOWN	FISSA	Madrid	7	NULL
TRUE	FRNDO	Madrid	1	FRNDO
TRUE	FRNDO	Madrid	2	FRNDO
FALSE	FRNDO	Madrid	3	KRLOS
FALSE	FRNDO	Madrid	4	KRLOS
FALSE	FRNDO	Madrid	5	KRLOS
FALSE	FRNDO	Madrid	6	MRPHS
UNKNOWN	FRNDO	Madrid	7	NULL
FALSE	KRLOS	Madrid	1	FRNDO
FALSE	KRLOS	Madrid	2	FRNDO
TRUE	KRLOS	Madrid	3	KRLOS
TRUE	KRLOS	Madrid	4	KRLOS
TRUE	KRLOS	Madrid	5	KRLOS
FALSE	KRLOS	Madrid	6	MRPHS
UNKNOWN	KRLOS	Madrid	7	NULL
FALSE	MRPHS	Zion	1	FRNDO
FALSE	MRPHS	Zion	2	FRNDO
FALSE	MRPHS	Zion	3	KRLOS
FALSE	MRPHS	Zion	4	KRLOS
FALSE	MRPHS	Zion	5	KRLOS
TRUE	MRPHS	Zion	6	MRPHS
UNKNOWN	MRPHS	Zion	7	NULL

Only rows for which the <*join_condition*> is TRUE are inserted to VT2—the input virtual table of the next step, shown in Table 1-6.

Table 1-6 Virtual Table VT2 Returned from Step 2

Match?	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
TRUE	FRNDO	Madrid	1	FRNDO
TRUE	FRNDO	Madrid	2	FRNDO
TRUE	KRLOS	Madrid	3	KRLOS
TRUE	KRLOS	Madrid	4	KRLOS
TRUE	KRLOS	Madrid	5	KRLOS
TRUE	MRPHS	Zion	6	MRPHS

Step 3: Adding Outer Rows

This step is relevant only for an outer join. For an outer join, you mark one or both input tables as *preserved* by specifying the type of outer join (LEFT, RIGHT, or FULL). Marking a table as preserved means that you want all of its rows returned, even when filtered out by the <*join_condition*>. A left outer join marks the left table as preserved, a right outer join marks the right, and a full outer join marks both. Step 3 returns the rows from VT2, plus rows from the preserved table for which a match was not found in step 2. These added rows are referred to as *outer rows*. NULLs are assigned to the attributes (column values) of the nonpreserved table in the outer rows. As a result, virtual table VT3 is generated.

In our example, the preserved table is Customers:

```
Customers AS C LEFT OUTER JOIN Orders AS O
```

Only customer FISSA did not find any matching orders (wasn't part of VT2). Therefore, FISSA is added to the rows from the previous step with NULLs for the Orders attributes, and as a result, virtual table VT3 (shown in Table 1-7) is generated.

Table 1-7 Virtual Table VT3 Returned from Step 3

C.customerid	*C.city*	*O.orderid*	*O.customerid*
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
MRPHS	Zion	6	MRPHS
FISSA	Madrid	NULL	NULL

Note If more than two tables are joined, steps 1 through 3 will be applied between VT3 and the third table in the FROM clause. This process will continue repeatedly if more tables appear in the FROM clause, and the final virtual table will be used as the input for the next step.

Step 4: Applying the WHERE Filter

The WHERE filter is applied to all rows in the virtual table returned by the previous step. Only rows for which <*where_condition*> is TRUE become part of the virtual table returned by this step (VT4).

Caution Because the data is not grouped yet, you cannot use aggregate filters here—for example, you cannot write *WHERE orderdate = MAX(orderdate)*. Also, you cannot refer to column aliases created by the SELECT list because the SELECT list was not processed yet—for example, you cannot write *SELECT YEAR(orderdate) AS orderyear ... WHERE orderyear > 2000*.

A confusing aspect of queries containing an OUTER JOIN clause is whether to specify a logical expression in the ON filter or in the WHERE filter. The main difference between the two is that ON is applied before adding outer rows (step 3), while WHERE is applied after step 3. An elimination of a row from the preserved table by the ON filter is not final because step 3 will add it back; while an elimination of a row by the WHERE filter is final. Bearing this in mind should help you make the right choice.

For example, suppose you want to return certain customers and their orders from the Customers and Orders tables. The customers you want to return are only Madrid customers, both those that made orders and those that did not. An outer join is designed exactly for such a request. You perform a left outer join between Customers and Orders, marking the Customers table as the preserved table. To be able to return customers that made no orders, you must specify the correlation between customers and orders in the ON clause (*ON C.customerid = O.customerid*). Customers with no orders are eliminated in step 2 but added back in step 3 as outer rows. However, because you want to keep only rows for Madrid customers, regardless of whether they made orders, you must specify the city filter in the WHERE clause (*WHERE C.city = 'Madrid'*). Specifying the city filter in the ON clause would cause non-Madrid customers to be added back to the result set by step 3.

Tip There's a logical difference between the ON and WHERE clauses only when using an outer join. When using an inner join, it doesn't matter where you specify your logical expressions because step 3 is skipped. The filters are applied one after the other with no intermediate step between them.

There's one exception that is relevant only when using the GROUP BY ALL option. I will discuss this option shortly in the next section, which covers the GROUP BY phase.

Apply the filter in the sample query:

```
WHERE C.city = 'Madrid'
```

The row for customer MRPHS from VT3 is removed because the city is not Madrid, and virtual table VT4, which is shown in Table 1-8, is generated.

Table 1-8 Virtual Table VT4 Returned from Step 4

C.customerid	*C.city*	*O.orderid*	*O.customerid*
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
FISSA	Madrid	NULL	NULL

Step 5: Grouping

The rows from the table returned by the previous step are arranged in groups. Each unique combination of values in the column list that appears in the GROUP BY clause makes a group. Each base row from the previous step is attached to one and only one group. Virtual table VT5 is generated. VT5 consists of two sections: the *groups* section that is made of the actual groups, and the *raw* section that is made of the attached base rows from the previous step.

Apply step 5 to the sample query:

```
GROUP BY C.customerid
```

You get the virtual table VT5 shown in Table 1-9.

Table 1-9 Virtual Table VT5 Returned from Step 5

Groups	**Raw**			
C.customerid	***C.customerid***	***C.city***	***O.orderid***	***O.customerid***
FRNDO	FRNDO	Madrid	1	FRNDO
	FRNDO	Madrid	2	FRNDO
KRLOS	KRLOS	Madrid	3	KRLOS
	KRLOS	Madrid	4	KRLOS
	KRLOS	Madrid	5	KRLOS
FISSA	FISSA	Madrid	NULL	NULL

If a GROUP BY clause is specified in a query, all following steps (HAVING, SELECT, and so on) can specify only expressions that result in a scalar (singular) value for a group. In other words, the results can be either a column/expression that participates in the GROUP BY list—for example, *C.customerid*—or an aggregate function, such as *COUNT(O.orderid)*. The reasoning behind this limitation is that a single row in the final result set will eventually be generated for each group (unless filtered out). Examine VT5 in Table 1-9, and think what the query should return for customer FRNDO if the SELECT list you specified had been *SELECT C.customerid, O.orderid*. There are two different *orderid* values in the group; therefore, the answer is nondeterministic. SQL doesn't allow such a request. On the other hand, if you specify: *SELECT C.customerid, COUNT(O.orderid) AS numorders*, the answer for FRNDO is deterministic: it's 2.

Note You're also allowed to group by the result of an expression—for instance, *GROUP BY YEAR(orderdate)*. If you do, when working in SQL Server 2000, all following steps cannot perform any further manipulation to the GROUP BY expression, unless it's a base column. For example, the following is not allowed in SQL Server 2000: *SELECT YEAR(orderdate) + 1 AS nextyear ... GROUP BY YEAR(orderdate)*. In SQL Server 2005, this limitation has been removed.

This phase considers NULLs as equal. That is, all NULLs are grouped into one group just like a known value.

As I mentioned earlier, the input to the GROUP BY phase is the virtual table returned by the previous step (VT4). If you specify GROUP BY ALL, groups that were removed by the fourth phase (WHERE filter) are added to this step's result virtual table (VT5) with an empty set in the raw section. This is the only case where there is a difference between specifying a logical expression in the ON clause and in the WHERE clause when using an inner join. If you revise our example to use the *GROUP BY ALL C.customerid* instead of *GROUP BY C.customerid*, you'll find that customer *MRPHS*, which was removed by the WHERE filter, will be added to VT5's groups section, along with an empty set in the raw section. The COUNT aggregate function in one of the following steps would be zero for such a group, while all other aggregate functions (SUM, AVG, MIN, MAX) would be NULL.

Note The GROUP BY ALL option is a nonstandard legacy feature. It introduces many semantic issues when Microsoft adds new T-SQL features. Even though this feature is fully supported in SQL Server 2005, you might want to refrain from using it because it might eventually be deprecated.

Step 6: Applying the CUBE or ROLLUP Option

If CUBE or ROLLUP is specified, supergroups are created and added to the groups in the virtual table returned by the previous step. Virtual table VT6 is generated.

Step 6 is skipped in our example because CUBE and ROLLUP are not specified in the sample query. CUBE and ROLLUP will be covered in Chapter 6.

Step 7: Applying the HAVING Filter

The HAVING filter is applied to the groups in the table returned by the previous step. Only groups for which the <*having_condition*> is TRUE become part of the virtual table returned by this step (VT7). The HAVING filter is the first and only filter that applies to the grouped data.

Apply this step to the sample query:

```
HAVING COUNT(O.orderid) < 3
```

The group for KRLOS is removed because it contains three orders. Virtual table VT7, which is shown in Table 1-10, is generated.

Table 1-10 Virtual Table VT7 Returned from Step 7

C.customerid	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
FRNDO	FRNDO	Madrid	1	FRNDO
	FRNDO	Madrid	2	FRNDO
FISSA	FISSA	Madrid	NULL	NULL

Note It is important to specify *COUNT(O.orderid)* here and not *COUNT(*)*. Because the join is an outer one, outer rows were added for customers with no orders. *COUNT(*)* would have added outer rows to the count, undesirably producing a count of one order for FISSA. *COUNT(O.orderid)* correctly counts the number of orders for each customer, producing the desired value 0 for FISSA. Remember that *COUNT(*<expression>*)* ignores NULLs just like any other aggregate function.

An aggregate function does not accept a subquery as an input—for example, *HAVING SUM((SELECT ...)) > 10*.

Step 8: Processing the SELECT List

Though specified first in the query, the SELECT list is processed only at the eighth step. The SELECT phase constructs the table that will eventually be returned to the caller. The expressions in the SELECT list can return base columns and manipulations of base columns from the virtual table returned by the previous step. Remember that if the query is an aggregate query, after step 5 you can refer to base columns from the previous step only if they are part of the groups section (GROUP BY list). If you refer to columns from the raw section, these must be aggregated. Base columns selected from the previous step maintain their column names unless you alias them (for example, *col1 AS c1*). Expressions that are not base columns should be aliased to have a column name in the result table—for example, *YEAR(orderdate) AS orderyear*.

Important Aliases created by the SELECT list cannot be used by earlier steps. In fact, expression aliases cannot even be used by other expressions within the same SELECT list. The reasoning behind this limitation is another unique aspect of SQL, being an all-at-once operation. For example, in the following SELECT list, the logical order in which the expressions are evaluated should not matter and is not guaranteed: *SELECT c1 + 1 AS e1, c2 + 1 AS e2*. Therefore, the following SELECT list is not supported: *SELECT c1 + 1 AS e1, e1 + 1 AS e2*. You're allowed to reuse column aliases only in steps following the SELECT list, such as the ORDER BY step—for example, *SELECT YEAR(orderdate) AS orderyear ... ORDER BY orderyear*.

Apply this step to the sample query:

```
SELECT C.customerid, COUNT(O.orderid) AS numorders
```

You get the virtual table VT8, which is shown in Table 1-11.

Table 1-11 Virtual Table VT8 Returned from Step 8

C.customerid	*numorders*
FRNDO	2
FISSA	0

The concept of an all-at-once operation can be hard to grasp. For example, in most programming environments, to swap values between variables you use a temporary variable. However, to swap table column values in SQL, you can use:

```
UPDATE dbo.T1 SET c1 = c2, c2 = c1;
```

Logically, you should assume that the whole operation takes place at once. It is as if the table is not modified until the whole operation finishes and then the result replaces the source. For similar reasons, this UPDATE

```
UPDATE dbo.T1 SET c1 = c1 + (SELECT MAX(c1) FROM dbo.T1);
```

would update all of T1's rows, adding to c1 the maximum c1 value from T1 when the update started. You shouldn't be concerned that the maximum c1 value would keep changing as the operation proceeds because the operation occurs all at once.

Step 9: Applying the DISTINCT Clause

If a DISTINCT clause is specified in the query, duplicate rows are removed from the virtual table returned by the previous step, and virtual table VT9 is generated.

Step 9 is skipped in our example because DISTINCT is not specified in the sample query. In fact, DISTINCT is redundant when GROUP BY is used, and it would remove no rows.

Step 10: Applying the ORDER BY Clause

The rows from the previous step are sorted according to the column list specified in the ORDER BY clause returning the cursor VC10. This step is the first and only step where column aliases created in the SELECT list can be reused.

According to both ANSI SQL:1992 and ANSI SQL:1999, if DISTINCT is specified, the expressions in the ORDER BY clause have access only to the virtual table returned by the previous step (VT9). That is, you can sort by only what you select. ANSI SQL:1992 has the same limitation even when DISTINCT is not specified. However, ANSI SQL:1999 enhances the ORDER BY support by allowing access to both the input and output virtual tables of the SELECT phase. That is, if DISTINCT is not specified, in the ORDER BY clause you can specify any expression that would have been allowed in the SELECT clause. Namely, you can sort by expressions that you don't end up returning in the final result set.

There is a reason for not allowing access to expressions you're not returning if DISTINCT is specified. When adding expressions to the SELECT list, DISTINCT can potentially change the number of rows returned. Without DISTINCT, of course, changes in the SELECT list don't affect the number of rows returned. T-SQL always implemented the ANSI SQL:1999 approach.

In our example, because DISTINCT is not specified, the ORDER BY clause has access to both VT7, shown in Table 1-10, and VT8, shown in Table 1-11.

In the ORDER BY clause, you can also specify ordinal positions of result columns from the SELECT list. For example, the following query sorts the orders first by customerid, and then by *orderid*:

```
SELECT orderid, customerid FROM dbo.Orders ORDER BY 2, 1;
```

However, this practice is not recommended because you might make changes to the SELECT list and forget to revise the ORDER BY list accordingly. Also, when the query strings are long, it's hard to figure out which item in the ORDER BY list corresponds to which item in the SELECT list.

Important This step is different than all other steps in the sense that it doesn't return a valid table; instead, it returns a cursor. Remember that SQL is based on set theory. A set doesn't have a predetermined order to its rows; it's a logical collection of members, and the order of the members shouldn't matter. A query that applies sorting to the rows of a table returns an object with rows organized in a particular physical order. ANSI calls such an object a *cursor*. Understanding this step is one of the most fundamental things in correctly understanding SQL.

Usually when describing the contents of a table, most people (including me) routinely depict the rows in a certain order. For example, I provided Tables 1-1 and 1-2 to describe the contents of the Customers and Orders tables. In depicting the rows one after the other, unintentionally I help cause some confusion by implying a certain order. A more correct way to depict the content of the Customers and Orders tables would be the one shown in Figure 1-1.

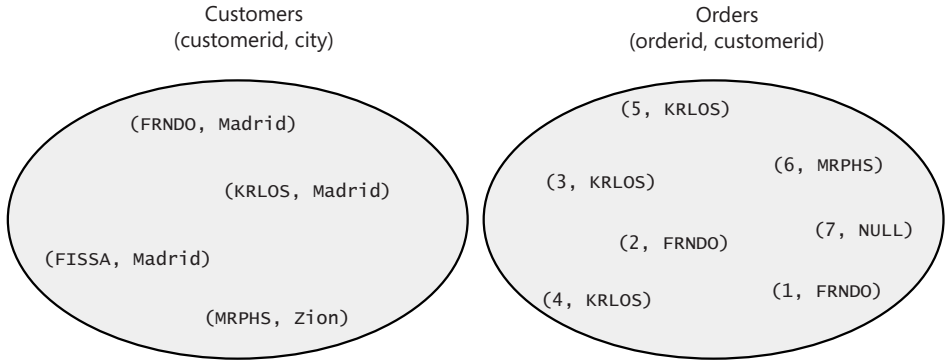

Figure 1-1 Customers and Orders sets

Note Although SQL doesn't assume any given order to a table's *rows*, it does maintain ordinal positions for *columns* based on creation order. Specifying *SELECT ** (although a bad practice for several reasons that I'll describe later in the book) guarantees the columns would be returned in creation order.

Because this step doesn't return a table (it returns a cursor), a query with an ORDER BY clause cannot be used as a table expression—that is, a view, inline table-valued function, subquery, derived table, or common table expression (CTE). Rather, the result must be returned to the client application that expects a physical record set back. For example, the following derived table query is invalid and produces an error:

```
SELECT *
FROM (SELECT orderid, customerid
      FROM dbo.Orders
      ORDER BY orderid) AS D;
```

Similarly, the following view is invalid:

```
CREATE VIEW dbo.VSortedOrders
AS

SELECT orderid, customerid
FROM dbo.Orders
ORDER BY orderid
GO
```

In SQL, no query with an ORDER BY clause is allowed in a table expression. In T-SQL, there is an exception to this rule that is described in the following step—applying the TOP option.

So remember, don't assume any particular order for a table's rows. Conversely, don't specify an ORDER BY clause unless you really need the rows sorted. Sorting has a cost—SQL Server needs to perform an ordered index scan or apply a sort operator.

The ORDER BY step considers NULLs as equal. That is, NULLs are sorted together. ANSI leaves the question of whether NULLs are sorted lower or higher than known values up to implementations, which must be consistent. T-SQL sorts NULLs as lower than known values (first).

Apply this step to the sample query:

```
ORDER BY numorders
```

You get the cursor VC10 shown in Table 1-12.

Table 1-12 Cursor VC10 Returned from Step 10

C.customerid	*numorders*
FISSA	0
FRNDO	2

Step 11: Applying the TOP Option

The TOP option allows you to specify a number or percentage of rows (rounded up) to return. In SQL Server 2000, the input to TOP must be a constant, while in SQL Server 2005, the input can be any self-contained expression. The specified number of rows is selected from the beginning of the cursor returned by the previous step. Table VT11 is generated and returned to the caller.

Note The TOP option is T-SQL specific and is not relational.

This step relies on the physical order of the rows to determine which rows are considered the "first" requested number of rows. If an ORDER BY clause with a unique ORDER BY list is specified in a query, the result is deterministic. That is, there's only one possible correct result, containing the first requested number of rows based on the specified sort. Similarly, when an ORDER BY clause is specified with a non-unique ORDER BY list but the TOP option is specified WITH TIES, the result is also deterministic. SQL Server inspects the last row that was returned physically and returns all other rows from the table that have the same sort values as the last row.

However, when a non-unique ORDER BY list is specified without the WITH TIES option, or ORDER BY is not specified at all, a TOP query is nondeterministic. That is, the rows returned are the ones that SQL Server happened to physically access first, and there might be different results that are considered correct. If you want to guarantee determinism, a TOP query must have either a unique ORDER BY list or the WITH TIES option.

As you can surmise, TOP queries are most commonly used with an ORDER BY clause that determines which rows to return. SQL Server allows you to specify TOP queries in table expressions. It wouldn't make much sense to allow TOP queries in table expressions without allowing you to also specify an ORDER BY clause. (See the limitation in step 10.) Thus, queries with an ORDER BY clause are in fact allowed in table expressions only if TOP is also specified. In other words, a query with both a TOP clause and an ORDER BY clause returns a relational result. The ironic thing is that by using the nonstandard, nonrelational TOP option, a query that would otherwise return a cursor returns a relational result. Support for nonstandard, nonrelational features (as practical as they might be) allows programmers to exploit them in some absurd ways that would not have been supported otherwise. Here's an example:

```
SELECT *
FROM (SELECT TOP 100 PERCENT orderid, customerid
      FROM dbo.Orders
      ORDER BY orderid) AS D;
```

Or:

```
CREATE VIEW dbo.VSortedOrders
AS

SELECT TOP 100 PERCENT orderid, customerid
FROM dbo.Orders
ORDER BY orderid
GO
```

Step 11 is skipped in our example because TOP is not specified.

New Logical Processing Phases in SQL Server 2005

This section covers the logical processing phases involved with the new T-SQL query elements in SQL Server 2005. These include new table operators (APPLY, PIVOT, and UNPIVOT), the new OVER clause, and new set operations (EXCEPT and INTERSECT).

Note APPLY, PIVOT, and UNPIVOT are not ANSI operators; rather, they are T-SQL specific extensions.

I find it a bit problematic to cover the logical processing phases involved with the new product version in detail in the first chapter. These elements are completely new, and there's so much to say about each. Instead, I will provide a brief overview of each element here and conduct much more detailed discussions later in the book in focused chapters.

As I mentioned earlier, my goal for this chapter is to give you a reference that you can return to later when in doubt regarding the logical aspects of query elements and the way they interact with each other. Bearing this in mind, the full meaning of the logical phases of query processing that handle the new elements might not be completely clear to you right now. Don't let that worry you. After reading the focused chapters discussing each element in detail, you will probably find the reference I provide in this chapter useful. Rest assured that everything will make more sense then.

Table Operators

SQL Server 2005 supports four types of table operators in the FROM clause of a query: JOIN, APPLY, PIVOT, and UNPIVOT.

I covered the logical processing phases involved with joins earlier and will also discuss joins in more details in Chapter 5. Here I will briefly describe the three new operators and how they interact with each other.

Table operators get one or two tables as inputs. Call them *left input* and *right input* based on their position in respect to the table operator keyword (JOIN, APPLY, PIVOT, UNPIVOT). Just like joins, all table operators get a virtual table as their left input. The first table operator that appears in the FROM clause gets a table expression as the left input and returns a virtual table

as a result. A table expression can stand for many things: a real table, temporary table, table variable, derived table, CTE, view, or table-valued function.

More Info For details on table expressions, please refer to Chapter 4.

The second table operator that appears in the FROM clause gets the virtual table returned from the previous table operation as its left input.

Each table operator involves a different set of steps. For convenience and clarity, I'll prefix the step numbers with the initial of the table operator (J for JOIN, A for APPLY, P for PIVOT, and U for UNPIVOT).

Following are the four table operators along with their elements:

```
(J) <left_table_expression>
      <join_type> JOIN <right_table_expression>
       ON <join_condition>

(A) <left_table_expression>
      {CROSS | OUTER} APPLY <table_expression>

(P) <left_table_expression>
      PIVOT (<aggregate_func(<expression>)> FOR
        <source_col> IN(<target_col_list>))
        AS <result_table_alias>

(U) <left_table_expression>
      UNPIVOT (<target_values_col> FOR
        <target_names_col> IN(<source_col_list>))
        AS <result_table_alias>
```

As a reminder, a join involves a subset (depending on the join type) of the following steps:

1. J1: Cross Left and Right Inputs
2. J2: Apply ON Clause
3. J3: Add Outer Rows

APPLY

The APPLY operator involves a subset (depending on the apply type) of the following two steps:

1. A1: Apply Right Table Expression to Left Table Input's Rows
2. A2: Add Outer Rows

The APPLY operator basically applies the right table expression to every row from the left input. You can think of it as being similar to a join, with one important difference—the right table expression can refer to the left input's columns as correlations. It's as though in a join there's no precedence between the two inputs when evaluating them. With APPLY, it's as

though the left input is evaluated first, and then the right input is evaluated once for each row from the left.

Step A1 is always applied in both CROSS APPLY and OUTER APPLY. Step A2 is applied only for OUTER APPLY. CROSS APPLY doesn't return an outer (left) row if the inner (right) table expression returns an empty set for it. OUTER APPLY will return such a row, with NULLs in the inner table expression's attributes.

For example, the following query returns the two most recent orders (assuming for the sake of this example that *orderid* represents chronological order) for each customer, generating the output shown in Table 1-13:

```
SELECT C.customerid, city, orderid
FROM dbo.Customers AS C
  CROSS APPLY
    (SELECT TOP(2) orderid, customerid
     FROM dbo.Orders AS O
     WHERE O.customerid = C.customerid
     ORDER BY orderid DESC) AS CA;
```

Table 1-13 Two Most Recent Orders for Each Customer

customerid	*city*	*orderid*
FRNDO	Madrid	2
FRNDO	Madrid	1
KRLOS	Madrid	5
KRLOS	Madrid	4
MRPHS	Zion	6

Notice that FISSA is missing from the output because the table expression CA returned an empty set for it. If you also want to return customers that made no orders, use OUTER APPLY as follows, generating the output shown in Table 1-14:

```
SELECT C.customerid, city, orderid
FROM dbo.Customers AS C
  OUTER APPLY
    (SELECT TOP(2) orderid, customerid
     FROM dbo.Orders AS O
     WHERE O.customerid = C.customerid
     ORDER BY orderid DESC) AS OA;
```

Table 1-14 Two Most Recent Orders for Each Customer, Including Customers that Made No Orders

customerid	*city*	*orderid*
FISSA	Madrid	NULL
FRNDO	Madrid	2
FRNDO	Madrid	1

Table 1-14 Two Most Recent Orders for Each Customer, Including Customers that Made No Orders

customerid	*city*	*orderid*
KRLOS	Madrid	5
KRLOS	Madrid	4
MRPHS	Zion	6

More Info For more details on the APPLY operator, please refer to Chapter 7.

PIVOT

The PIVOT operator essentially allows you to rotate, or pivot, data from a state of groups of multiple rows to a state of multiple columns in a single row per group, performing aggregations along the way.

Before I explain and demonstrate the logical steps involved with using the PIVOT operator, examine the following query, which I will later use as the left input to the PIVOT operator:

```
SELECT C.customerid, city,
  CASE
    WHEN COUNT(orderid)  = 0 THEN 'no_orders'
    WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
    WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
  END AS category
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.customerid = O.customerid
GROUP BY C.customerid, city;
```

This query returns customer categories based on count of orders (no orders, up to two orders, more than two orders), yielding the result set shown in Table 1-15.

Table 1-15 Customer Categories Based on Count of Orders

customerid	*city*	*category*
FISSA	Madrid	*no_orders*
FRNDO	Madrid	*upto_two_orders*
KRLOS	Madrid	*more_than_two_orders*
MRPHS	Zion	*upto_two_orders*

Suppose you wanted to know the number of customers that fall into each category per city. The following PIVOT query allows you to achieve this, generating the output shown in Table 1-16:

```
SELECT city, no_orders, upto_two_orders, more_than_two_orders
FROM (SELECT C.customerid, city,
        CASE
          WHEN COUNT(orderid)  = 0 THEN 'no_orders'
```

```
        WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
        WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
      END AS category
    FROM dbo.Customers AS C
      LEFT OUTER JOIN dbo.Orders AS O
        ON C.customerid = O.customerid
    GROUP BY C.customerid, city) AS D
PIVOT(COUNT(customerid) FOR
  category IN([no_orders],
              [upto_two_orders],
              [more_than_two_orders])) AS P;
```

Table 1-16 Number of Customers that Fall into Each Category per City

city	*no_orders*	*upto_two_orders*	*more_than_two_orders*
Madrid	1	1	1
Zion	0	1	0

Don't get distracted by the query that generates the derived table D. As far as you're concerned, the PIVOT operator gets a table expression called D, containing the customer categories as its left input.

The PIVOT operator involves the following three logical phases:

1. P1: Implicit Grouping
2. P2: Isolating Values
3. P3: Applying the Aggregate Function

The first phase (P1) is very tricky to grasp. You can see in the query that the PIVOT operator refers to two of the columns from D as input arguments (*customerid* and *category*). The first phase implicitly groups the rows from D based on all columns that weren't mentioned in PIVOT's inputs, as though there were a hidden GROUP BY there. In our case, only the city column wasn't mentioned anywhere in PIVOT's input arguments. So you get a group for each city (Madrid and Zion, in our case).

Note PIVOT's implicit grouping phase doesn't substitute an explicit GROUP BY clause, should one appear in a query. PIVOT will eventually yield a result virtual table, which in turn will be input to the next logical phase, be it another table operation or the WHERE phase. And as I described earlier in the chapter, following the WHERE phase, there might be a GROUP BY phase. So when both PIVOT and GROUP BY appear in a query, you get two separate grouping phases—one as the first phase of PIVOT (P1), and a later one as the query's GROUP BY phase.

PIVOT's second phase (P2) isolates values corresponding to target columns. Logically, it uses the following CASE expression for each target column specified in the IN clause:

```
CASE WHEN <source_col> = <target_col_element> THEN <expression> END
```

In this situation, the following three expressions are logically applied:

```
CASE WHEN category = 'no_orders'            THEN customerid END,
CASE WHEN category = 'upto_two_orders'      THEN customerid END,
CASE WHEN category = 'more_than_two_orders' THEN customerid END
```

Note A CASE expression with no ELSE clause has an implicit ELSE NULL.

For each target column, the CASE expression will return the customer ID only if the source row had the corresponding category; otherwise, CASE will return a NULL.

PIVOT's third phase (P3) applies the specified aggregate function on top of each CASE expression, generating the result columns. In our case, the expressions logically become the following:

```
COUNT(CASE WHEN category = 'no_orders'
            THEN customerid END) AS [no_orders],
COUNT(CASE WHEN category = 'upto_two_orders'
            THEN customerid END) AS [upto_two_orders],
COUNT(CASE WHEN category = 'more_than_two_orders'
            THEN customerid END) AS [more_than_two_orders]
```

In summary, the previous PIVOT query is logically equivalent to the following query:

```
SELECT city,
  COUNT(CASE WHEN category = 'no_orders'
              THEN customerid END) AS [no_orders],
  COUNT(CASE WHEN category = 'upto_two_orders'
              THEN customerid END) AS [upto_two_orders],
  COUNT(CASE WHEN category = 'more_than_two_orders'
              THEN customerid END) AS [more_than_two_orders]
FROM (SELECT C.customerid, city,
        CASE
          WHEN COUNT(orderid)  = 0 THEN 'no_orders'
          WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
          WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
        END AS category
      FROM dbo.Customers AS C
        LEFT OUTER JOIN dbo.Orders AS O
          ON C.customerid = O.customerid
      GROUP BY C.customerid, city) AS D
GROUP BY city;
```

More Info For more details on the PIVOT operator, please refer to Chapter 6.

UNPIVOT

UNPIVOT is the inverse of PIVOT, rotating data from a state of multiple column values from the same row to multiple rows, each with a different source column value.

Before I demonstrate UNPIVOT's logical phases, first run the code in Listing 1-4, which creates and populates the PivotedCategories table.

Listing 1-4 Creating and populating the PivotedCategories table

```
SELECT city, no_orders, upto_two_orders, more_than_two_orders
INTO dbo.PivotedCategories
FROM (SELECT C.customerid, city,
        CASE
          WHEN COUNT(orderid)  = 0 THEN 'no_orders'
          WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
          WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
        END AS category
      FROM dbo.Customers AS C
        LEFT OUTER JOIN dbo.Orders AS O
          ON C.customerid = O.customerid
      GROUP BY C.customerid, city) AS D
  PIVOT(COUNT(customerid) FOR
    category IN([no_orders],
                [upto_two_orders],
                [more_than_two_orders])) AS P;

UPDATE dbo.PivotedCategories
  SET no_orders = NULL, upto_two_orders = 3
WHERE city = 'Madrid';
```

After you run the code in Listing 1-4, the PivotedCategories table will contain the data shown in Table 1-17.

Table 1-17 Contents of PivotedCategories Table

city	*no_orders*	*upto_two_orders*	*more_than_two_orders*
Madrid	NULL	3	1
Zion	0	1	0

I will use the following query as an example to describe the logical processing phases involved with the UNPIVOT operator:

```
SELECT city, category, num_custs
FROM dbo.PivotedCategories
  UNPIVOT(num_custs FOR
    category IN([no_orders],
                [upto_two_orders],
                [more_than_two_orders])) AS U
```

This query unpivots (or splits) the customer categories from each source row to a separate row per category, generating the output shown in Table 1-18.

Table 1-18 Unpivoted Customer Categories

city	*category*	*num_custs*
Madrid	*upto_two_orders*	3
Madrid	*more_than_two_orders*	1
Zion	*no_orders*	0

Table 1-18 Unpivoted Customer Categories

city	category	num_custs
Zion	*upto_two_orders*	1
Zion	*more_than_two_orders*	0

The following three logical processing phases are involved in an UNPIVOT operation:

1. U1: Generating Duplicates
2. U2: Isolating Target Column Values
3. U3: Filtering Out Rows with NULLs

The first step (U1) duplicates rows from the left table expression provided to UNPIVOT as an input (PivotedCategories, in our case). Each row is duplicated once for each source column that appears in the IN clause. Because there are three column names in the IN clause, each source row will be duplicated three times. The result virtual table will contain a new column holding the source column names as character strings. The name of this column will be the one specified right before the IN clause (category, in our case). The virtual table returned from the first step in our example is shown in Table 1-19.

Table 1-19 Virtual Table Returned from UNPIVOT's First Step

city	no_orders	upto_two_orders	more_than_two_orders	category
Madrid	NULL	3	1	*no_orders*
Madrid	NULL	3	1	*upto_two_orders*
Madrid	NULL	3	1	*more_than_two_orders*
Zion	0	1	0	*no_orders*
Zion	0	1	0	*upto_two_orders*
Zion	0	1	0	*more_than_two_orders*

The second step (U2) isolates the target column values. The name of the target column that will hold the values is specified right before the FOR clause (*num_custs*, in our case). The target column name will contain the value from the column corresponding to the current row's category from the virtual table. The virtual table returned from this step in our example is shown in Table 1-20.

Table 1-20 Virtual Table Returned from UNPIVOT's Second Step

city	category	num_custs
Madrid	*no_orders*	NULL
Madrid	*upto_two_orders*	3
Madrid	*more_than_two_orders*	1
Zion	*no_orders*	0
Zion	*upto_two_orders*	1
Zion	*more_than_two_orders*	0

UNPIVOT's third and final step (U3) is to filter out rows with NULLs in the result value column (*num_custs*, in our case). The virtual table returned from this step in our example is shown in Table 1-21.

Table 1-21 Virtual Table Returned from UNPIVOT's Third Step

city	*category*	*num_custs*
Madrid	*upto_two_orders*	3
Madrid	*more_than_two_orders*	1
Zion	*no_orders*	0
Zion	*upto_two_orders*	1
Zion	*more_than_two_orders*	0

When you're done experimenting with the UNPIVOT operator, drop the PivotedCategories table:

```
DROP TABLE dbo.PivotedCategories;
```

More Info For more details on the UNPIVOT operator, please refer to Chapter 6.

OVER Clause

The OVER clause allows you to request window-based calculations. In SQL Server 2005, this clause is a new option for aggregate functions (both built-in and custom Common Language Runtime [CLR]-based aggregates) and it is a required element for the four new analytical ranking functions (ROW_NUMBER, RANK, DENSE_RANK, and NTILE). When an OVER clause is specified, its input, instead of the query's GROUP BY list, specifies the window of rows over which the aggregate or ranking function is calculated.

I won't discuss applications of windows-based calculations here, nor will I go into detail about exactly how these functions work; I'll only explain the phases in which the OVER clause is applicable. I'll cover the OVER clause in more detail in Chapters 4 and 6.

The OVER clause is applicable only in one of two phases: the SELECT phase (8) and the ORDER BY phase (10). This clause has access to whichever virtual table is provided to that phase as input. Listing 1-5 highlights the logical processing phases in which the OVER clause is applicable.

Listing 1-5 OVER clause in logical query processing

```
(8)  SELECT (9) DISTINCT (11) TOP <select_list>
(1)  FROM <left_table>
(3)    <join_type> JOIN <right_table>
(2)      ON <join_condition>
(4)  WHERE <where_condition>
```

```
(5)  GROUP BY <group_by_list>
(6)  WITH {CUBE | ROLLUP}
(7)  HAVING <having_condition>
(10) ORDER BY <order_by_list>
```

You specify the OVER clause following the function to which it applies in either the *select_list* or the *order_by_list*.

Even though I didn't really explain in detail how the OVER clause works, I'd like to demonstrate its use in both phases where it's applicable. In the following example, an OVER clause is used with the COUNT aggregate function in the SELECT list; the output of this query is shown in Table 1-22:

```
SELECT orderid, customerid,
  COUNT(*) OVER(PARTITION BY customerid) AS num_orders
FROM dbo.Orders
WHERE customerid IS NOT NULL
  AND orderid % 2 = 1;
```

Table 1-22 OVER Clause Applied in SELECT Phase

orderid	*customerid*	*num_orders*
1	FRNDO	1
3	KRLOS	2
5	KRLOS	2

The PARTITION BY clause defines the window for the calculation. The *COUNT(*)* function counts the number of rows in the virtual table provided to the SELECT phase as input, where the *customerid* is equal to the one in the current row. Remember that the virtual table provided to the SELECT phase as input has already undergone WHERE filtering—that is, NULL customer IDs and even order IDs have been eliminated.

You can also use the OVER clause in the ORDER BY list. For example, the following query sorts the rows according to the total number of output rows for the customer (in descending order), and generates the output shown in Table 1-23:

```
SELECT orderid, customerid
FROM dbo.Orders
WHERE customerid IS NOT NULL
  AND orderid % 2 = 1
ORDER BY COUNT(*) OVER(PARTITION BY customerid) DESC;
```

Table 1-23 OVER Clause Applied in ORDER BY Phase

orderid	*customerid*
3	KRLOS
5	KRLOS
1	FRNDO

More Info For details on using the OVER clause with aggregate functions, please refer to Chapter 6. For details on using the OVER clause with analytical ranking functions, please refer to Chapter 4.

Set Operations

SQL Server 2005 supports three set operations: UNION, EXCEPT, and INTERSECT. Only UNION is available in SQL Server 2000. These SQL operators correspond to operators defined in mathematical set theory. This is the syntax for a query applying a set operation:

```
[(]left_query[)] {UNION [ALL] | EXCEPT | INTERSECT} [(]right_query[)]
[ORDER BY <order_by_list>]
```

Set operations compare complete rows between the two inputs. UNION returns one result set with the rows from both inputs. If the ALL option is not specified, UNION removes duplicate rows from the result set. EXCEPT returns distinct rows that appear in the left input but not in the right. INTERSECT returns the distinct rows that appear in both inputs. There's much more to say about these set operations, but here I'd just like to focus on the logical processing steps involved in a set operation.

An ORDER BY clause is not allowed in the individual queries. You are allowed to specify an ORDER BY clause at the end of the query, but it will apply to the result of the set operation.

In terms of logical processing, each input query is first processed separately with all its relevant phases. The set operation is then applied, and if an ORDER BY clause is specified, it is applied to the result set.

Take the following query, which generates the output shown in Table 1-24, as an example:

```
SELECT 'O' AS letter, customerid, orderid FROM dbo.Orders
WHERE customerid LIKE '%O%'

UNION ALL

SELECT 'S' AS letter, customerid, orderid FROM dbo.Orders
WHERE customerid LIKE '%S%'

ORDER BY letter, customerid, orderid;
```

Table 1-24 Result of a UNION ALL Set Operation

letter	*customerid*	*orderid*
O	FRNDO	1
O	FRNDO	2
O	KRLOS	3
O	KRLOS	4

Table 1-24 Result of a UNION ALL Set Operation

letter	*customerid*	*orderid*
O	KRLOS	5
S	KRLOS	3
S	KRLOS	4
S	KRLOS	5
S	MRPHS	6

First, each input query is processed separately following all the relevant logical processing phases. The first query returns a table with orders placed by customers containing the letter *O*. The second query returns a table with orders placed by customers containing the letter *S*. The set operation UNION ALL combines the two sets into one. Finally, the ORDER BY clause sorts the rows by *letter*, *customerid*, and *orderid*.

As another example for logical processing phases of a set operation, the following query returns customers that have made no orders:

```
SELECT customerid FROM dbo.Customers
EXCEPT
SELECT customerid FROM dbo.Orders;
```

The first query returns the set of customer IDs from Customers ({FISSA, FRNDO, KRLOS, MRPHS}), and the second query returns the set of customer IDs from Orders ({FRNDO, FRNDO, KRLOS, KRLOS, KRLOS, MRPHS, NULL}). The set operation returns ({FISSA}), the set of rows from the first set that do not appear in the second set. Finally, the set operation removes duplicates from the result set. In this case, there are no duplicates to remove.

The result set's column names are determined by the set operation's left input. Columns in corresponding positions must match in their datatypes or be implicitly convertible. Finally, an interesting aspect of set operations is that they treat NULLs as equal.

More Info You can find a more detailed discussion about set operations in Chapter 5.

Conclusion

Understanding logical query processing phases and the unique aspects of SQL is important to get into the special mindset required to program in SQL. By being familiar with those aspects of the language, you will be able to produce efficient solutions and explain your choices. Remember, the idea is to master the basics.

Chapter 6

Aggregating and Pivoting Data

In this chapter:

This chapter covers various data-aggregation techniques, including the new OVER clause, tiebreakers, running aggregates, pivoting, unpivoting, custom aggregations, histograms, grouping factors, and the CUBE and ROLLUP options.

Throughout this chapter, in my solutions I'll reuse techniques that I introduced earlier. I'll also introduce new techniques for you to familiarize yourself with.

Logic will naturally be an integral element in the solutions. Remember that at the heart of every querying problem lies a logical puzzle.

OVER Clause

The OVER clause allows you to request window-based calculations—that is, the calculation is performed on a whole window of values. In Chapter 4, I described in detail how you use the OVER clause with the new analytical ranking functions. Microsoft SQL Server 2005 also introduces support for the OVER clause with scalar aggregate functions; however, currently it can be used only with the PARTITION BY clause. Hopefully, future versions of SQL Server will also support the other ANSI elements of aggregate window functions, including the ORDER BY and ROWS clauses.

The purpose of using the OVER clause with scalar aggregates is to calculate, for each row, an aggregate based on a window of values that extends beyond the scope of the row—and to do all this without using a GROUP BY clause in the query. In other words, the OVER clause allows you to add aggregate calculations to the results of an ungrouped query. This capability

provides an alternative to requesting aggregates with subqueries, in case you need to include both base row attributes and aggregates in your results.

As a reminder, in Chapter 5 I presented a problem in which you were required to calculate two aggregates for each sales row: the percentage the row contributed to the total sales quantity and the difference between the row's sales quantity and the average quantity over all sales. I showed the following optimized query in which I used a cross join between the base table and a derived table of aggregates, instead of using multiple subqueries:

```
SET NOCOUNT ON;
USE pubs;

SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / sumqty * 100 AS DECIMAL(5, 2)) AS per,
  CAST(qty - avgqty AS DECIMAL(9, 2)) as diff
FROM dbo.sales,
  (SELECT SUM(qty) AS sumqty, AVG(1.*qty) AS avgqty
   FROM dbo.sales) AS AGG;
```

This query produces the output shown in Table 6-1.

Table 6-1 Sales Percentage of Total and Diff from Average

stor_id	*ord_num*	*title_id*	*ord_date*	*qty*	*per*	*diff*
6380	6871	BU1032	1994-09-14	5	1.01	-18.48
6380	722a	PS2091	1994-09-13	3	0.61	-20.48
7066	A2976	PC8888	1993-05-24	50	10.14	26.52
7066	QA7442.3	PS2091	1994-09-13	75	15.21	51.52
7067	D4482	PS2091	1994-09-14	10	2.03	-13.48
7067	P2121	TC3218	1992-06-15	40	8.11	16.52
7067	P2121	TC4203	1992-06-15	20	4.06	-3.48
7067	P2121	TC7777	1992-06-15	20	4.06	-3.48
7131	N914008	PS2091	1994-09-14	20	4.06	-3.48
7131	N914014	MC3021	1994-09-14	25	5.07	1.52
7131	P3087a	PS1372	1993-05-29	20	4.06	-3.48
7131	P3087a	PS2106	1993-05-29	25	5.07	1.52
7131	P3087a	PS3333	1993-05-29	15	3.04	-8.48
7131	P3087a	PS7777	1993-05-29	25	5.07	1.52
7896	QQ2299	BU7832	1993-10-28	15	3.04	-8.48
7896	TQ456	MC2222	1993-12-12	10	2.03	-13.48
7896	X999	BU2075	1993-02-21	35	7.10	11.52
8042	423LL922	MC3021	1994-09-14	15	3.04	-8.48
8042	423LL930	BU1032	1994-09-14	10	2.03	-13.48
8042	P723	BU1111	1993-03-11	25	5.07	1.52
8042	QA879.1	PC1035	1993-05-22	30	6.09	6.52

The motivation for calculating the two aggregates in a single derived table instead of as two separate subqueries stemmed from the fact that each subquery accessed the table/index, while the derived table calculated the aggregates using a single scan of the data.

Similarly, you can calculate multiple aggregates using the same OVER clause, and SQL Server will scan the required source data only once for all. Here's how you use the OVER clause to answer the same request:

```
SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / SUM(qty) OVER() * 100 AS DECIMAL(5, 2)) AS per,
  CAST(qty - AVG(1.*qty) OVER() AS DECIMAL(9, 2)) AS diff
FROM dbo.sales;
```

Note In Chapter 4, I described the PARTITION BY clause, which is used with window functions, including aggregate window functions. This clause is optional. When not specified, the aggregate is based on the whole input rather than being calculated per partition.

Here, because I didn't specify a PARTITION BY clause, the aggregates were calculated based on the whole input. Logically, *SUM(qty) OVER()* is equivalent here to the subquery *(SELECT SUM(qty) FROM dbo.sales)*. Physically, it's a different story. As an exercise, you can compare the execution plans of the following two queries, each requesting a different number of aggregates using the same OVER clause:

```
SELECT stor_id, ord_num, title_id,
  SUM(qty) OVER() AS sumqty
FROM dbo.sales;

SELECT stor_id, ord_num, title_id,
  SUM(qty)   OVER() AS sumqty,
  COUNT(qty) OVER() AS cntqty,
  AVG(qty)   OVER() AS avgqty,
  MIN(qty)   OVER() AS minqty,
  MAX(qty)   OVER() AS maxqty
FROM dbo.sales;
```

You'll find the two plans nearly identical, with the only difference being that the single *Stream Aggregate* operator calculates a different number of aggregates for each. The query costs are identical. On the other hand, compare the execution plans of the following two queries, each requesting a different number of aggregates using subqueries:

```
SELECT stor_id, ord_num, title_id,
  (SELECT SUM(qty) FROM dbo.sales) AS sumqty
FROM dbo.sales;

SELECT stor_id, ord_num, title_id,
  (SELECT SUM(qty)   FROM dbo.sales) AS sumqty,
  (SELECT COUNT(qty) FROM dbo.sales) AS cntqty,
  (SELECT AVG(qty)   FROM dbo.sales) AS avgqty,
```

```
  (SELECT MIN(qty)   FROM dbo.sales) AS minqty,
  (SELECT MAX(qty)   FROM dbo.sales) AS maxqty
FROM dbo.sales;
```

You'll find that they have different plans, with the latter being more expensive, as it rescans the source data for each aggregate.

Another benefit of the OVER clause is that it allows for shorter and simpler code. This is especially apparent when you need to calculate partitioned aggregates. Using OVER, you simply specify a PARTITION BY clause. Using subqueries, you have to correlate the inner query to the outer, making the query longer and more complex.

As an example for using the PARTITION BY clause, the following query calculates the percentage of the quantity out of the store total and the difference from the store average, yielding the output shown in Table 6-2:

```
SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / SUM(qty) OVER(PARTITION BY stor_id) * 100
    AS DECIMAL(5, 2)) AS per,
  CAST(qty - AVG(1.*qty) OVER(PARTITION BY stor_id)
    AS DECIMAL(9, 2)) AS diff
FROM dbo.sales
ORDER BY stor_id;
```

Table 6-2 Sales Percentage of Store Total and Diff from Store Average

stor_id	*ord_num*	*title_id*	*ord_date*	*qty*	*per*	*diff*
6380	6871	BU1032	1994-09-14	5	62.50	1.00
6380	722a	PS2091	1994-09-13	3	37.50	-1.00
7066	A2976	PC8888	1993-05-24	50	40.00	-12.50
7066	QA7442.3	PS2091	1994-09-13	75	60.00	12.50
7067	D4482	PS2091	1994-09-14	10	11.11	-12.50
7067	P2121	TC3218	1992-06-15	40	44.44	17.50
7067	P2121	TC4203	1992-06-15	20	22.22	-2.50
7067	P2121	TC7777	1992-06-15	20	22.22	-2.50
7131	N914008	PS2091	1994-09-14	20	15.38	-1.67
7131	N914014	MC3021	1994-09-14	25	19.23	3.33
7131	P3087a	PS1372	1993-05-29	20	15.38	-1.67
7131	P3087a	PS2106	1993-05-29	25	19.23	3.33
7131	P3087a	PS3333	1993-05-29	15	11.54	-6.67
7131	P3087a	PS7777	1993-05-29	25	19.23	3.33
7896	QQ2299	BU7832	1993-10-28	15	25.00	-5.00
7896	TQ456	MC2222	1993-12-12	10	16.67	-10.00
7896	X999	BU2075	1993-02-21	35	58.33	15.00

Table 6-2 Sales Percentage of Store Total and Diff from Store Average

stor_id	ord_num	title_id	ord_date	qty	per	diff
8042	423LL922	MC3021	1994-09-14	15	18.75	-5.00
8042	423LL930	BU1032	1994-09-14	10	12.50	-10.00
8042	P723	BU1111	1993-03-11	25	31.25	5.00
8042	QA879.1	PC1035	1993-05-22	30	37.50	10.00

In short, the OVER clause allows for shorter and faster queries.

Tiebreakers

In this section, I want to introduce a new technique based on aggregates to solve tiebreaker problems, which I started discussing in Chapter 4. I'll use the same example as I used there—returning the most recent order for each employee—using different combinations of tiebreaker attributes that uniquely identify an order for each employee. Keep in mind that the performance of the solutions that use subqueries very strongly depends on indexing. That is, you need an index on the partitioning column, sort column, and tiebreaker attributes. But in practice, you don't always have the option to add as many indexes as you like. The subquery-based solutions will greatly suffer in performance from a lack of appropriate indexes. Using aggregation techniques, you'll see that the solution will yield good performance even when an optimal index is not in place—in fact, even when no good index is in place.

Let's start with using the *MAX(OrderID)* as the tiebreaker. To recap, you're after the most recent order for each employee, using the *MAX(OrderID)* as the tiebreaker. For each order, you're supposed to return the *EmployeeID*, *OrderDate*, *OrderID*, *CustomerID*, and *RequiredDate*.

The aggregate technique to solve the problem applies the following logical idea in pseudocode:

```
SELECT EmployeeID, MAX(OrderDate, OrderID, CustomerID, RequiredDate)
FROM dbo.Orders
GROUP BY EmployeeID;
```

There's no such ability in T-SQL, so don't try to run this query. The idea here is to generate a row for each employee, with the *MAX(OrderDate)* (most recent) and the *MAX(OrderID)*—the tiebreaker—among orders on the most recent *OrderDate*. Because the combination *EmployeeID*, *OrderDate*, *OrderID* is already unique, all other attributes (*CustomerID*, *RequiredDate*) are simply returned from the selected row. Because a MAX of more than one attribute does not exist in T-SQL, you must mimic it somehow, and you can do so by concatenating all attributes to provide a scalar input value to the MAX function, and then in an outer query, extract back the individual elements.

The question is this: what technique should you use to concatenate the attributes? The trick is to use a fixed-width string for each attribute and to convert the attributes in a way that will not

change the sorting behavior. When dealing exclusively with positive numbers, you can use an arithmetic calculation to merge values. For example, say you have the numbers *m* and *n*, each with a valid range of 1 through 999. To merge *m* and *n*, use the following formula: *m*1000 + n AS r*. To later extract the individual pieces, use *r* divided by 1000 to get *m*, and use *r* modulo 1000 to get *n*. However, in many cases you'll probably have non-numeric data to concatenate, so arithmetic concatenation would be out of the question. You might want to consider converting all values to fixed-width character strings (*CHAR(n)/NCHAR(n)*) or to fixed-width binary strings (*BINARY(n)*).

Here's an example for returning the order with the *MAX(OrderDate)* for each employee, using *MAX(OrderID)* as the tiebreaker, using binary concatenation:

```
USE Northwind;

SELECT EmployeeID,
  CAST(SUBSTRING(binstr, 1, 8)   AS DATETIME) AS OrderDate,
  CAST(SUBSTRING(binstr, 9, 4)   AS INT)      AS OrderID,
  CAST(SUBSTRING(binstr, 13, 10) AS NCHAR(5)) AS CustomerID,
  CAST(SUBSTRING(binstr, 23, 8)  AS DATETIME) AS RequiredDate
FROM (SELECT EmployeeID,
        MAX(CAST(OrderDate        AS BINARY(8))
              + CAST(OrderID      AS BINARY(4))
              + CAST(CustomerID   AS BINARY(10))
              + CAST(RequiredDate AS BINARY(8))) AS binstr
      FROM dbo.Orders
      GROUP BY EmployeeID) AS D;
```

The derived table D contains the maximum concatenated string for each employee. Notice that each value was converted to the appropriate fixed-size string before concatenation based on its datatype (DATETIME—8 bytes, INT—4 bytes, and so on).

Note When converting numbers to binary strings, only nonnegative values will preserve their original sort behavior. As for character strings, converting them to binary values makes them use similar sort behavior to a binary sort order.

The outer query uses SUBSTRING functions to extract the individual elements, and it converts them back to their original datatypes.

The real benefit in this solution is that it scans the data only once regardless of whether you have a good index or not. If you do, you'll probably get an ordered scan of the index and a sort-based aggregate. If you don't—as is the case here—you'll probably get a hash-based aggregate, as you can see in Figure 6-1.

Figure 6-1 Execution plan for a tiebreaker query

Things get trickier when the sort columns and tiebreaker attributes have different sort directions within them. For example, suppose the tiebreaker was *MIN(OrderID)*. In that case, you would need to apply a MAX to *OrderDate*, and MIN to *OrderID*. There is a logical solution when the attribute with the opposite direction is numeric. Say you need to calculate the MIN value of a nonnegative integer column *n*, using only MAX. This can be achieved by using *<maxint> - MAX(<maxint> - n)*.

The following query incorporates this logical technique:

```
SELECT EmployeeID,
  CAST(SUBSTRING(binstr, 1, 8)   AS DATETIME) AS OrderDate,
  2147483647 - CAST(SUBSTRING(binstr, 9, 4) AS INT) AS OrderID,
  CAST(SUBSTRING(binstr, 13, 10) AS NCHAR(5)) AS CustomerID,
  CAST(SUBSTRING(binstr, 23, 8)  AS DATETIME) AS RequiredDate
FROM (SELECT EmployeeID,
        MAX(CAST(OrderDate        AS BINARY(8))
              + CAST(2147483647 - OrderID AS BINARY(4))
              + CAST(CustomerID   AS BINARY(10))
              + CAST(RequiredDate AS BINARY(8))) AS binstr
      FROM dbo.Orders
      GROUP BY EmployeeID) AS D;
```

Of course, you can play with the tiebreakers you're using in any way you like. For example, here's the query that will return the most recent order for each employee, using *MAX(RequiredDate), MAX(OrderID)* as the tiebreaker:

```
SELECT EmployeeID,
  CAST(SUBSTRING(binstr, 1, 8)   AS DATETIME) AS OrderDate,
  CAST(SUBSTRING(binstr, 9, 8)   AS DATETIME) AS RequiredDate,
  CAST(SUBSTRING(binstr, 17, 4)  AS INT)      AS OrderID,
  CAST(SUBSTRING(binstr, 21, 10) AS NCHAR(5)) AS CustomerID
FROM (SELECT EmployeeID,
        MAX(CAST(OrderDate        AS BINARY(8))
              + CAST(RequiredDate AS BINARY(8))
              + CAST(OrderID      AS BINARY(4))
              + CAST(CustomerID   AS BINARY(10))
              ) AS binstr
      FROM dbo.Orders
      GROUP BY EmployeeID) AS D;
```

Running Aggregations

Running aggregations are aggregations of data over a sequence (typically temporal). There are many variations of running aggregate problems, and I'll describe several important ones here.

In my examples, I'll use a summary table called EmpOrders that contains one row for each employee and month, with the total quantity of orders made by that employee in that month. Run the code in Listing 6-1 to create the EmpOrders table, and populate the table with sample data.

Listing 6-1 Creating and populating the EmpOrders table

```
USE tempdb;
GO

IF OBJECT_ID('dbo.EmpOrders') IS NOT NULL
  DROP TABLE dbo.EmpOrders;
GO

CREATE TABLE dbo.EmpOrders
(
  empid    INT      NOT NULL,
  ordmonth DATETIME NOT NULL,
  qty      INT      NOT NULL,
  PRIMARY KEY(empid, ordmonth)
);

INSERT INTO dbo.EmpOrders(empid, ordmonth, qty)
  SELECT O.EmployeeID,
    CAST(CONVERT(CHAR(6), O.OrderDate, 112) + '01'
      AS DATETIME) AS ordmonth,
    SUM(Quantity) AS qty
  FROM Northwind.dbo.Orders AS O
    JOIN Northwind.dbo.[Order Details] AS OD
      ON O.OrderID = OD.OrderID
  GROUP BY EmployeeID,
    CAST(CONVERT(CHAR(6), O.OrderDate, 112) + '01'
      AS DATETIME);
```

Tip I will represent each month by its start date stored as a DATETIME. This will allow flexible manipulation of the data using date-related functions. To ensure the value would be valid in the datatype, I stored the first day of the month as the day portion. Of course, I'll ignore it in my calculations.

Run the following query to get the contents of the EmpOrders table, which is shown in abbreviated form in Table 6-3:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty
FROM dbo.EmpOrders
ORDER BY empid, ordmonth;
```

Table 6-3 Contents of EmpOrders Table (Abbreviated)

empid	*ordmonth*	*qty*
1	1996-07	121
1	1996-08	247
1	1996-09	255
1	1996-10	143
1	1996-11	318

Table 6-3 Contents of EmpOrders Table (Abbreviated)

empid	*ordmonth*	*qty*
1	1996-12	536
1	1997-01	304
1	1997-02	168
1	1997-03	275
1	1997-04	20
...	...	...
2	1996-07	50
2	1996-08	94
2	1996-09	137
2	1996-10	248
2	1996-11	237
2	1996-12	319
2	1997-01	230
2	1997-02	36
2	1997-03	151
2	1997-04	468
...	...	...

I'll discuss three types of running aggregation problems: cumulative, sliding, and year-to-date (YTD).

Cumulative Aggregations

Cumulative aggregations accumulate data from the first element within the sequence up to the current point. For example, imagine the following request: for each employee and month, return the total quantity and average monthly quantity from the beginning of the employee's activity to the month in question.

Recall the pre–SQL Server 2005 set-based techniques for calculating row numbers; using these techniques, you scan the same rows we need to scan now to calculate the total quantities. The difference is that for row numbers you used the aggregate COUNT, and here you're asked for the SUM and the AVG. I demonstrated two solutions to calculate row numbers—one using subqueries and one using joins. In the solution using joins, I applied what I called an *expand-collapse technique*. To me, the subquery solution is much more intuitive than the join solution, with its artificial expand-collapse technique. So, when there's no performance difference, I'd rather use subqueries. Typically, you won't see a performance difference when only one aggregate is involved, as the plans would be similar. However, when you request multiple aggregates, the subquery solution might result in a plan that scans the data separately for each aggregate. Compare this to the plan for the join solution, which typically calculates all aggregates during a single scan of the source data.

So my choice is usually simple—use a subquery for one aggregate, and a join for multiple aggregates. The following query applies the expand-collapse approach to produce the desired result, which is shown in abbreviated form in Table 6-4:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

Table 6-4 Cumulative Aggregates Per Employee, Month (Abbreviated)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
1	1996-12	536	1620	270.00
1	1997-01	304	1924	274.86
1	1997-02	168	2092	261.50
1	1997-03	275	2367	263.00
1	1997-04	20	2387	238.70
...	...	...	...	...
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	1996-12	319	1085	180.83
2	1997-01	230	1315	187.86
2	1997-02	36	1351	168.88
2	1997-03	151	1502	166.89
2	1997-04	468	1970	197.00
...	...	...	...	...

Now let's say that you were asked to return only one aggregate (say, total quantity). You can safely use the subquery approach:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
```

```
  (SELECT SUM(O2.qty)
   FROM dbo.EmpOrders AS O2
   WHERE O2.empid = O1.empid
     AND O2.ordmonth <= O1.ordmonth) AS totalqty
FROM dbo.EmpOrders AS O1
GROUP BY O1.empid, O1.ordmonth, O1.qty;
```

Note In both cases, the same N^2 performance issues I discussed with regard to row numbers apply here as well. Because running aggregates typically are calculated on a fairly small number of rows per group, you won't be adversely affected by performance issues, assuming you have appropriate indexes (grouping_columns, sort_columns, covering_columns).

ANSI SQL:2003 and OLAP extensions to ANSI SQL:1999 provide support for running aggregates by means of aggregate window functions. As I mentioned earlier, SQL Server 2005 implemented the OVER clause for aggregate functions only with the PARTITION BY clause. Per ANSI, you could provide a solution relying exclusively on window functions like so:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty,
  SUM(O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth) AS totalqty,
  CAST(AVG(1.*O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth)
  AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders;
```

When this code is finally supported in SQL Server, you can expect dramatic performance improvements, and obviously much simpler queries.

You might also be requested to filter the data—for example, return monthly aggregates for each employee only for months before the employee reached a certain target. Typically, you'll have a target for each employee stored in a Targets table that you'll need to join to. To make this example simple, I'll assume that all employees have the same target total quantity—1000. In practice, you'll use the target attribute from the Targets table. Because you need to filter an aggregate, not an attribute, you must specify the filter expression (in this case, *SUM(O2.qty) < 1000*) in the HAVING clause, not the WHERE clause. The solution is as follows and will yield the output shown in abbreviated form in Table 6-5:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) < 1000
ORDER BY O1.empid, O1.ordmonth;
```

Table 6-5 Cumulative Aggregates, Where totalqty < 1000 (Abbreviated)

empid	ordmonth	qtythismonth	totalqty	avgqty
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
3	...	...	...	...

Things get a bit tricky if you also need to include the rows for those months in which the employees reached their target. If you specify *SUM(O2.qty) <= 1000* (that is, write <= instead of <), you still won't get the row in which the employee reached the target unless the total through that month is exactly 1000. But remember that you have access to both the cumulative total and the current month's quantity, and using these two values together, you can solve this problem. If you change the HAVING filter to *SUM(O2.qty) – O1.qty < 1000*, you will get the months in which the employee's total quantity, *excluding the current month's orders*, had not reached the target. In particular, the first month in which an employee reached or exceeded the target satisfies this new criterion, and that month will appear in the results. The complete solution follows, and it yields the output shown in abbreviated form in Table 6-6:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) - O1.qty < 1000
ORDER BY O1.empid, O1.ordmonth;
```

Table 6-6 Cumulative Aggregates, Until totalqty First Reaches or Exceeds 1000 (Abbreviated)

empid	ordmonth	qtythismonth	totalqty	avgqty
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80

Table 6-6 Cumulative Aggregates, Until totalqty First Reaches or Exceeds 1000 (Abbreviated)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	1996-12	319	1085	180.83
3	...	...	...	...

Note You might have another solution in mind that would seem like a plausible and simpler alternative—to leave the SUM condition alone but change the join condition to *O2.ordmonth < O1.ordmonth*. This way, the query would select rows where the total through the previous month did not meet the target. However, in the end, this solution is not any easier (the AVG is hard to generate, for example); and worse, you might come up with a solution that does not work for employees who reach the target in their first month.

Suppose you were interested in seeing results only for the specific month in which the employee reached the target of 1000, without seeing results for preceding months. What's true for only those rows of Table 6-6? What you're looking for are rows from Table 6-6 where the total quantity is greater than or equal to 1000. Simply add this criterion to the HAVING filter. Here's the query, which will yield the output shown in Table 6-7:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) - O1.qty < 1000
  AND SUM(O2.qty) >= 1000
ORDER BY O1.empid, O1.ordmonth;
```

Table 6-7 Cumulative Aggregates only for Months in Which totalqty First Reaches or Exceeds 1000

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-11	318	1084	216.80
2	1996-12	319	1085	180.83
3	1997-01	364	1304	186.29
4	1996-10	613	1439	359.75
5	1997-05	247	1213	173.29

Table 6-7 Cumulative Aggregates only for Months in Which totalqty First Reaches or Exceeds 1000

empid	ordmonth	qtythismonth	totalqty	avgqty
6	1997-01	64	1027	171.17
7	1997-03	191	1069	152.71
8	1997-01	305	1228	175.43
9	1997-06	161	1007	125.88

Sliding Aggregations

Sliding aggregates are calculated over a sliding window in a sequence (again, typically temporal), as opposed to being calculated from the beginning of the sequence until the current point. A *moving average*—such as the employee's average quantity over the last three months—is one example of a sliding aggregate.

> **Note** Without clarification, expressions like "last three months" are ambiguous. The last three months could mean the previous three months (*not including this month*), or it could mean the previous two months *along with this month*. When you get a problem like this, be sure you know precisely what window of time you are using for aggregation—for a particular row, exactly when does the window begin and end?
>
> In our example, the window of time is: greater than the point in time starting three months ago and smaller than or equal to the current point in time. Note that this definition will work well even in cases where you track finer time granularities than a month (including day, hour, minute, second, and millisecond). This definition also addresses implicit conversion issues due to the accuracy level supported by SQL Server for the DATETIME datatype—3.33 milliseconds. It's wiser to use > and <= predicates than the BETWEEN predicate to avoid implicit conversion issues.

The main difference between the solution for cumulative aggregates and the solution for running aggregates is in the join condition (or in the subquery's filter, in the case of the alternate solution using subqueries). Instead of using *O2.ordmonth* <= *O1.current_month*, you use *O2.ordmonth* > *three_months_before_current AND O2.ordmonth* <= *current_month*. In T-SQL, this translates to the following query, yielding the output shown in abbreviated form in Table 6-8:

```
SELECT O1.empid,
  CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND (O2.ordmonth > DATEADD(month, -3, O1.ordmonth)
     AND O2.ordmonth <=  O1.ordmonth)
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

Table 6-8 Sliding Aggregates Per Employee over Three Months Leading to Current (Abbreviated)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	645	215.00
1	1996-11	318	716	238.67
1	1996-12	536	997	332.33
1	1997-01	304	1158	386.00
1	1997-02	168	1008	336.00
1	1997-03	275	747	249.00
1	1997-04	20	463	154.33
...	...	...	...	...
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	479	159.67
2	1996-11	237	622	207.33
2	1996-12	319	804	268.00
2	1997-01	230	786	262.00
2	1997-02	36	585	195.00
2	1997-03	151	417	139.00
2	1997-04	468	655	218.33
...	...	...	...	...

Note that this solution includes aggregates for three-month periods that don't include three months of actual data. If you want to return only periods with three full months accumulated, without the first two periods which do not cover three months, you can add the criterion *MIN(O2.ordmonth) = DATEADD(month, -2, O1.ordmonth)* to the HAVING filter.

Note In addition to supporting both the PARTITION BY and ORDER BY elements in the OVER clause for window-based aggregations, ANSI also supports a ROWS clause that allows you to request sliding aggregates. For example, here's the query that would return the desired result for the last sliding aggregates request (assuming the data has exactly one row per month):

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth,
  qty AS qtythismonth,
  SUM(O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth
                   ROWS 2 PRECEDING) AS totalqty,
  CAST(AVG(1.*O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth
                           ROWS 2 PRECEDING)
    AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders;
```

Year-To-Date (YTD)

YTD aggregates accumulate values from the beginning of a period based on some DATETIME unit (say, a year) until the current point. The calculation is very similar to the sliding aggregates solution. The only difference is the low bound provided in the query's filter, which is the calculation of the beginning of the year. For example, the following query returns YTD aggregates for each employee and month, yielding the output shown in abbreviated form in Table 6-9:

```
SELECT O1.empid,
  CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND (O2.ordmonth >= CAST(CAST(YEAR(O1.ordmonth) AS CHAR(4))
                             + '0101' AS DATETIME)
         AND O2.ordmonth <= O1.ordmonth)
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

Table 6-9 YTD Aggregates Per Employee, Month (Abbreviated)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
1	1996-12	536	1620	270.00
1	1997-01	304	304	304.00
1	1997-02	168	472	236.00
1	1997-03	275	747	249.00
1	1997-04	20	767	191.75
...	...	...	...	...
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	1996-12	319	1085	180.83
2	1997-01	230	230	230.00
2	1997-02	36	266	133.00

Table 6-9 YTD Aggregates Per Employee, Month (Abbreviated)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
2	1997-03	151	417	139.00
2	1997-04	468	885	221.25
...	...	...	...	...

Pivoting

Pivoting is a technique that allows you to rotate rows to columns, possibly performing aggregations along the way. The number of applications for pivoting is simply astounding. In this section, I'll present a few, including pivoting attributes in an Open Schema environment, solving relational division problems, and formatting aggregated data. Later in the chapter and also in other chapters in the book, I'll show additional applications. As usual for this book, I'll present solutions that apply to versions earlier than SQL Server 2005 as well as solutions that use newly introduced specialized operators and therefore work only in SQL Server 2005.

Pivoting Attributes

I'll use *open schema* as the scenario for pivoting attributes. Open schema is a schema design you create to deal with frequent schema changes. The relational model and SQL do a very good job with data manipulation (DML), which includes changing and querying data. However, SQL's data definition language (DDL) does not make it easy to deal with frequent schema changes. Whenever you need to add new entities, you must create new tables; whenever existing entities change their structures, you must add, alter, or drop columns. Such changes usually require downtime of the affected objects, and they also bring about substantial revisions to the application.

In a scenario with frequent schema changes, you can store all data in a single table, where each attribute value resides in its own row along with the entity or object ID and the attribute name or ID. You represent the attribute values using the datatype SQL_VARIANT to accommodate multiple attribute types in a single column.

In my examples, I'll use the OpenSchema table, which you can create and populate by running the code in Listing 6-2.

Listing 6-2 Creating and populating the OpenSchema table

```
SET NOCOUNT ON;
USE tempdb;
GO

IF OBJECT_ID('dbo.OpenSchema') IS NOT NULL
  DROP TABLE dbo.OpenSchema;
GO
```

```
CREATE TABLE dbo.OpenSchema
(
  objectid  INT          NOT NULL,
  attribute NVARCHAR(30) NOT NULL,
  value     SQL_VARIANT  NOT NULL,
  PRIMARY KEY (objectid, attribute)
);

INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(1, N'attr1', CAST('ABC'      AS VARCHAR(10))  );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(1, N'attr2', CAST(10         AS INT)          );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(1, N'attr3', CAST('20040101' AS SMALLDATETIME));
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr2', CAST(12         AS INT)          );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr3', CAST('20060101' AS SMALLDATETIME));
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr4', CAST('Y'        AS CHAR(1))      );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr5', CAST(13.7       AS DECIMAL(9,3)) );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(3, N'attr1', CAST('XYZ'      AS VARCHAR(10))  );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(3, N'attr2', CAST(20         AS INT)          );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(3, N'attr3', CAST('20050101' AS SMALLDATETIME));
```

The contents of the OpenSchema table are shown in Table 6-10.

Table 6-10 Contents of OpenSchema Table

objectid	*attribute*	value
1	attr1	ABC
1	attr2	10
1	attr3	2004-01-01 00:00:00.000
2	attr2	12
2	attr3	2006-01-01 00:00:00.000
2	attr4	Y
2	attr5	13.700
3	attr1	XYZ
3	attr2	20
3	attr3	2005-01-01 00:00:00.000

Representing data this way allows logical schema changes to be implemented without adding, altering, or dropping tables and columns, but by using DML INSERTs, UPDATEs, and DELETEs instead. Of course, other aspects of working with the data (such as enforcing integrity, tuning, and querying) become more complex and expensive with such a representation.

There are other approaches to deal with frequent data definition changes—for example, storing the data in XML format. However, when you weigh the advantages and disadvantages of each representation, you might find the open schema representation demonstrated here more favorable in some scenarios—for example, representing auction data.

Keep in mind that this representation of the data requires very complex queries even for simple requests, because different attributes of the same entity instance are spread over multiple rows. Before you query such data, you might want to rotate it to a traditional form with one column for each attribute—perhaps store the result in a temporary table, index it, query it, and then get rid of the temporary table. To rotate the data from its open schema form into a traditional form, you need to use a pivoting technique.

In the following section, I'll describe the steps involved in solving pivoting problems. I'd like to point out that to understand the steps of the solution, it can be very helpful if you think about query logical processing phases, which I described in detail in Chapter 1. I discussed the query processing phases involved with the PIVOT table operator in SQL Server 2005, but those phases apply just as well to the solution in SQL Server 2000. Moreover, in SQL 2000 the phases are more apparent in the code, while in SQL Server 2005 they are implicit.

The first step you might want to try when solving pivoting problems is to figure out how the number of rows in the result correlates to the number of rows in the source data. Here, you need to create a single result row out of the multiple base rows for each object. This can mean creating a GROUP BY *objectid*.

As the next step in a pivoting problem, you can think in terms of the result columns. You need a result column for each unique attribute. Because the data contains five unique attributes (attr1, attr2, attr3, attr4, and attr5), you need five expressions in the SELECT list. Each expression is supposed to extract, out of the rows belonging to the grouped object, the value corresponding to a specific attribute. This can be done with the following MAX(CASE...) expression, which in this example is applied to the attribute attr2:

```
MAX(CASE WHEN attribute = 'attr2' THEN value END) AS attr2
```

Remember that with no ELSE clause CASE assumes an implicit ELSE NULL. The CASE expression just shown will yield NULL for rows where *attribute* does not equal 'attr2' and yield *value* when *attribute* does equal 'attr2'. This means that among the rows with a given value of *objectid* (say, 1), the CASE expression would yield several NULLs and, at most, one known value (10 in our example), which represents the value of the target attribute (*attr2* in our example) for the given *objectid*. The trick to extracting the one known value is to use MAX or MIN. Both ignore NULLs and will return the one non-NULL value present, because both the minimum and the maximum of a set containing one value is that value. Here's the complete query that pivots the attributes from OpenSchema, yielding the output shown in Table 6-11:

```
SELECT objectid,
  MAX(CASE WHEN attribute = 'attr1' THEN value END) AS attr1,
  MAX(CASE WHEN attribute = 'attr2' THEN value END) AS attr2,
```

```
  MAX(CASE WHEN attribute = 'attr3' THEN value END) AS attr3,
  MAX(CASE WHEN attribute = 'attr4' THEN value END) AS attr4,
  MAX(CASE WHEN attribute = 'attr5' THEN value END) AS attr5
FROM dbo.OpenSchema
GROUP BY objectid;
```

Table 6-11 Pivoted OpenSchema

objectid	*attr1*	*attr2*	*attr3*	*attr4*	*attr5*
1	ABC	10	2004-01-01 00:00:00.000	NULL	NULL
2	NULL	12	2006-01-01 00:00:00.000	Y	13.700
3	XYZ	20	2005-01-01 00:00:00.000	NULL	NULL

Note To write this query, you have to know the names of the attributes. If you don't, you'll need to construct the query string dynamically.

More Info For details about dynamic pivoting (and unpivoting), please refer to *Inside Microsoft SQL Server 2005: T-SQL Programming* (Microsoft Press, 2006).

This technique for pivoting data is very efficient because it scans the base table only once.

SQL Server 2005 introduces PIVOT, a native specialized operator for pivoting. I have to say that I find it very confusing and nonintuitive. I don't see much advantage in using it, except that it allows for shorter code. It doesn't support dynamic pivoting, and underneath the covers, it applies very similar logic to the one I presented in the last solution. So you probably won't even find noticeable performance differences. At any rate, here's how you would pivot the OpenSchema data using the PIVOT operator:

```
SELECT objectid, attr1, attr2, attr3, attr4, attr5
FROM dbo.OpenSchema
  PIVOT(MAX(value) FOR attribute
    IN([attr1],[attr2],[attr3],[attr4],[attr5])) AS P;
```

Within this solution, you can identify all the elements I used in the previous solution. The inputs to the PIVOT operator are as follows:

- The aggregate that will apply to the values in the group. In our case, it's *MAX(value)*, which extracts the single non-NULL value corresponding to the target attribute. In other cases, you might have more than one non-NULL value per group and want a different aggregate (for example, SUM or AVG).
- Following the FOR keyword, the source column holding the target column names (*attribute*, in our case).
- The list of actual target column names in parentheses following the keyword IN.

The tricky bit here is that there's no explicit GROUP BY clause, but implicit grouping does take place. It's as if the pivoting activity is based on groups defined by the list of all columns that were not mentioned in PIVOT's inputs (in the parentheses) following the PIVOT keyword). In our case, *objectid* is the column that defines the groups.

Caution Because all unspecified columns define the groups, unintentionally, you might end up with undesired grouping. To solve this, use a derived table or a common table expression (CTE) that returns only the columns of interest, and apply PIVOT to that table expression and not to the base table. I'll demonstrate this shortly.

Tip The input to the aggregate function must be a base column with no manipulation—it cannot be an expression (for example: *SUM(qty * price)*). If you want to provide the aggregate function with an expression as input, create a derived table or CTE where you assign the expression with a column alias (*qty * price AS value*), and in the outer query use that column as input to PIVOT's aggregate function (*SUM(value)*).

Also, you cannot rotate attributes from more than one column (the column that appears after the FOR keyword. If you need to pivot more that one column's attributes (say, *empid* and *YEAR(orderdate)*), you can use a similar approach to the previous suggestion; create a derived table or a CTE where you concatenate the values from all columns you want to rotate and assign the expression with a column alias (*CAST(empid AS VARCHAR(10)) + '_' + CAST(YEAR(orderdate) AS CHAR(4)) AS empyear*). Then, in the outer query, specify that column after PIVOT's FOR keyword (*FOR empyear IN([1_2004], [1_2005], [1_2006], [2_2004], ...)*).

Relational Division

Pivoting can also be used to solve relational division problems when the number of elements in the divisor set is fairly small. In my examples, I'll use the OrderDetails table, which you create and populate by running the code in Listing 6-3.

Listing 6-3 Creating and populating the OrderDetails table

```
USE tempdb;
GO

IF OBJECT_ID('dbo.OrderDetails') IS NOT NULL
  DROP TABLE dbo.OrderDetails;
GO

CREATE TABLE dbo.OrderDetails
(
  orderid   VARCHAR(10) NOT NULL,
  productid INT         NOT NULL,
  PRIMARY KEY(orderid, productid)
  /* other columns */
);
```

```
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 1);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 2);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 3);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 4);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('B', 2);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('B', 3);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('B', 4);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('C', 3);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('C', 4);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('D', 4);
```

A classic relational division problem is to return orders that contain a certain basket of products—say, products 2, 3, and 4. You use a pivoting technique to rotate only the relevant products into separate columns for each order. Instead of returning an actual attribute value, you produce a 1 if the product exists in the order and a 0 otherwise. Create a derived table out of the pivot query, and in the outer query filter only orders that contain a 1 in all product columns. Here's the full query, which correctly returns orders A and B:

```
SELECT orderid
FROM (SELECT
        orderid,
        MAX(CASE WHEN productid = 2 THEN 1 END) AS P2,
        MAX(CASE WHEN productid = 3 THEN 1 END) AS P3,
        MAX(CASE WHEN productid = 4 THEN 1 END) AS P4
      FROM dbo.OrderDetails
      GROUP BY orderid) AS P
WHERE P2 = 1 AND P3 = 1 AND P4 = 1;
```

If you run only the derived table query, you get the pivoted products for each order as shown in Table 6-12.

Table 6-12 Contents of Derived Table P

orderid	*P2*	*P3*	*P4*
A	1	1	1
B	1	1	1
C	NULL	1	1
D	NULL	NULL	1

To answer the request at hand using the new PIVOT operator, use the following query:

```
SELECT orderid
FROM (SELECT *
      FROM dbo.OrderDetails
        PIVOT(MAX(productid) FOR productid IN([2],[3],[4])) AS P) AS T
WHERE [2] = 2 AND [3] = 3 AND [4] = 4;
```

The aggregate function must accept a column as input, so I provided the *productid* itself. This means that if the product exists within an order, the corresponding value will contain the actual *productid* and not 1. That's why the filter looks a bit different here.

Note that you can make both queries more intuitive and similar to each other in their logic by using the COUNT aggregate instead of MAX. This way, both queries would produce a 1 where the product exists and a 0 where it doesn't (instead of NULL). Here's what the SQL Server 2000 query would look like:

```
SELECT orderid
FROM (SELECT
        orderid,
        COUNT(CASE WHEN productid = 2 THEN 1 END) AS P2,
        COUNT(CASE WHEN productid = 3 THEN 1 END) AS P3,
        COUNT(CASE WHEN productid = 4 THEN 1 END) AS P4
      FROM dbo.OrderDetails
      GROUP BY orderid) AS P
WHERE P2 = 1 AND P3 = 1 AND P4 = 1;
```

And here's the query you would use in SQL Server 2005:

```
SELECT orderid
FROM (SELECT *
      FROM dbo.OrderDetails
        PIVOT(COUNT(productid) FOR productid IN([2],[3],[4])) AS P) AS T
WHERE [2] = 1 AND [3] = 1 AND [4] = 1;
```

Aggregating Data

You can also use a pivoting technique to format aggregated data, typically for reporting purposes. In my examples, I'll use the Orders table, which you create and populate by running the code in Listing 6-4.

Listing 6-4 Creating and populating the Orders table

```
USE tempdb;
GO

IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
GO

CREATE TABLE dbo.Orders
(
  orderid   int        NOT NULL PRIMARY KEY NONCLUSTERED,
  orderdate datetime   NOT NULL,
  empid     int        NOT NULL,
  custid    varchar(5) NOT NULL,
  qty       int        NOT NULL
);

CREATE UNIQUE CLUSTERED INDEX idx_orderdate_orderid
  ON dbo.Orders(orderdate, orderid);

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30001, '20020802', 3, 'A', 10);
```

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(10001, '20021224', 1, 'A', 12);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(10005, '20021224', 1, 'B', 20);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(40001, '20030109', 4, 'A', 40);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(10006, '20030118', 1, 'C', 14);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(20001, '20030212', 2, 'B', 12);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(40005, '20040212', 4, 'A', 10);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(20002, '20040216', 2, 'C', 20);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30003, '20040418', 3, 'B', 15);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30004, '20020418', 3, 'C', 22);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30007, '20020907', 3, 'D', 30);
```

The contents of the Orders table are shown in Table 6-13.

Table 6-13 Contents of Orders Table

orderid	*orderdate*	*empid*	*custid*	*qty*
30004	2002-04-18 00:00:00.000	3	C	22
30001	2002-08-02 00:00:00.000	3	A	10
30007	2002-09-07 00:00:00.000	3	D	30
10001	2002-12-24 00:00:00.000	1	A	12
10005	2002-12-24 00:00:00.000	1	B	20
40001	2003-01-09 00:00:00.000	4	A	40
10006	2003-01-18 00:00:00.000	1	C	14
20001	2003-02-12 00:00:00.000	2	B	12
40005	2004-02-12 00:00:00.000	4	A	10
20002	2004-02-16 00:00:00.000	2	C	20
30003	2004-04-18 00:00:00.000	3	B	15

Suppose you want to return a row for each customer, with the total yearly quantities in a different column for each year. You use a pivoting technique very similar to the previous ones I showed, only this time instead of using a MAX, you use a SUM aggregate, which will return the output shown in Table 6-14:

```
SELECT custid,
  SUM(CASE WHEN orderyear = 2002 THEN qty END) AS [2002],
  SUM(CASE WHEN orderyear = 2003 THEN qty END) AS [2003],
  SUM(CASE WHEN orderyear = 2004 THEN qty END) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
GROUP BY custid;
```

Table 6-14 Total Yearly Quantities per Customer

custid	*2002*	*2003*	*2004*
A	22	40	10
B	20	12	15
C	22	14	20
D	30	NULL	NULL

Here you can see the use of a derived table to isolate only the relevant elements for the pivoting activity (*custid*, *orderyear*, *qty*).

One of the main issues with this pivoting solution is that you might end up with lengthy query strings when the number of elements you need to rotate is large. In an effort to shorten the query string, you can use a matrix table that contains a column and a row for each attribute that you need to rotate (*orderyear*, in this case). Only column values in the intersections of corresponding rows and columns contain the value 1, and the other column values are populated with a NULL or a 0, depending on your needs. Run the code in Listing 6-5 to create and populate the Matrix table.

Listing 6-5 Creating and populating the Matrix table

```
USE tempdb;
GO

IF OBJECTPROPERTY(OBJECT_ID('dbo.Matrix'), 'IsUserTable') = 1
  DROP TABLE dbo.Matrix;
GO

CREATE TABLE dbo.Matrix
(
  orderyear INT NOT NULL PRIMARY KEY,
  y2002 INT NULL,
  y2003 INT NULL,
  y2004 INT NULL
);

INSERT INTO dbo.Matrix(orderyear, y2002) VALUES(2002, 1);
INSERT INTO dbo.Matrix(orderyear, y2003) VALUES(2003, 1);
INSERT INTO dbo.Matrix(orderyear, y2004) VALUES(2004, 1);
```

The contents of the Matrix table are shown in Table 6-15.

Table 6-15 Contents of Matrix Table

orderyear	*y2002*	*y2003*	*y2004*
2002	1	NULL	NULL
2003	NULL	1	NULL
2004	NULL	NULL	1

You join the base table (or table expression) with the Matrix table based on a match in *orderyear*. This means that each row from the base table will be matched with one row from Matrix—the one with the same *orderyear*. In that row, only the corresponding *orderyear*'s column value will contain a 1. So you can substitute the expression

```
SUM(CASE WHEN orderyear = <some_year> THEN qty END) AS [<some_year>]
```

with the logically equivalent expression

```
SUM(qty*y<some_year>) AS [<some_year>]
```

Here's what the full query looks like:

```
SELECT custid,
  SUM(qty*y2002) AS [2002],
  SUM(qty*y2003) AS [2003],
  SUM(qty*y2004) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  JOIN dbo.Matrix AS M ON D.orderyear = M.orderyear
GROUP BY custid;
```

If you need the number of orders instead of the sum of *qty*, in the original solution you produce a 1 instead of the *qty* column for each order, and use the COUNT aggregate function, which will produce the output shown in Table 6-16:

```
SELECT custid,
  COUNT(CASE WHEN orderyear = 2002 THEN 1 END) AS [2002],
  COUNT(CASE WHEN orderyear = 2003 THEN 1 END) AS [2003],
  COUNT(CASE WHEN orderyear = 2004 THEN 1 END) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
GROUP BY custid;
```

Table 6-16 Count of Yearly Quantities per Customer

custid	*2002*	*2003*	*2004*
A	2	1	1
B	1	1	1
C	1	1	1
D	1	0	0

With the Matrix table, simply specify the column corresponding to the target year:

```
SELECT custid,
  COUNT(y2002) AS [2002],
  COUNT(y2003) AS [2003],
  COUNT(y2004) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
  JOIN dbo.Matrix AS M ON D.orderyear = M.orderyear
GROUP BY custid;
```

Of course, using the PIVOT operator in SQL Server 2005, the query strings are short to begin with. Here's the query using the PIVOT operator to calculate total yearly quantities per customer:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2002],[2003],[2004])) AS P;
```

And here's a query that counts the orders:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
  PIVOT(COUNT(orderyear) FOR orderyear IN([2002],[2003],[2004])) AS P;
```

Remember that static queries performing pivoting require you to know ahead of time the list of attributes you're going to rotate. For dynamic pivoting, you need to construct the query string dynamically.

Unpivoting

Unpivoting is the opposite of pivoting—namely, rotating columns to rows. Unpivoting is usually used to normalize data, but it has other applications as well.

Note Unpivoting is not an exact inverse of pivoting, as it won't necessarily allow you to regenerate source rows that were pivoted. However, for the sake of simplicity, think of it as the opposite of pivoting.

In my examples, I'll use the PvtCustOrders table, which you create and populate by running the code in Listing 6-6.

Listing 6-6 Creating and populating the PvtCustOrders table

```
USE tempdb;
GO
IF OBJECT_ID('dbo.PvtCustOrders') IS NOT NULL
  DROP TABLE dbo.PvtCustOrders;
GO

SELECT *
INTO dbo.PvtCustOrders
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2002],[2003],[2004])) AS P;
```

The contents of the PvtCustOrders table are shown in Table 6-17.

Table 6-17 Contents of PvtCustOrders Table

custid	*2002*	*2003*	*2004*
A	22	40	10
B	20	12	15
C	22	14	20
D	30	NULL	NULL

The goal in this case will be to generate a result row for each customer and year, containing the customer ID (*custid*), order year (*orderyear*), and quantity (*qty*).

I'll start with a solution that applies to versions earlier than SQL Server 2005. Here as well, try to think in terms of query logical processing as described in Chapter 1.

The first and most important step in the solution is to generate three copies of each base row—one for each year. This can be achieved by performing a cross join between the base table and a virtual auxiliary table that has one row per year. The SELECT list can then return the *custid* and *orderyear*, and also calculate the target year's *qty* with the following CASE expression:

```
CASE orderyear
  WHEN 2002 THEN [2002]
  WHEN 2003 THEN [2003]
  WHEN 2004 THEN [2004]
END AS qty
```

You achieve unpivoting this way, but you'll also get rows corresponding to NULL values in the source table (for example, for customer D in years 2003 and 2004). To eliminate those rows, create a derived table out of the solution query and, in the outer query, eliminate the rows with the NULL in the *qty* column.

Note In practice, you'd typically store a *0* and not a NULL as the quantity for a customer with no orders in a certain year; the order quantity is known to be zero, and not unknown. However, I used NULLs here to demonstrate the treatment of NULLs, which is a very common need in unpivoting problems.

Here's the complete solution, which returns the desired output as shown in Table 6-18:

```
SELECT custid, orderyear, qty
FROM (SELECT custid, orderyear,
        CASE orderyear
          WHEN 2002 THEN [2002]
          WHEN 2003 THEN [2003]
          WHEN 2004 THEN [2004]
        END AS qty
```

```
      FROM dbo.PvtCustOrders,
        (SELECT 2002 AS orderyear
         UNION ALL SELECT 2003
         UNION ALL SELECT 2004) AS OrderYears) AS D
WHERE qty IS NOT NULL;
```

Table 6-18 Unpivoted Total Quantities per Customer and Order Year

custid	*orderyear*	*qty*
A	2002	22
B	2002	20
C	2002	22
D	2002	30
A	2003	40
B	2003	12
C	2003	14
A	2004	10
B	2004	15
C	2004	20

In SQL Server 2005, things are dramatically simpler. You use the UNPIVOT table operator as follows:

```
SELECT custid, orderyear, qty
FROM dbo.PvtCustOrders
  UNPIVOT(qty FOR orderyear IN([2002],[2003],[2004])) AS U
```

Unlike the PIVOT operator, I find the UNPIVOT operator simple and intuitive, and obviously it requires significantly less code. UNPIVOT's first input is the target column name to hold the rotated attribute values (*qty*). Then, following the FOR keyword, you specify the target column name to hold the rotated column names (*orderyear*). Finally, in the parentheses of the IN clause, you specify the source column names that you want to rotate (*[2002],[2003],[2004]*).

Tip All source attributes that are unpivoted must share the same datatype. If you want to unpivot attributes defined with different datatypes, create a derived table or CTE where you first convert all those attributes to SQL_VARIANT. The target column that will hold unpivoted values will also be defined as SQL_VARIANT, and within that column, the values will preserve their original types.

Note Like PIVOT, UNPIVOT requires a static list of column names to be rotated.

Custom Aggregations

Custom aggregations are aggregations that are not provided as built-in aggregate functions—for example, concatenating strings, calculating products, performing bitwise manipulations, calculating medians, and many others. In this section, I'll provide solutions to several custom aggregate requests. Some techniques that I'll cover are generic—in the sense that you can use similar logic for other aggregate requests—while others are specific to one kind of aggregate request.

More Info One of the generic custom aggregate techniques uses cursors. For details about cursors, including handling of custom aggregates with cursors, please refer to *Inside Microsoft SQL Server 2005: T-SQL Programming*.

In my examples, I'll use the generic Groups table, which you create and populate by running the code in Listing 6-7.

Listing 6-7 Creating and populating the Groups table

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Groups') IS NOT NULL
  DROP TABLE dbo.Groups;
GO

CREATE TABLE dbo.Groups
(
  groupid  VARCHAR(10) NOT NULL,
  memberid INT         NOT NULL,
  string   VARCHAR(10) NOT NULL,
  val      INT         NOT NULL,
  PRIMARY KEY (groupid, memberid)
);

INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 3, 'stra1', 6);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 9, 'stra2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 2, 'strb1', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 4, 'strb2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 5, 'strb3', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 9, 'strb4', 11);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 3, 'strc1', 8);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 7, 'strc2', 10);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 9, 'strc3', 12);
```

The contents of the Groups table are shown in Table 6-19.

Table 6-19 Contents of Groups Table

groupid	*memberid*	*string*	*val*
a	3	stra1	6
a	9	stra2	7
b	2	strb1	3
b	4	strb2	7
b	5	strb3	3
b	9	strb4	11
c	3	strc1	8
c	7	strc2	10
c	9	strc3	12

The Groups table has a column representing the group (*groupid*), a column representing a unique identifier within the group (*memberid*), and some value columns (*string* and *val*) that will need to be aggregated. I like to use such a generic form of data because it allows you to focus on the techniques and not on the data. Note that this is merely a generic form of a table containing data that you want to aggregate. For example, it could represent a Sales table where *groupid* stands for *empid*, *val* stands for *qty*, and so on.

Custom Aggregations Using Pivoting

One key technique for solving custom aggregate problems is pivoting. You basically pivot the values that need to participate in the aggregate calculation; when they all appear in the same result row, you perform the calculation as a linear one across the columns. For two reasons, this pivoting technique is limited to situations where there is a small number of elements per group. First, with a large number of elements you'll end up with very lengthy query strings, which is not realistic. Second, unless you have a sequencing column within the group, you'll need to calculate row numbers that will be used to identify the position of elements within the group. For example, if you need to concatenate all values from the *string* column per group, what will you specify as the pivoted attribute list? The values in the *memberid* column are not known ahead of time, plus they differ in each group. Row numbers representing positions within the group solve your problem. Remember that in versions prior to SQL Server 2005, the calculation of row numbers is expensive for large groups.

String Concatenation Using Pivoting

As the first example, the following query calculates an aggregate string concatenation over the column *string* for each group with a pivoting technique, which generates the output shown in Table 6-20:

```
SELECT groupid,
    MAX(CASE WHEN rn = 1 THEN string ELSE '' END)
  + MAX(CASE WHEN rn = 2 THEN ',' + string ELSE '' END)
  + MAX(CASE WHEN rn = 3 THEN ',' + string ELSE '' END)
  + MAX(CASE WHEN rn = 4 THEN ',' + string ELSE '' END) AS string
```

```
FROM (SELECT groupid, string,
        (SELECT COUNT(*)
         FROM dbo.Groups AS B
         WHERE B.groupid = A.groupid
           AND B.memberid <= A.memberid) AS rn
      FROM dbo.Groups AS A) AS D
GROUP BY groupid;
```

Table 6-20 Concatenated Strings

groupid	*string*
a	stra1,stra2
b	strb1,strb2,strb3,strb4
c	strc1,strc2,strc3

The query that generates the derived table D calculates a row number within the group based on *memberid* order. The outer query pivots the values based on the row numbers, and it performs linear concatenation. I'm assuming here that there are at most four rows per group, so I specified four MAX(CASE...) expressions. You need as many MAX(CASE...) expressions as the maximum number of elements you anticipate.

Note It's important to return an empty string rather than a NULL in the ELSE clause of the CASE expressions. Remember that a concatenation between a known value and a NULL yields a NULL.

Aggregate Product Using Pivoting

In a similar manner, you can calculate the product of the values in the *val* column for each group, yielding the output shown in Table 6-21:

```
SELECT groupid,
    MAX(CASE WHEN rn = 1 THEN val ELSE 1 END)
  * MAX(CASE WHEN rn = 2 THEN val ELSE 1 END)
  * MAX(CASE WHEN rn = 3 THEN val ELSE 1 END)
  * MAX(CASE WHEN rn = 4 THEN val ELSE 1 END) AS product
FROM (SELECT groupid, val,
        (SELECT COUNT(*)
         FROM dbo.Groups AS B
         WHERE B.groupid = A.groupid
           AND B.memberid <= A.memberid) AS rn
      FROM dbo.Groups AS A) AS D
GROUP BY groupid;
```

Table 6-21 Aggregate Product

groupid	*product*
a	42
b	693
c	960

The need for an aggregate product is common in financial applications—for example, to calculate compound interest rates.

User Defined Aggregates (UDA)

SQL Server 2005 introduces the ability to create your own user-defined aggregates (UDA). You write UDAs in a .NET language of your choice (for example, C# or Microsoft Visual Basic .NET), and you use them in T-SQL. This book is dedicated to T-SQL and not to common language runtime (CLR), so it won't conduct lengthy discussions explaining CLR UDAs. Rather, you'll be provided with a couple of examples with step-by-step instructions and, of course, the T-SQL interfaces involved. Examples will be provided in both C# and Visual Basic.

CLR Code in a Database

This section discusses .NET common language runtime (CLR) integration in SQL Server 2005; therefore, it's appropriate to spend a couple of words explaining the reasoning behind CLR integration in a database. It is also important to identify the scenarios where using CLR objects is more appropriate than using T-SQL.

Developing in .NET languages such as C# and Visual Basic .NET gives you an incredibly rich programming model. The .NET Framework includes literally thousands of prepared classes, and it is up to you to make astute use of them. .NET languages are not just data-oriented like SQL, so you are not as limited. For example, regular expressions are extremely useful for validating data, and they are fully supported in .NET. SQL languages are set-oriented and slow to perform row-oriented (row-by-row or one-row-at-a-time) operations. Sometimes you need row-oriented operations inside the database; moving away from cursors to CLR code should improve the performance. Another benefit of CLR code is that it can be much faster than T-SQL code in computationally intensive calculations.

Although SQL Server supported programmatic extensions even before CLR integration was introduced, CLR integration in .NET code is superior in a number of ways.

For example, you could add functionality to earlier versions of SQL Server using extended stored procedures. However, such procedures can compromise the integrity of SQL Server processes because their memory and thread management is not integrated well enough with SQL Server's resource management. .NET code is managed by the CLR inside SQL Server, and because the CLR itself is managed by SQL Server, it is much safer to use than extended procedure code.

T-SQL—a set-oriented language—was designed mainly to deal with data and is optimized for data manipulation. You should not rush to translate all your T-SQL code to CLR code. T-SQL is still SQL Server's primary language. Data access can be achieved through T-SQL only. If an operation can be expressed as a set-oriented one, you should program it in T-SQL.

There's another important decision that you need to make before you start using CLR code inside SQL Server. You need to decide where your CLR code is going to run—at the server or at the client. CLR code is typically faster and more flexible than T-SQL for

computations, and thus it extends the opportunities for server-side computations. However, the server side is typically a single working box, and load balancing at the data tier is still in its infancy. Therefore, you should consider whether it would be more sensible to process those computations at the client side.

With CLR code, you can write stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregate functions. The last two objects can't be written with declarative T-SQL; rather, they can be written only with CLR code. A User-Defined Type (UDT) is the most complex CLR object type and demands extensive coverage.

More Info For details about programming CLR UDTs, as well as programming CLR routines, please refer to *Inside Microsoft SQL Server 2005: T-SQL Programming*.

Let's start with a concrete implementation of two UDAs. The steps involved in creating a CLR-based UDA are as follows:

- Define the UDA as a class in a .NET language.
- Compile the class you defined to build a CLR assembly.
- Register the assembly in SQL Server using the CREATE ASSEMBLY command.
- Use the CREATE AGGREGATE command in T-SQL to create the UDA that references the registered assembly.

Note You can register an assembly and create a CLR object from Microsoft Visual Studio 2005 directly, using the project deployment option (Build>Deploy menu item). This section will show you how to deploy CLR objects directly from Visual Studio. Also be aware that direct deployment from Visual Studio is supported only with the Professional edition or higher; if you're using the Standard edition, your only option is explicit deployment in SQL Server.

This section will provide examples for creating aggregate string concatenation and aggregate product functions in both C# and Visual Basic .NET. You can find the code for the C# classes in Listing 6-8 and the code for the Visual Basic .NET classes in Listing 6-9. You'll be provided with the requirements for a CLR UDA alongside the development of a UDA.

Listing 6-8 C# UDAs Code

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
using System.IO;
using System.Runtime.InteropServices;
```

```
[Serializable]
[SqlUserDefinedAggregate(
   Format.UserDefined,              // use user-defined serialization
   IsInvariantToDuplicates = false, // duplicates make difference
    // for the result
   IsInvariantToNulls = true,       // don't care about NULLs
   IsInvariantToOrder = false,      // whether order makes difference
   IsNullIfEmpty = false,           // do not yield a NULL
    // for a set of zero strings
   MaxByteSize = 8000)]             // maximum size in bytes of persisted value
public struct CSStrAgg : IBinarySerialize
{
    private StringBuilder sb;
    private bool firstConcat;

    public void Init()
    {
        this.sb = new StringBuilder();
        this.firstConcat = true;
    }

    public void Accumulate(SqlString s)
    {
        if (s.IsNull)
        {
            return;                 // simply skip Nulls approach
        }
        if (this.firstConcat)
        {
            this.sb.Append(s.Value);
            this.firstConcat = false;
        }
        else
        {
            this.sb.Append(",");
            this.sb.Append(s.Value);
        }
    }

    public void Merge(CSStrAgg Group)
    {
        this.sb.Append(Group.sb);
    }

    public SqlString Terminate()
    {
        return new SqlString(this.sb.ToString());
    }

    public void Read(BinaryReader r)
    {
        sb = new StringBuilder(r.ReadString());
    }
```

```
    public void Write(BinaryWriter w)
    {
        if (this.sb.Length > 4000)  // check we don't
                                    // go over 8000 bytes

                                    // simply return first 8000 bytes
            w.Write(this.sb.ToString().Substring(0, 4000));
        else
            w.Write(this.sb.ToString());
    }

}   // end CSStrAgg

[Serializable]
[StructLayout(LayoutKind.Sequential)]
[SqlUserDefinedAggregate(
   Format.Native,                   // use native serialization
   IsInvariantToDuplicates = false, // duplicates make difference
    // for the result
   IsInvariantToNulls = true,       // don't care about NULLs
   IsInvariantToOrder = false)]     // whether order makes difference
public class CSProdAgg
{
    private SqlInt64 si;

    public void Init()
    {
        si = 1;
    }


    public void Accumulate(SqlInt64 v)
    {
        if (v.IsNull || si.IsNull)  // Null input = Null output approach
        {
            si = SqlInt64.Null;
            return;
        }
        if (v == 0 || si == 0)      // to prevent an exception in next if
        {
            si = 0;
            return;
        }
                                    // stop before we reach max value
        if (Math.Abs(v.Value) <= SqlInt64.MaxValue / Math.Abs(si.Value))
        {
            si = si * v;
        }
        else
        {
            si = 0;                 // if we reach too big value, return 0
        }

    }
```

```
    public void Merge(CSProdAgg Group)
    {
        Accumulate(Group.Terminate());
    }

    public SqlInt64 Terminate()
    {
        return (si);
    }

}  // end CSProdAgg
```

Listing 6-9 Visual Basic .NET UDAs Code

```
Imports System
Imports System.Data
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Text
Imports System.IO
Imports System.Runtime.InteropServices


<Serializable(), _
 SqlUserDefinedAggregate( _
               Format.UserDefined, _
               IsInvariantToDuplicates:=True, _
               IsInvariantToNulls:=True, _
               IsInvariantToOrder:=False, _
               IsNullIfEmpty:=False, _
               MaxByteSize:=8000)> _
Public Class VBStrAgg
    Implements IBinarySerialize

    Private sb As StringBuilder
    Private firstConcat As Boolean = True

    Public Sub Init()
        Me.sb = New StringBuilder()
        Me.firstConcat = True
    End Sub


    Public Sub Accumulate(ByVal s As SqlString)
        If s.IsNull Then
            Return
        End If
        If Me.firstConcat = True Then
            Me.sb.Append(s.Value)
            Me.firstConcat = False
        Else
```

```
            Me.sb.Append(",")
            Me.sb.Append(s.Value)
        End If
    End Sub

    Public Sub Merge(ByVal Group As VBStrAgg)
        Me.sb.Append(Group.sb)
    End Sub

    Public Function Terminate() As SqlString
        Return New SqlString(sb.ToString())
    End Function

    Public Sub Read(ByVal r As BinaryReader) _
      Implements IBinarySerialize.Read
        sb = New StringBuilder(r.ReadString())
    End Sub

    Public Sub Write(ByVal w As BinaryWriter) _
      Implements IBinarySerialize.Write
        If Me.sb.Length > 4000 Then
            w.Write(Me.sb.ToString().Substring(0, 4000))
        Else
            w.Write(Me.sb.ToString())
        End If
    End Sub

End Class


<Serializable(), _
 StructLayout(LayoutKind.Sequential), _
 SqlUserDefinedAggregate( _
               Format.Native, _
               IsInvariantToOrder:=False, _
               IsInvariantToNulls:=True, _
               IsInvariantToDuplicates:=True)> _
Public Class VBProdAgg

    Private si As SqlInt64

    Public Sub Init()
        si = 1
    End Sub

    Public Sub Accumulate(ByVal v As SqlInt64)
        If v.IsNull = True Or si.IsNull = True Then
            si = SqlInt64.Null
            Return
        End If
        If v = 0 Or si = 0 Then
            si = 0
            Return
        End If
```

```
        If (Math.Abs(v.Value) <= SqlInt64.MaxValue / Math.Abs(si.Value)) _
          Then
            si = si * v
        Else
            si = 0
        End If
    End Sub

    Public Sub Merge(ByVal Group As VBProdAgg)
        Accumulate(Group.Terminate())
    End Sub

    Public Function Terminate() As SqlInt64
        If si.IsNull = True Then
            Return SqlInt64.Null
        Else
            Return si
        End If
    End Function

End Class
```

Here are the step-by-step instructions you need to follow to create the assemblies in Visual Studio 2005:

Creating an Assembly in Visual Studio 2005

1. In Visual Studio 2005, create a new C# project. Use the Database folder and the SQL Server Project template.

> **Note** This template is not available in Visual Studio 2005, Standard edition. If you're working with the Standard edition, use the Class Library template and manually write all the code.

2. In the New Project dialog box, specify the following information:
   - Name: CSUDAs
   - Location: C:\
   - Solution Name: UDAs

   When you're done entering the information, confirm that it is correct.
3. At this point, you'll be requested to specify a database reference. Create a new database reference to the tempdb database in the SQL Server instance you're working with, and choose it. The database reference you choose tells Visual Studio where to deploy the UDAs that you develop.

4. After confirming the choice of database reference, a question box will pop up asking you whether you want to enable SQL/CLR debugging on this connection. Choose No. The sample UDAs you'll build in this chapter are quite simple, and there won't be a need for debugging.
5. In the Solution Explorer window, right-click the CSUDAs project, select the menu items Add and Aggregate, and then choose the Aggregate template. Rename the class Aggregate1.cs to **CSUDAs_Classes.cs**, and confirm.
6. Examine the code of the template. You'll find that a UDA is implemented as a structure (*struct* in C#, *Structure* in Visual Basic .NET). It can be implemented as a class as well. The first block of code in the template includes namespaces that are used in the assembly (lines of code starting with "using"). Add three more statements to include the following namespaces: *System.Text*, *System.IO*, and *System.Runtime.InteropServices*. (You can copy those from Listing 6-8.) You are going to use the *StringBuilder* class from the *System.Text* namespace, the *BinaryReader* and *BinaryWriter* classes from the *System.IO* namespace, and finally the *StructLayout* attribute from the *System.Runtime.InteropServices* namespace (in the second UDA).
7. Rename the default name of the UDA—which is currently the same name as the name of the class (*CSUDAs_Classes*)—to **CSStrAgg**.
8. You'll find four methods that are already provided by the template. These are the methods that every UDA must implement. However, if you use the Class Library template for your project, you have to write them manually. Using the Aggregate template, all you have to do is fill them with your code. Following is a description of the four methods:
   - *Init*: This method is used to initialize the computation. It is invoked once for each group that the query processor is aggregating.
   - *Accumulate*: The name of the method gives you a hint of its purpose—accumulating the aggregate values, of course. This method is invoked once for each value (that is, for every single row) in the group that is being aggregated. It uses an input parameter, and the parameter has to be of the datatype corresponding to the native SQL Server datatype of the column you are going to aggregate. The datatype of the input can also be a CLR UDT.
   - *Merge*: You'll notice that this method uses an input parameter with the type that is the aggregate class. The method is used to merge multiple partial computations of an aggregation.
   - *Terminate*: This method finishes the aggregation and returns the result.
9. Add two internal (private) variables—*sb* and *firstConcat*—to the class just before the *Init* method. You can do so by simply copying the code that declares them from Listing 6-8. The variable *sb* is of type *StringBuilder* and will hold the intermediate aggregate value. The *firstConcat* variable is of type *Boolean* and is used to tell whether the input string is

the first you are concatenating in the group. For all input values except the first, you are going to add a comma in front of the value you are concatenating.

10. Override the current code for the four methods with the code implementing them from Listing 6-8. Keep in mind the following points for each method:
    - In the *Init* method, you initialize *sb* with an empty string and *firstConcat* with true.
    - In the *Accumulate* method, note that if the value of the parameter is NULL, the accumulated value will be NULL as well. Also, notice the different treatment of the first value, which is just appended, and the following values, which are appended with the addition of a leading comma.
    - In the *Merge* method, you are simply adding a partial aggregation to the current one. You do so by calling the *Accumulate* method of the current aggregation, and adding the termination (final value) of the other partial aggregation. The input of the *Merge* function refers to the class name, which you revised earlier to *CSStrAgg*.
    - The *Terminate* method is very simple as well; it just returns the string representation of the aggregated value.
11. Delete the last two rows of the code in the class from the template; these are a placeholder for a member field. You already defined all member fields you need at the beginning of the UDA.
12. Next, go back to the top of the UDA, right after the inclusion of the namespaces. You'll find attribute names that you want to include. Attributes help Visual Studio in deployment, and they help SQL Server to optimize the usage of the UDA. UDAs have to include the *Serializable* attribute. Serialization in .NET means saving the values of the fields of a class persistently. UDAs need serialization for intermediate results. The format of the serialization can be native, meaning they are left to SQL Server or defined by the user. Serialization can be native if you use only .NET value types; it has to be user-defined if you use .NET reference types. Unfortunately, the *string* type is a reference type in .NET. Therefore, you have to prepare your own serialization. You have to implement the *IBinarySerialize* interface, which defines just two methods: *Read* and *Write*. The implementation of these methods in our UDA is very simple. The *Read* method uses the *ReadString* method of the *StringBuilder* class. The *Write* method uses the default *ToString* method. The *ToString* method is inherited by all .NET classes from the topmost class, called *System.Object*.

    Continue implementing the UDA by following these steps:

    a. Specify that you are going to implement the *IBinarySerialize* interface in the structure. You do so by adding a colon and the name of the interface right after the name of the structure (the UDA name).

    b. Copy the *Read* and *Write* methods from Listing 6-8 to the end of your UDA.

c. Change the *Format.Native* property of the *SqlUserDefinedAggregate* attribute to **Format.UserDefined**. With user-defined serialization, your aggregate is limited to 8000 bytes only. You have to specify how many bytes your UDA can return at maximum with the *MaxByteSize* property of the *SqlUserDefinedAggregate* attribute. To get the maximum possible string length, specify *MaxByteSize = 8000*.

13. You'll find some other interesting properties of the *SqlUserDefinedAggregate* attribute in Listing 6-8. Let's explore them:

- *IsInvariantToDuplicates*: This is an optional property. For example, the MAX aggregate is invariant to duplicates, while SUM is not.
- *IsInvariantToNulls*: This is another optional property. It specifies whether the aggregate is invariant to NULLs.
- *IsInvariantToOrder*: This property is reserved for future use. It is currently ignored by the query processor. Therefore, order is currently not guaranteed.
- *IsNullIfEmpty*: This property indicates whether the aggregate will return a NULL if no values have been accumulated.

14. Add the aforementioned properties to your UDA by copying them from Listing 6-8. Your first UDA is now complete!

15. Listing 6-8 also has the code to implement a product UDA (*CSProdAgg*). Copy the complete code implementing *CSProgAgg* to your script. Note that this UDA involves handling of big integers only. Because the UDA internally deals only with value types, it can use native serialization. Native serialization requires that the *StructLayoutAttribute* be specified as *StructLayout.LayoutKindSequential* if the UDA is defined in a class and not a structure. Otherwise, the UDA implements the same four methods as your previous UDA. There is an additional check in the *Accumulate* method that prevents out-of-range values.

16. Finally, add the Visual Basic .NET version of both UDAs created so far:

a. From the File menu, choose the menu items Add and New Project to load the Add New Project dialog box. Navigate through the Visual Basic project type and the Database folder, and choose SQL Server Project. Don't confirm yet.

b. In the Add New Project dialog box, specify Name as **VBUDAs** and Location as **C:**. Then confirm that the information is correct.

c. Use the same database connection you created for the C# project (the connection to tempdb). The name of the database connection you created earlier should be *instancename.tempdb.dbo*.

d. In the Solution Explorer window, right-click the VBUDAs project, select Add, and choose the Aggregate template. Before confirming, rename the class Aggregate1.vb to **VBUDAs_Classes.vb**.

e. Replace all code in *VBUDAs_Classes.vb* with the Visual Basic .NET code implementing the UDAs from Listing 6-9.

17. Save all files by choosing the File menu item and then Save All.
18. Create the assemblies by building the solution. You do this by choosing the Build menu item and then Build Solution.
19. Finally, deploy the solution by choosing the Build menu item and then Deploy Solution.

Both assemblies should be cataloged at this point, and all four UDAs should be created. All these steps are done if you deploy the assembly from Visual Studio .NET.

Note To work with CLR-based functions in SQL Server, you need to enable the server configuration option 'clr enabled' (which is disabled by default).

You can check whether the deployment was successful by browsing the *sys.assemblies* and *sys.assembly_modules* catalog views, which are in the tempdb database in our case. To enable CLR and query these views, run the code in Listing 6-10.

Listing 6-10 Enabling CLR and querying catalog views

```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE WITH OVERRIDE;
GO
USE tempdb;
GO
SELECT * FROM sys.assemblies;
SELECT * FROM sys.assembly_modules;
```

That's basically it. You use UDAs just like you use any other built-in aggregate function. To test the new functions, run the following code, and you'll get the same results returned by the other solutions to custom aggregates I presented earlier.

Testing UDAs

```
SELECT groupid, dbo.CSStrAgg(string) AS string
FROM tempdb.dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.VBStrAgg(string) AS string
FROM tempdb.dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.CSProdAgg(val) AS product
FROM tempdb.dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.VBProdAgg(val) AS product
FROM tempdb.dbo.Groups
GROUP BY groupid;
```

When you're done experimenting with the UDAs, run the following code to disable CLR support:

```
EXEC sp_configure 'clr enabled', 0;
RECONFIGURE WITH OVERRIDE;
```

Specialized Solutions

Another type of solution for custom aggregates is developing a specialized, optimized solution for each aggregate. The advantage is usually the improved performance of the solution. The disadvantage is that you probably won't be able to use similar logic for other aggregate calculations.

Specialized Solution for Aggregate String Concatenation

A specialized solution for aggregate string concatenation uses the PATH mode of the FOR XML query option. This beautiful (and extremely fast) technique was devised by Michael Rys, a program manager with the Microsoft SQL Server development team in charge of SQL Server XML technologies, and Eugene Kogan, a technical lead on the Microsoft SQL Server Engine team. The PATH mode provides an easier way to mix elements and attributes than the EXPLICIT directive. Here's the specialized solution for aggregate string concatenation:

```
SELECT groupid,
  STUFF((SELECT ',' + string AS [text()]
         FROM dbo.Groups AS G2
         WHERE G2.groupid = G1.groupid
         ORDER BY memberid
         FOR XML PATH('')), 1, 1, '') AS string
FROM dbo.Groups AS G1
GROUP BY groupid;
```

The subquery basically returns an ordered path of all strings within the current group. Because an empty string is provided to the PATH clause as input, a wrapper element is not generated. An expression with no alias (for example, *',' + string*) or one aliased as *[text()]* is inlined, and its contents are inserted as a text node. The purpose of the STUFF function is simply to remove the first comma (by substituting it with an empty string).

Specialized Solution for Aggregate Product

Keep in mind that to calculate an aggregate product you have to scan all values in the group. So the performance potential your solution can reach is to achieve the calculation by scanning the data only once, using a set-based query. In the case of an aggregate product, this can be achieved using mathematical manipulation based on logarithms. I'll rely on the following logarithmic equations:

Equation 1: $\log_a(b) = x$ if and only if $a^x = b$

Equation 2: $\log_a(v1 * v2 * \ldots * vn) = \log_a(v1) + \log_a(v2) + \ldots + \log_a(vn)$

Basically, what you're going to do here is a transformation of calculations. You have support in T-SQL for LOG, POWER, and SUM functions. Using those, you can generate the missing product. Group the data by the *groupid* column, as you would with any built-in aggregate. The expression *SUM(LOG10(val))* corresponds to the right side of Equation 2, where the base *a* is equal to 10 in our case, because you used the LOG10 function. To get the product of the elements, all you have left to do is raise the base (10) to the power of the right side of the equation. In other words, the expression *POWER(10., SUM(LOG10(val)))* gives you the product of elements within the group. Here's what the full query looks like:

```
SELECT groupid, POWER(10., SUM(LOG10(val))) AS product
FROM dbo.Groups
GROUP BY groupid;
```

This is the final solution if you're dealing only with positive values. However, the logarithm function is undefined for zero and negative numbers. You can use pivoting techniques to identify and deal with zeros and negatives as follows:

```
SELECT groupid,
  CASE
    WHEN MAX(CASE WHEN val = 0 THEN 1 END) = 1 THEN 0
    ELSE
      CASE WHEN COUNT(CASE WHEN val < 0 THEN 1 END) % 2 = 0
        THEN 1 ELSE -1
      END * POWER(10., SUM(LOG10(NULLIF(ABS(val), 0))))
  END AS product
FROM dbo.Groups
GROUP BY groupid;
```

The outer CASE expression first uses a pivoting technique to check whether a 0 value appears in the group, in which case it returns a 0 as the result. The ELSE clause invokes another CASE expression, which also uses a pivoting technique to count the number of negative values in the group. If that number is even, it produces a +1; if it's odd, it produces a −1. The purpose of this calculation is to determine the numerical sign of the result. The sign (−1 or +1) is then multiplied by the product of the absolute values of the numbers in the group to give the desired product.

Note that NULLIF is used here to substitute zeros with NULLs. You might expect this part of the expression not to be evaluated at all if a zero is found. But remember that the optimizer can consider many different physical plans to execute your query. As a result, you can't be certain of the actual order in which parts of an expression will be evaluated. By substituting zeros with NULLs, you ensure that you'll never get a domain error if the LOG10 function ends up being invoked with a zero as an input. This use of NULLIF, together with the use of ABS, allow this solution to accommodate inputs of any sign (negative, zero, and positive).

You could also use a pure mathematical approach to handle zeros and negative values using the following query:

```
SELECT groupid,
  CAST(ROUND(EXP(SUM(LOG(ABS(NULLIF(val,0))))*
    (1-SUM(1-SIGN(val))%4)*(1-SUM(1-SQUARE(SIGN(val)))),0) AS INT)
 AS product
FROM dbo.Groups
GROUP BY groupid;
```

This example shows that you should never lose hope when searching for an efficient solution. If you invest the time and think outside the box, in most cases you'll find a solution.

Specialized Solutions for Aggregate Bitwise Operations

Next, I'll introduce specialized solutions for aggregating the T-SQL bitwise operations—bitwise OR (|), bitwise AND (&), and bitwise XOR (^). I'll assume that you're familiar with the basics of bitwise operators and their uses, and provide only a brief overview. If you're not, please refer first to the section "Bitwise Operators" in Books Online.

Bitwise operations are operations performed on the individual bits of integer data. Each bit has two possible values, 1 and 0. Integers can be used to store *bitmaps* or strings of bits, and in fact they are used internally by SQL Server to store metadata information—for example, properties of indexes (clustered, unique, and so on) and properties of databases (read only, restrict access, auto shrink, and so on). You might also choose to store bitmaps yourself to represent sets of binary attributes—for example, a set of permissions where each bit represents a different permission.

Some experts advise against using such a design because it violates 1NF (first normal form—no repeating groups). You might well prefer to design your data in a more normalized form, where attributes like this are stored in separate columns. I don't want to get into a debate about which design is better. Here I'll assume a given design that does store bitmaps with sets of flags, and I'll assume that you need to perform aggregate bitwise activities on these bitmaps. I just want to introduce the techniques for cases where you do find the need to use them.

Bitwise OR (|) is usually used to construct bitmaps or to generate a result bitmap that accumulates all bits that are turned on. In the result of bitwise OR, bits are turned on (that is, have value 1) if they are turned on in at least one of the separate bitmaps.

Bitwise AND (&) is usually used to check whether a certain bit (or a set of bits) are turned on by ANDing the source bitmap and a mask. It's also used to accumulate only bits that are turned on in all bitmaps. It generates a result bit that is turned on if that bit is turned on in all the individual bitmaps.

Bitwise XOR (^) is usually used to calculate parity or as part of a scheme to encrypt data. For each bit position, the result bit is turned on if it is on in an odd number of the individual bitmaps.

Note Bitwise XOR is the only bitwise operator that is reversible. That's why it's used for parity calculations and encryption.

Aggregate versions of the bitwise operators are not provided in SQL Server, and I'll provide solutions here to perform aggregate bitwise operations. I'll use the same Groups table that I used in my other custom aggregate examples. Assume that the integer column *val* represents a bitmap. To see the bit representation of each integer, first create the function *fn_dectobase* by running the code in Listing 6-11.

Listing 6-11 Creation script for the *fn_dectobase* function

```
IF OBJECT_ID('dbo.fn_dectobase') IS NOT NULL
  DROP FUNCTION dbo.fn_dectobase;
GO
CREATE FUNCTION dbo.fn_dectobase(@val AS BIGINT, @base AS INT)
  RETURNS VARCHAR(63)
AS
BEGIN
  IF @val < 0 OR @base < 2 OR @base > 36 RETURN NULL;
  DECLARE @r AS VARCHAR(63), @alldigits AS VARCHAR(36);

  SET @alldigits = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';

  SET @r = '';
  WHILE @val > 0
  BEGIN
    SET @r = SUBSTRING(@alldigits, @val % @base + 1, 1) + @r;
    SET @val = @val / @base;
  END

  RETURN @r;
END
GO
```

The function accepts two inputs: a 64-bit integer holding the source bitmap, and a base in which you want to represent the data. Use the following query to return the bit representation of the integers in the *val* column of Groups. An abbreviated form of the result (only the 10 rightmost digits of *binval*) is shown in Table 6-22:

```
SELECT groupid, val,
  RIGHT(REPLICATE('0', 32) + CAST(dbo.fn_dectobase(val, 2) AS VARCHAR(64)),
        32) AS binval
FROM dbo.Groups;
```

Table 6-22 Binary Representation of Values

groupid	*val*	*binval*
a	6	0000000110
a	7	0000000111
b	3	0000000011
b	7	0000000111
b	3	0000000011

Table 6-22 Binary Representation of Values

groupid	*val*	*binval*
b	11	0000001011
c	8	0000001000
c	10	0000001010
c	12	0000001100

The *binval* column shows the *val* column in base 2 representation, with leading zeros to create a string with a fixed number of digits. Of course, you can adjust the number of leading zeros according to your needs. In my code samples, I did not incorporate the invocation of this function to avoid distracting you from the techniques I want to focus on. But I did invoke it to generate the bit representations in all the outputs that I'll show.

Aggregate Bitwise OR With no further ado, let's start with calculating an aggregate bitwise OR. To give tangible context to the problem, imagine that you're maintaining application security in the database. The *groupid* column represents a user, and the *val* column represents a bitmap with permission states (either 1 for granted or 0 for not granted) of a role the user is a member of. You're after the effective permissions bitmap for each user (group), which should be calculated as the aggregate bitwise OR between all bitmaps of roles the user is a member of.

The main aspect of a bitwise OR operation that I'll rely on in my solutions is the fact that it's equivalent to the arithmetic sum of the values represented by each distinct bit value that is turned on in the individual bitmaps. Within an integer, a bit represents the value *2^(bit_pos-1)*. For example, the bit value of the third bit is *2^2 = 4*. Take for example the bitmaps for user c: 8 (1000), 10 (1010), and 12 (1100). The bitmap 8 has only one bit turned on—the bit value representing 8, 10 has the bits representing 8 and 2 turned on, and 12 has the 8 and 4 bits turned on. The distinct bits turned on in any of the integers 8, 10, and 12 are the 2, 4, and 8 bits, so the aggregate bitwise OR of 8, 10, and 12 is equal to *2 + 4 + 8 = 14 (1110)*.

The following solution relies on the aforementioned logic by extracting the individual bit values that are turned on in any of the participating bitmaps. The extraction is achieved using the expression *MAX(val & <bitval>)*. The query then performs an arithmetic sum of the individual bit values:

```
SELECT groupid,
    MAX(val & 1)
  + MAX(val & 2)
  + MAX(val & 4)
  + MAX(val & 8)
-- ...
  + MAX(val & 1073741824) AS agg_or
FROM dbo.Groups
GROUP BY groupid;
```

The result of the aggregate bitwise OR operation is shown in Table 6-23, including the 10 rightmost digits of the binary representation of the result value.

Table 6-23 Aggregate Bitwise OR

groupid	*agg_or*	*agg_or_binval*
a	7	0000000111
b	15	0000001111
c	14	0000001110

Similarly, you can use *SUM(DISTINCT val & <bitval>)* instead of *MAX(val & <bitval>)*, because the only possible results are *<bitval>* and *0*:

```
SELECT groupid,
    SUM(DISTINCT val & 1)
  + SUM(DISTINCT val & 2)
  + SUM(DISTINCT val & 4)
  + SUM(DISTINCT val & 8)
-- ...
  + SUM(DISTINCT val & 1073741824) AS agg_or
FROM dbo.Groups
GROUP BY groupid;
```

Both solutions suffer from the same limitation—lengthy query strings—because of the need for a different expression for each bit value. In an effort to shorten the query strings, you can use an auxiliary table. You join the Groups table with an auxiliary table that contains all relevant bit values, using *val & bitval = bitval* as the join condition. The result of the join will include all bit values that are turned on in any of the bitmaps. You can then find *SUM(DISTINCT <bitval>)* for each group. The auxiliary table of bit values can be easily generated from the Nums table used earlier. Filter as many numbers as the bits that you might need, and raise 2 to the power *n–1*. Here's the complete solution:

```
SELECT groupid, SUM(DISTINCT bitval) AS agg_or
FROM dbo.Groups
  JOIN (SELECT POWER(2, n-1) AS bitval
        FROM dbo.Nums
        WHERE n <= 31) AS Bits
    ON val & bitval = bitval
GROUP BY groupid;
```

Aggregate Bitwise AND In a similar manner, you can calculate an aggregate bitwise AND. In the permissions scenario, an aggregate bitwise AND would represent the most restrictive permission set. Just keep in mind that a bit value should be added to the arithmetic sum only if it's turned on in all bitmaps. So first group the data by *groupid* and *bitval*, and filter only the groups where *MIN(val & bitval) > 0*, meaning that the bit value was turned on in all bitmaps. In an outer query, group the data by *groupid* and perform the arithmetic sum of the bit values from the inner query:

```
SELECT groupid, SUM(bitval) AS agg_and
FROM (SELECT groupid, bitval
      FROM dbo.Groups,
        (SELECT POWER(2, n-1) AS bitval
         FROM dbo.Nums
         WHERE n <= 31) AS Bits
      GROUP BY groupid, bitval
      HAVING MIN(val & bitval) > 0) AS D
GROUP BY groupid;
```

The result of the aggregate bitwise AND operation is shown in Table 6-24.

Table 6-24 Aggregate Bitwise AND

groupid	*agg_or*	*agg_or_binval*
a	6	0000000110
b	3	0000000011
c	8	0000001000

Aggregate Bitwise XOR To calculate an aggregate bitwise XOR operation, filter only the *groupid, bitval* groups that have an odd number of bits that are turned on as shown in the following code, which illustrates an aggregate bitwise XOR using *Nums* and generates the output shown in Table 6-25:

```
SELECT groupid, SUM(bitval) AS agg_xor
FROM (SELECT groupid, bitval
      FROM dbo.Groups,
        (SELECT POWER(2, n-1) AS bitval
         FROM dbo.Nums
         WHERE n <= 31) AS Bits
      GROUP BY groupid, bitval
      HAVING SUM(SIGN(val & bitval)) % 2 = 1) AS D
GROUP BY groupid;
```

Table 6-25 Aggregate Bitwise XOR

groupid	*agg_or*	*agg_or_binval*
a	1	0000000001
b	12	0000001100
c	14	0000001110

Median

As the last example for a specialized custom aggregate solution, I'll use the statistical median calculation. Suppose that you need to calculate the median of the *val* column for each group. There are two different definitions of median. Here we will return the middle value in case there's an odd number of elements, and the average of the two middle values in case there's an even number of elements.

The following code shows a technique for calculating the median, producing the output shown in Table 6-26:

```
WITH Tiles AS
(
  SELECT groupid, val,
    NTILE(2) OVER(PARTITION BY groupid ORDER BY val) AS tile
  FROM dbo.Groups
),
GroupedTiles AS
(
  SELECT groupid, tile, COUNT(*) AS cnt,
    CASE WHEN tile = 1 THEN MAX(val) ELSE MIN(val) END AS val
  FROM Tiles
  GROUP BY groupid, tile
)
SELECT groupid,
  CASE WHEN MIN(cnt) = MAX(cnt) THEN AVG(1.*val)
       ELSE MIN(val) END AS median
FROM GroupedTiles
GROUP BY groupid;
```

Table 6-26 Median

groupid	*median*
a	6.500000
b	5.000000
c	10.000000

The Tiles CTE calculates the *NTILE(2)* value within the group, based on *val* order. When there's an even number of elements, the first half of the values will get tile number 1 and the second half will get tile number 2. In an even case, the median is supposed to be the average of the highest value within the first tile and the lowest in the second. When there's an odd number of elements, remember that an additional row is added to the first group. This means that the highest value in the first tile is the median.

The second CTE (GroupedTiles) groups the data by group and tile number, returning the row count for each group and tile as well as the *val* column, which for the first tile is the maximum value within the tile and for the second tile the minimum value within the tile.

The outer query groups the two rows in each group (one representing each tile). A CASE expression in the SELECT list determines what to return based on the parity of the group's row count. When the group has an even number of rows (that is, the group's two tiles have the same row count), you get the average of the maximum in the first tile and the minimum in the second. When the group has an odd number of elements (that is, the group's two tiles have different row counts), you get the minimum of the two values, which happens to be the maximum within the first tile, which in turn, happens to be the median.

Using the ROW_NUMBER function, you can come up with additional solutions to finding the median that are more elegant and somewhat simpler. Here's the first example:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER()
      OVER(PARTITION BY groupid ORDER BY val, memberid) AS rna,
    ROW_NUMBER()
      OVER(PARTITION BY groupid ORDER BY val DESC, memberid DESC) AS rnd
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE ABS(rna - rnd) <= 1
GROUP BY groupid;
```

The idea is to calculate two row numbers for each row: one based on *val*, *memberid* (the tiebreaker) in ascending order (*rna*), and the other based on the same attributes in descending order (*rnd*). There's an interesting mathematical relationship between two sequences sorted in opposite directions that you can use to your advantage. The absolute difference between the two is smaller than or equal to 1 only for the elements that need to participate in the median calculation. Take, for example, a group with an odd number of elements; *ABS(rna − rnd)* is equal to 0 only for the middle row. For all other rows, it is greater than 1. Given an even number of elements, the difference is 1 for the two middle rows and greater than 1 for all others.

The reason for using *memberid* as a tiebreaker is to guarantee determinism of the row number calculations. Because you're calculating two different row numbers, you want to make sure that a value that appears at the *n*th position from the beginning in ascending order will appear at the *n*th position from the end in descending order.

Once the values that need to participate in the median calculation are isolated, you just need to group them by *groupid* and calculate the average per group.

You can avoid the need to calculate two separate row numbers by deriving the second from the first. The descending row numbers can be calculated by subtracting the ascending row numbers from the count of rows in the group and adding one. For example, in a group of four elements, the row that got an ascending row number 1, would get the descending row number *4−1+1 = 4*. Ascending row number 2, would get the descending row number *4−2+1 = 3*, and so on. Deriving the descending row number from the ascending one eliminates the need for a tiebreaker. You're not dealing with two separate calculations; therefore, nondeterminism is not an issue anymore.

So the calculation *rna − rnd* becomes the following: *rn − (cnt-rn+1) = 2*rn − cnt − 1*. Here's a query that implements this logic:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY val) AS rn,
    COUNT(*) OVER(PARTITION BY groupid) AS cnt
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE ABS(2*rn - cnt - 1) <= 1
GROUP BY groupid;
```

There's another way to figure out which rows participate in the median calculation based on the row number and the count of rows in the group: *rn IN((cnt+1)/2, (cnt+2)/2)*. For an odd number of elements, both expressions yield the middle row number. For example, if you have 7 rows, both *(7+1)/2* and *(7+2)/2* equal 4. For an even number of elements, the first expression yields the row number just before the middle point and the second yields the row number just after it. If you have 8 rows, *(8+1)/2* yields 4 and *(8+2)/2* yields 5. Here's the query that implements this logic:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY val) AS rn,
    COUNT(*) OVER(PARTITION BY groupid) AS cnt
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE rn IN((cnt+1)/2, (cnt+2)/2)
GROUP BY groupid;
```

Histograms

Histograms are powerful analytical tools that express the distribution of items. For example, suppose you need to figure out from the order information in the Orders table how many small, medium, and large orders you have, based on the order quantities. In other words, you need a histogram with three steps. What defines quantities as small, medium, or large are the extreme quantities (the minimum and maximum quantities). In our Orders table, the minimum order quantity is 10 and the maximum is 40. Take the difference between the two extremes (*40 − 10 = 30*), and divide it by the number of steps (3) to get the step size. In our case, it's 30 divided by 3 is 10. So the boundaries of step 1 (small) would be 10 and 20; for step 2 (medium), they would be 20 and 30; and for step 3 (large), they would be 30 and 40.

To generalize this, let *mn = MIN(qty)* and *mx = MAX(qty)*, and let *stepsize = (mx − mn) / @numsteps*. Given a step number *n*, the lower bound of the step (*lb*) is *mn + (n − 1) * stepsize* and the

higher bound (*hb*) is *mn + n * stepsize*. There's a tricky bit here. What predicate will you use to bracket the elements that belong in a specific step? You can't use *qty BETWEEN lb and hb* because a value that is equal to *hb* will appear in this step, and also in the next step, where it will equal the lower bound. Remember that the same calculation yielded the higher bound of one step and the lower bound of the next step. One approach to deal with this problem is to increase each of the lower bounds by one, so they exceed the previous step's higher bounds. With integers that's fine, but with another data type it won't work because there will be potential values in between two steps, but not inside either one—between the cracks, so to speak.

What I like to do to solve the problem is keep the same value in both bounds, and instead of using BETWEEN I use *qty >= lb* and *qty < hb*. This technique has its own issues, but I find it easier to deal with than the previous technique. The issue here is that the item with the highest quantity (40, in our case) is left out of the histogram. To solve this, I add a very small number to the maximum value before calculating the step size: *stepsize = ((1E0*mx + 0.0000000001) − mn) / @numsteps*. This is a technique that allows the item with the highest value to be included, and the effect on the histogram will otherwise be negligible. I multiplied *mx* by the float value *1E0* to protect against the loss of the upper data point when *qty* is typed as MONEY or SMALLMONEY.

So the ingredients you need to generate the lower and higher bounds of the histogram's steps are these: *@numsteps* (given as input), step number (the *n* column from the Nums auxiliary table), *mn*, and *stepsize*, which I described earlier.

Here's the T-SQL code required to produce the step number, lower bound, and higher bound for each step of the histogram, generating the output shown in Table 6-27:

```
DECLARE @numsteps AS INT;
SET @numsteps = 3;

SELECT n AS step,
  mn + (n - 1) * stepsize AS lb,
  mn + n * stepsize AS hb
FROM dbo.Nums,
  (SELECT MIN(qty) AS mn,
     ((1E0*MAX(qty) + 0.0000000001) - MIN(qty))
     / @numsteps AS stepsize
   FROM dbo.Orders) AS D
WHERE n <= @numsteps;
```

Table 6-27 Histogram Steps Table

Step	*lb*	*hb*
1	10	20.0000000000333
2	20.0000000000333	30.0000000000667
3	30.0000000000667	40.0000000001

You might want to encapsulate this code in a user-defined function to simplify the queries that return the actual histograms. Run the code in Listing 6-12 to do just that.

Listing 6-12 Creation script for *fn_histsteps* function

```
CREATE FUNCTION dbo.fn_histsteps(@numsteps AS INT) RETURNS TABLE
AS
RETURN
  SELECT n AS step,
    mn + (n - 1) * stepsize AS lb,
    mn + n * stepsize AS hb
  FROM dbo.Nums,
    (SELECT MIN(qty) AS mn,
       ((1E0*MAX(qty) + 0.0000000001) - MIN(qty))
       / @numsteps AS stepsize
    FROM dbo.Orders) AS D
  WHERE n <= @numsteps;
GO
```

To test the function, run the following query, which will give you a three-row histogram steps table:

```
SELECT * FROM dbo.fn_histsteps(3) AS S;
```

To return the actual histogram, simply join the steps table and the Orders table on the predicate I described earlier (*qty >= lb AND qty < hb*), group the data by step number, and return the step number and row count:

```
SELECT step, COUNT(*) AS numorders
FROM dbo.fn_histsteps(3) AS S
  JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

This query generates the histogram shown in Table 6-28.

Table 6-28 Histogram with Three Steps

step	*numorders*
1	8
2	2
3	1

You can see that there are eight small orders, two medium orders, and one large order. To return a histogram with ten steps, simply provide 10 as the input to the *fn_histsteps* function, and the query will yield the histogram shown in Table 6-29:

```
SELECT step, COUNT(*) AS numorders
FROM dbo.fn_histsteps(10) AS S
  JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

Table 6-29 Histogram with Ten Steps

step	*numorders*
1	4
2	2
4	3
7	1
10	1

Note that because you're using an inner join, empty steps are not returned. To return empty steps also, you can use the following outer join query, which generates the output shown in Table 6-30:

```
SELECT step, COUNT(qty) AS numorders
FROM dbo.fn_histsteps(10) AS S
  LEFT OUTER JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

Table 6-30 Histogram with Ten Steps, Including Empty Steps

step	*numorders*
1	4
2	2
3	0
4	3
5	0
6	0
7	1
8	0
9	0
10	1

Note Notice that *COUNT(qty)* is used here and not *COUNT(*)*. *COUNT(*)* would incorrectly return 1 for empty steps because there's an outer row in the group. You have to provide the COUNT function an attribute from the nonpreserved side (*Orders*) to get the correct count.

Instead of using an outer join query, you can use a cross join, with a filter that matches orders to steps, and the GROUP BY ALL option which insures that also empty steps will also be returned:

```
SELECT step, COUNT(qty) AS numcusts
FROM dbo.fn_histsteps(10) AS S, dbo.Orders AS O
WHERE qty >= lb AND qty < hb
GROUP BY ALL step;
```

I just wanted to show that you can write a simpler solution using the GROUP BY ALL option. But remember that it is advisable to refrain from using this non standard legacy feature, as it will probably be removed from the product in some future version.

There's another alternative to taking care of the issue with the step boundaries and the predicate used to identify a match. You can simply check whether the step number is 1, in which case you subtract 1 from the lower bound. Then, in the query generating the actual histogram, you use the predicate *qty > lb AND qty <= hb*.

Another approach is to check whether the step is the last, and if it is, add 1 to the higher bound. Then use the predicate *qty >= lb AND qty < hb*.

Listing 6-13 has the revised function implementing the latter approach:

Listing 6-13 Altering the implementation of the *fn_histsteps* function

```
ALTER FUNCTION dbo.fn_histsteps(@numsteps AS INT) RETURNS TABLE
AS
RETURN
  SELECT n AS step,
    mn + (n - 1) * stepsize AS lb,
    mn + n * stepsize + CASE WHEN n = @numsteps THEN 1 ELSE 0 END AS hb
  FROM dbo.Nums,
    (SELECT MIN(qty) AS mn,
       (1E0*MAX(qty) - MIN(qty)) / @numsteps AS stepsize
    FROM dbo.Orders) AS D
  WHERE n < = @numsteps;
GO
```

And the following query generates the actual histogram:

```
SELECT step, COUNT(qty) AS numorders
FROM dbo.fn_histsteps(10) AS S
  LEFT OUTER JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

Grouping Factor

In earlier chapters, in particular in Chapter 4, I described a concept called a *grouping factor*. In particular, I used it in a problem to isolate islands, or ranges of consecutive elements in a sequence. Recall that the grouping factor is the factor you end up using in your GROUP BY clause to identify the group. In the earlier problem, I demonstrated two techniques to calculate the grouping factor. One method was calculating the maximum value within the group (specifically, the smallest value that is both greater than or equal to the current value and followed by a gap). The other method used row numbers.

Because this chapter covers aggregates, it is appropriate to revisit this very practical problem. In my examples here, I'll use the Stocks table, which you create and populate by running the code in Listing 6-14.

Listing 6-14 Creating and populating the Stocks table

```
USE tempdb;
GO
IF OBJECT_ID('Stocks') IS NOT NULL
  DROP TABLE Stocks;
GO

CREATE TABLE dbo.Stocks
(
  dt    DATETIME NOT NULL PRIMARY KEY,
  price INT      NOT NULL
);

INSERT INTO dbo.Stocks(dt, price) VALUES('20060801', 13);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060802', 14);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060803', 17);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060804', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060805', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060806', 52);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060807', 56);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060808', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060809', 70);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060810', 30);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060811', 29);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060812', 29);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060813', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060814', 45);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060815', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060816', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060817', 55);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060818', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060819', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060820', 15);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060821', 20);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060822', 30);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060823', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060824', 20);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060825', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060826', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060827', 70);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060828', 70);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060829', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060830', 30);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060831', 10);

CREATE UNIQUE INDEX idx_price_dt ON Stocks(price, dt);
```

The Stocks table contains daily stock prices.

Note Stock prices are rarely restricted to integers, and there is usually more than one stock, but I'll use integers and a single stock for simplification purposes. Also, stock markets usually don't have activity on Saturdays; because I want to demonstrate a technique over a sequence with no gaps, I introduced rows for Saturdays as well, with the same value that was stored in the preceding Friday.

The request is to isolate consecutive periods where the stock price was greater than or equal to 50. Figure 6-2 has a graphical depiction of the stock prices over time, and the arrows represent the periods you're supposed to return.

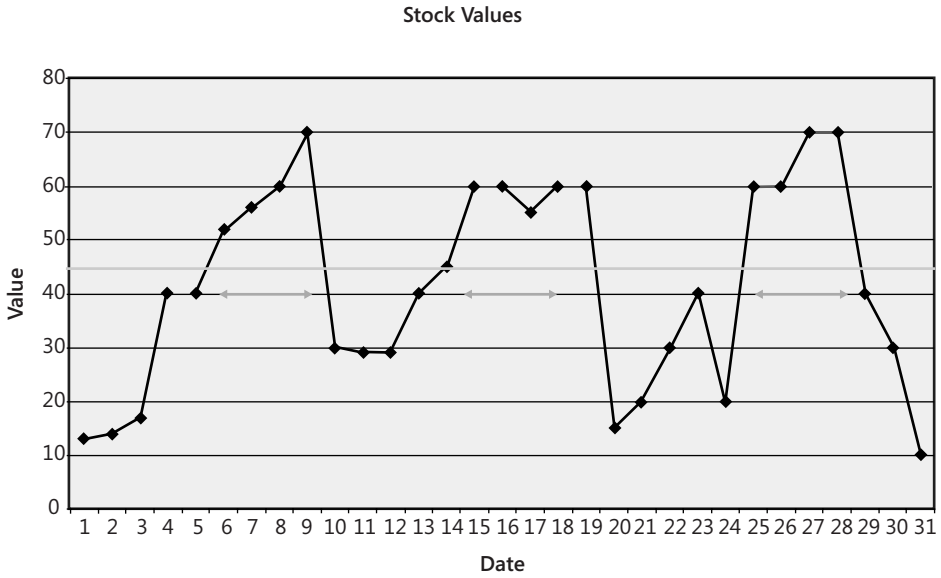

Figure 6-2 Periods in which stock values were greater than or equal to 50

For each such period, you need to return the starting date, ending date, duration in days, and the peak (maximum) price.

Let's start with a solution that does not use row numbers. The first step here is to filter only the rows where the price is greater than or equal to 50. Unlike the traditional problem where you really have gaps in the data, here the gaps appear only after filtering. The whole sequence still appears in the Stocks table. You can use this fact to your advantage. Of course, you could take the long route of calculating the maximum date within the group (the first date that is both later than or equal to the current date and followed by a gap). However, a much simpler and faster technique to calculate the grouping factor would be to return the first date that is greater than the current, on which the stock's price is less than 50. Here, you still get the same grouping factor for all elements of the same target group, yet you need only one nesting level of subqueries instead of two.

Here's the query that generates the desired result shown in Table 6-31:

```
SELECT MIN(dt) AS startrange, MAX(dt) AS endrange,
  DATEDIFF(day, MIN(dt), MAX(dt)) + 1 AS numdays,
  MAX(price) AS maxprice
FROM (SELECT dt, price,
        (SELECT MIN(dt)
         FROM dbo.Stocks AS S2
         WHERE S2.dt > S1.dt
           AND price < 50) AS grp
      FROM dbo.Stocks AS S1
      WHERE price >= 50) AS D
GROUP BY grp;
```

Table 6-31 Ranges Where Stock Values Were >= 50

startrange	*endrange*	*numdays*	*maxprice*
2006-08-06 00:00:00.000	2006-08-10 00:00:00.000	4	70
2006-08-15 00:00:00.000	2006-08-20 00:00:00.000	5	60
2006-08-25 00:00:00.000	2006-08-29 00:00:00.000	4	70

Of course, in SQL Server 2005 you can use the ROW_NUMBER function as I described in Chapter 4:

```
SELECT MIN(dt) AS startrange, MAX(dt) AS endrange,
  DATEDIFF(day, MIN(dt), MAX(dt)) + 1 AS numdays,
  MAX(price) AS maxprice
FROM (SELECT dt, price,
        dt - ROW_NUMBER() OVER(ORDER BY dt) AS grp
      FROM dbo.Stocks AS S1
      WHERE price >= 50) AS D
GROUP BY grp;
```

CUBE and ROLLUP

CUBE and ROLLUP are options available to queries that contain a GROUP BY clause. They are useful for applications that need to provide a changing variety of data aggregations based on varying sets of attributes or *dimensions*. (In the context of cubes, the word *dimension* is often used, either as a synonym for *attribute* or to describe a domain of values for an attribute.) I'll first describe the CUBE option, and then follow with a description of the ROLLUP option, which is a special case of CUBE.

CUBE

Imagine that your application needs to provide the users with the ability to request custom aggregates based on various sets of dimensions. Say, for example, that your base data is the Orders table that I used earlier in the chapter, and that the users need to analyze the data

based on three dimensions: employee, customer, and order year. If you group the data by all three dimensions, you've covered only one of the possibilities the users might be interested in. However, the users might request any set of dimensions (for example, employee alone, customer alone, order year alone, employee and customer, and so on). For each request, you would need to construct a different GROUP BY query and submit it to SQL Server, returning the result set to the client. That's a lot of roundtrips and a lot of network traffic.

As the number of dimensions grows, the number of possible GROUP BY queries increases dramatically. For *n* dimensions, there are 2^n different queries. With 3 dimensions, you're looking at 8 possible requests; with 4 dimensions, there are 16. With 10 dimensions (the maximum number of grouping expressions we will be able to use with CUBE), users could request any one of 1024 different GROUP BY queries.

Simply put, adding the option WITH CUBE to a query with all dimensions specified in the GROUP BY clause generates one unified result set out of the result sets of all the different GROUP BY queries over subsets of the dimensions. If you think about it, Analysis Services cubes give you similar functionality, but on a much larger scale and with substantially more sophisticated options. However, when you don't need to support dynamic analysis on such a scale and at such a level of sophistication, the option WITH CUBE allows you to achieve this within the relational database.

Because each set of dimensions generates a result set with a different subset of all possible result columns, the designers who implemented CUBE and ROLLUP had to come up with a placeholder for the values in the unneeded columns. The designers chose NULL. So, for example, all rows from the result set of a GROUP BY *empid, custid* would have NULL in the *orderyear* result column. This allows all result sets to be unified into one result set with one schema.

As an example, the following CUBE query returns all possible aggregations (total quantities) of orders based on the dimensions *empid*, *custid*, and *orderyear*, generating the output shown in Table 6-32:

```
SELECT empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY empid, custid, YEAR(orderdate)
WITH CUBE;
```

Table 6-32 Cube's Result

empid	*custid*	*orderyear*	*totalqty*
1	A	2002	12
1	A	NULL	12
1	B	2002	20
1	B	NULL	20

Table 6-32 Cube's Result

empid	*custid*	*orderyear*	*totalqty*
1	C	2003	14
1	C	NULL	14
1	NULL	NULL	46
2	B	2003	12
2	B	NULL	12
2	C	2004	20
2	C	NULL	20
2	NULL	NULL	32
3	A	2002	10
3	A	NULL	10
3	B	2004	15
3	B	NULL	15
3	C	2002	22
3	C	NULL	22
3	D	2002	30
3	D	NULL	30
3	NULL	NULL	77
4	A	2003	40
4	A	2004	10
4	A	NULL	50
4	NULL	NULL	50
NULL	NULL	NULL	205
NULL	A	2002	22
NULL	A	2003	40
NULL	A	2004	10
NULL	A	NULL	72
NULL	B	2002	20
NULL	B	2003	12
NULL	B	2004	15
NULL	B	NULL	47
NULL	C	2002	22
NULL	C	2003	14
NULL	C	2004	20
NULL	C	NULL	56
NULL	D	2002	30
NULL	D	NULL	30

Table 6-32 Cube's Result

empid	*custid*	*orderyear*	*totalqty*
1	NULL	2002	32
3	NULL	2002	62
NULL	NULL	2002	94
1	NULL	2003	14
2	NULL	2003	12
4	NULL	2003	40
NULL	NULL	2003	66
2	NULL	2004	20
3	NULL	2004	15
4	NULL	2004	10
NULL	NULL	2004	45

As long as the dimension columns in the table don't have NULLs, wherever you see a NULL in the result of the CUBE query, it logically means all. Later I'll discuss how to deal with NULLs in the queried table. For example, the row containing NULL, NULL, 2004, 45 shows the total quantity (45) for the orders of all employees and all customers for the order year 2004. You might want to cache the result set from a CUBE query in the client or middle tier, or you might want to save it in a temporary table and index it. The code in Listing 6-15 selects the result set into the temporary table #Cube and then creates a clustered index on all dimensions.

Listing 6-15 Populating a #Cube with CUBE query's result set

```
SELECT empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
INTO #Cube
FROM dbo.Orders
GROUP BY empid, custid, YEAR(orderdate)
WITH CUBE;

CREATE CLUSTERED INDEX idx_emp_cust_year
  ON #Cube(empid, custid, orderyear);
```

Any request for an aggregate can be satisfied using a seek operation within the clustered index. For example, the following query returns the total quantity for employee 1, generating the execution plan shown in Figure 6-3:

```
SELECT totalqty
FROM #Cube
WHERE empid = 1
  AND custid IS NULL
  AND orderyear IS NULL;
```

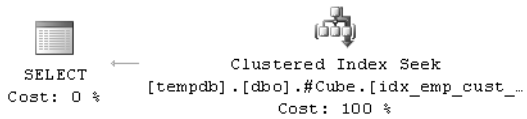

Figure 6-3 Execution plan for a query against the #Cube table

Once you're done querying the #Cube table, drop it:

```
DROP TABLE #Cube;
```

An issue might arise if dimension columns allow NULLs. For example, run the following code to allow NULLs in the *empid* column and introduce some actual NULL values:

```
ALTER TABLE dbo.Orders ALTER COLUMN empid INT NULL;
UPDATE dbo.Orders SET empid = NULL WHERE orderid IN(10001, 20001);
```

You should realize that when you run a CUBE query now, a NULL in the *empid* column is ambiguous. When it results from NULL in the *empid* column, it represents the group of unknown employees. When it is generated by the CUBE option, it represents all employees. However, without any specific treatment of the NULLs, you won't be able to tell which it is. I like to simply substitute for NULL a value that I know can't be used in the data—for example, −1 as the *empid*. I use the COALESCE or ISNULL function for this purpose. After this substitution, the value −1 would represent unknown employees, and NULL can only mean all employees. Here's a query that incorporates this logic:

```
SELECT COALESCE(empid, -1) AS empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY COALESCE(empid, -1), custid, YEAR(orderdate)
WITH CUBE;
```

Another option is to use the T-SQL function GROUPING, which was designed to address the ambiguity of NULL in the result set. You supply the function with the dimension column name as input. The value of GROUPING(<dimension>) indicates whether or not the value of <dimension> in the row represents the value for a group (in this case, GROUPING returns 0) or is a placeholder that represents all values (in this case, GROUPING returns 1). Specifically for the dimension value NULL, GROUPING returns 1 if the NULL is a result of the CUBE option (meaning all) and 0 if it represents the group of source NULLs. Here's a query that uses the function GROUPING:

```
SELECT empid, GROUPING(empid) AS grp_empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY empid, custid, YEAR(orderdate)
WITH CUBE;
```

If you're spooling the result set of a CUBE query to a temporary table, don't forget to include the grouping columns in the index, and also be sure to include them in your filters. For example, assume you spooled the result set of the preceding query to a temporary table called #Cube. The following query would return the total quantity for customer A:

```
SELECT totalqty
FROM #Cube
WHERE empid IS NULL AND grp_empid = 1
  AND custid = 'A'
  AND orderyear IS NULL;
```

ROLLUP

ROLLUP is a special case of CUBE that you can use when there's a hierarchy on the dimensions. For example, suppose you want to analyze order quantities based on the dimensions order year, order month, and order day. Assume you don't really care about totals of an item in one level of granularity across all values in a higher level of granularity—for example, the totals of the third day in all months and all years. You care only about the totals of an item in one level of granularity for all lower level values—for example, the total for year 2004, all months, all days. ROLLUP gives you just that. It eliminates all "noninteresting" aggregations in a hierarchical case. More accurately, it doesn't even bother to calculate them at all, so you should expect better performance from a ROLLUP query than a CUBE query based on the same dimensions.

As an example for using ROLLUP, the following query returns the total order quantities for the dimensions order year, order month, and order day, and it returns the output shown in Table 6-33:

```
SELECT
  YEAR(orderdate)  AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate)   AS orderday,
  SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY YEAR(orderdate), MONTH(orderdate), DAY(orderdate)
WITH ROLLUP;
```

Table 6-33 Rollup's Result

orderyear	*ordermonth*	*orderday*	*totalqty*
2002	4	18	22
2002	4	NULL	22
2002	8	2	10
2002	8	NULL	10
2002	9	7	30
2002	9	NULL	30

Table 6-33 Rollup's Result

orderyear	*ordermonth*	*orderday*	*totalqty*
2002	12	24	32
2002	12	NULL	32
2002	NULL	NULL	94
2003	1	9	40
2003	1	18	14
2003	1	NULL	54
2003	2	12	12
2003	2	NULL	12
2003	NULL	NULL	66
2004	2	12	10
2004	2	16	20
2004	2	NULL	30
2004	4	18	15
2004	4	NULL	15
2004	NULL	NULL	45
NULL	NULL	NULL	205

Conclusion

This chapter covered various solutions to data-aggregation problems that reused key querying techniques I introduced earlier in the book. It also introduced new techniques, such as dealing with tiebreakers by using concatenation, calculating a minimum using the MAX function, pivoting, unpivoting, calculating custom aggregates by using specialized techniques, and others.

As you probably noticed, data-aggregation techniques involve a lot of logical manipulation. If you're looking for ways to improve your logic, you can practice pure logical puzzles, as they have a lot in common with querying problems in terms of the thought processes involved. You can find pure logic puzzles in Appendix A.