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.

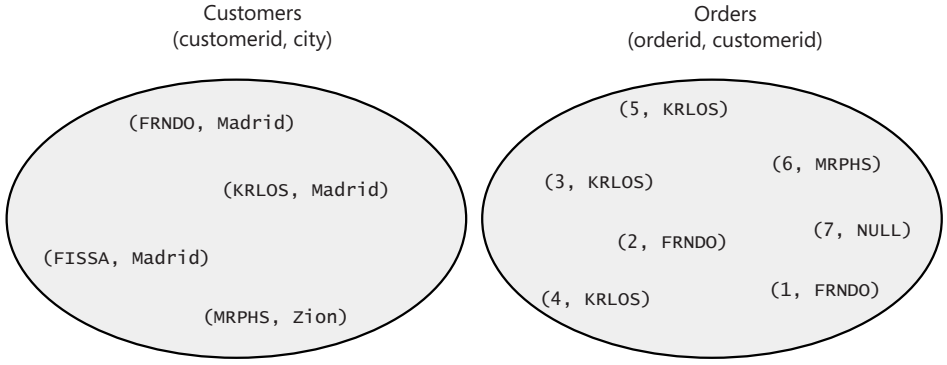

Figure 1-1 Customers and Orders sets

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.