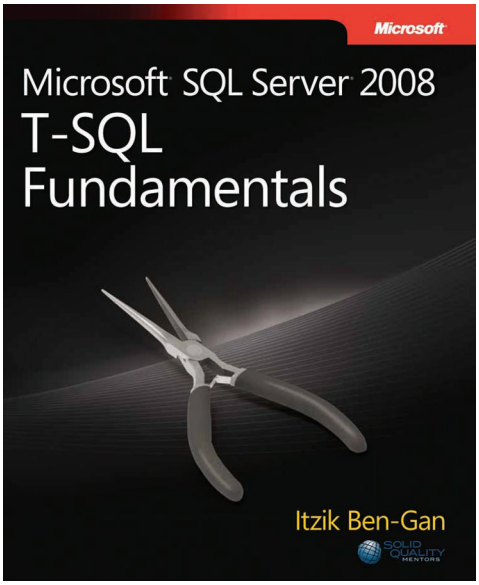

Microsoft® SQL Server® 2008 T-SQL Fundamentals

Itzik Ben-Gan
(Solid Quality Mentors)

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

9780735626010

Microsoft Press

Table of Contents

Chapter 3

Joins

In this chapter:

The FROM clause of a query is the first clause to be logically processed, and within the FROM clause table operators operate on input tables. Microsoft SQL Server 2008 supports four table operators—JOIN, APPLY, PIVOT, and UNPIVOT. The JOIN table operator is standard, while APPLY, PIVOT, and UNPIVOT are T-SQL extensions to the standard. These last three were introduced in SQL Server 2005. Each table operator acts on tables provided to it as input, applies a set of logical query processing phases, and returns a table result. This chapter focuses on the JOIN table operator. The APPLY operator will be covered in Chapter 5, "Table Expressions," and the PIVOT and UNPIVOT operators will be covered in Chapter 7, "Pivot, Unpivot, and Grouping Sets."

A JOIN table operator operates on two input tables. The three fundamental types of joins are cross, inner, and outer. The three types of joins differ in how they apply their logical query processing phases; each type applies a different set of phases. A cross join applies only one phase—Cartesian Product. An inner join applies two phases—Cartesian Product and Filter. An outer join applies three phases—Cartesian Product, Filter, and Add Outer Rows. This chapter explains each of the join types and the phases involved in detail.

Logical query processing describes a generic series of logical steps that for any given query produces the correct result, while physical query processing is the way the query is processed by the RDBMS engine in practice. Some phases of logical query processing of joins may sound inefficient, but the physical implementation may be optimized. It's important to stress the term *logical* in logical query processing. The steps in the process apply operations to the input tables based on relational algebra. The database engine does not have to follow logical query processing phases literally as long as it can guarantee that the result that it produces is the same as dictated by logical query processing. The SQL Server relational engine often applies many shortcuts for optimization purposes when it knows that it can still produce the correct result. Even though this book's focus is to understand the logical aspects of querying, I want to stress this point to avoid any misunderstanding and confusion.

Cross Joins

Logically, a cross join is the simplest type of join. A cross join implements only one logical query processing phase—a Cartesian Product. This phase operates on the two tables provided as inputs to the join, and produces a Cartesian product of the two. That is, each row from one input is matched with all rows from the other. So if you have *m* rows in one table and *n* rows in the other, you get *m* × *n* rows in the result.

SQL Server supports two standard syntaxes for cross joins—the ANSI SQL-92 and ANSI SQL-89 syntaxes. I recommend that you use the ANSI-SQL 92 syntax for reasons that I'll describe shortly. Therefore, ANSI-SQL 92 syntax is the main syntax that I use throughout the book. For the sake of completeness, I describe both syntaxes in this section.

ANSI SQL-92 Syntax

The following query applies a cross join between the Customers and Employees tables (using the ANSI SQL-92 syntax) in the TSQLFundamentals2008 database, and returns the custid and empid attributes in the result set:

```
USE TSQLFundamentals2008;

SELECT C.custid, E.empid
FROM Sales.Customers AS C
  CROSS JOIN HR.Employees AS E;
```

Because there are 91 rows in the Customers table and 9 rows in the Employees table, this query produces a result set with 819 rows, as shown here in abbreviated form:

```
custid      empid
----------- -----------
1           1
1           2
1           3
1           4
1           5
1           6
1           7
1           8
1           9
2           1
2           2
2           3
2           4
2           5
2           6
2           7
2           8
2           9
...

(819 row(s) affected)
```

Using the ANSI SQL-92 syntax, you specify the *CROSS JOIN* keywords between the two tables involved in the join.

Notice that in the FROM clause of the preceding query, I assigned the aliases C and E to the Customers and Employees tables, respectively. The result set produced by the cross join is a virtual table with attributes that originate from both sides of the join. Because I assigned aliases to the source tables, the names of the columns in the virtual table are prefixed by the table aliases (for example, C.custid, E.empid). If you do not assign aliases to the tables in the FROM clause, the names of the columns in the virtual table are prefixed by the full source table names (for example, Customers.custid, Employees.empid). The purpose of the prefixes is to enable the identification of columns in an unambiguous manner when the same column name appears in both tables. The aliases of the tables are assigned for brevity. Note that you are required to use column prefixes only when referring to ambiguous column names (column names that appear in more than one table); in unambiguous cases column prefixes are optional. However, some people find it a good practice to always use column prefixes for the sake of clarity. Also note that if you assign an alias to a table, it is invalid to use the full table name as a column prefix; in ambiguous cases you have to use the table alias as a prefix.

ANSI SQL-89 Syntax

SQL Server also supports an older syntax for cross joins that was introduced in ANSI SQL-89. In this syntax you simply specify a comma between the table names like so:

```
SELECT C.custid, E.empid
FROM Sales.Customers AS C, HR.Employees AS E;
```

There is no logical or performance difference between the two syntaxes. Both syntaxes are integral parts of the latest SQL standard (ANSI SQL:2006 at the time of this writing), and both are fully supported by the latest version of SQL Server (SQL Server 2008 at the time of this writing). I am not aware of any plans to deprecate the older syntax, and I don't see any reason to do so while it's an integral part of the standard. However, I recommend using the ANSI SQL-92 syntax for reasons that will become clear after inner joins are explained.

Self Cross Joins

You can join multiple instances of the same table. This capability is known as *self-join* and is supported with all fundamental join types (cross, inner, and outer). For example, the following query performs a self cross join between two instances of the Employees table:

```
SELECT
  E1.empid, E1.firstname, E1.lastname,
  E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
  CROSS JOIN HR.Employees AS E2;
```

This query produces all possible combinations of pairs of employees. Because the Employees table has 9 rows, this query returns 81 rows, shown here in abbreviated form:

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

(81 row(s) affected)
```

In a self-join, aliasing tables is not optional. Without table aliases, all column names in the result of the join would be ambiguous.

Producing Tables of Numbers

One situation in which cross joins can be very handy is when they are used to produce a result set with a sequence of integers (1, 2, 3, and so on). Such a sequence of numbers is an extremely powerful tool that I use for many purposes. Using cross joins you can produce the sequence of integers in a very efficient manner.

You can start by creating a table called Digits with a column called digit, and populate the table with 10 rows with the digits 0 through 9. Run the following code to create the Digits table in the tempdb database (for test purposes) and populate it with the 10 digits:

```
USE tempdb;
IF OBJECT_ID('dbo.Digits', 'U') IS NOT NULL DROP TABLE dbo.Digits;
CREATE TABLE dbo.Digits(digit INT NOT NULL PRIMARY KEY);

INSERT INTO dbo.Digits(digit)
  VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9);

/*
Note:
Above INSERT syntax is new in Microsoft SQL Server 2008.
```

```
In earlier versions use:

INSERT INTO dbo.Digits(digit) VALUES(0);
INSERT INTO dbo.Digits(digit) VALUES(1);
INSERT INTO dbo.Digits(digit) VALUES(2);
INSERT INTO dbo.Digits(digit) VALUES(3);
INSERT INTO dbo.Digits(digit) VALUES(4);
INSERT INTO dbo.Digits(digit) VALUES(5);
INSERT INTO dbo.Digits(digit) VALUES(6);
INSERT INTO dbo.Digits(digit) VALUES(7);
INSERT INTO dbo.Digits(digit) VALUES(8);
INSERT INTO dbo.Digits(digit) VALUES(9);
*/

SELECT digit FROM dbo.Digits;
```

This code uses a couple of syntax elements for the first time in this book, so I'll briefly explain them. Any text residing within a block starting with /* and ending with */ is treated as a block comment and is ignored by SQL Server. This code also uses an *INSERT* statement to populate the Digits table. If you're not familiar with the syntax of the *INSERT* statement, see Chapter 8, "Data Modification," for details. Note, however, that this code uses new syntax that was introduced in SQL Server 2008 for the *INSERT VALUES* statement, allowing a single statement to insert multiple rows. A block comment embedded in the code explains that in earlier versions you need to use a separate *INSERT VALUES* statement for each row.

The contents of the Digits table are shown here:

```
digit
-----------
0
1
2
3
4
5
6
7
8
9
```

Suppose you need to write a query that produces a sequence of integers in the range 1 through 1,000. You can cross three instances of the Digits table, each representing a different power of 10 (1, 10, 100). By crossing three instances of the same table, each instance with 10 rows, you get a result set with 1,000 rows. To produce the actual number, multiply the digit from each instance by the power of 10 it represents, sum the results, and add 1. Here's the complete query:

```
SELECT D3.digit * 100 + D2.digit * 10 + D1.digit + 1 AS n
FROM         dbo.Digits AS D1
  CROSS JOIN dbo.Digits AS D2
  CROSS JOIN dbo.Digits AS D3
ORDER BY n;
```

This query returns the following output, shown here in abbreviated form:

```
n
-----------
1
2
3
4
5
6
7
8
9
10
...
998
999
1000

(1000 row(s) affected)
```

This was just an example producing a sequence of 1,000 integers. If you need more, you can add more instances of the Digits table to the query. For example, if you need to produce a sequence of 1,000,000 rows, you would need to join six instances.

Inner Joins

An inner join applies two logical query processing phases—it applies a Cartesian product between the two input tables like a cross join, and then it filters rows based on a predicate that you specify. Like cross joins, inner joins have two standard syntaxes: ANSI SQL-92 and ANSI SQL-89.

ANSI SQL-92 Syntax

Using the ANSI SQL-92 syntax, you specify the *INNER JOIN* keywords between the table names. The *INNER* keyword is optional because an inner join is the default, so you can specify the *JOIN* keyword alone. You specify the predicate that is used to filter rows in a designated clause called ON. This predicate is also known as the *join condition*.

For example, the following query performs an inner join between the Employees and Orders tables in the TSQLFundamentals2008 database, matching employees and orders based on the predicate E.empid = O.empid:

```
USE TSQLFundamentals2008;

SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
  JOIN Sales.Orders AS O
    ON E.empid = O.empid;
```

This query produces the following result set, shown here in abbreviated form:

```
empid       firstname  lastname             orderid
----------- ---------- -------------------- -----------
1           Sara       Davis                10258
1           Sara       Davis                10270
1           Sara       Davis                10275
1           Sara       Davis                10285
1           Sara       Davis                10292
...
2           Don        Funk                 10265
2           Don        Funk                 10277
2           Don        Funk                 10280
2           Don        Funk                 10295
2           Don        Funk                 10300
...

(830 row(s) affected)
```

For most people the easiest way to think of such an inner join is as matching each employee row to all order rows that have the same employee ID as the employee's employee ID. This is a simplified way to think of the join. The more formal way to think of the join based on relational algebra is that first the join performs a Cartesian product of the two tables (9 employee rows × 830 order rows = 7,470 rows), and then filters rows based on the predicate E.empid = O.empid, eventually returning 830 rows. As mentioned earlier, that's just the logical way the join is processed; in practice, physical processing of the query by the database engine can be different.

Recall the discussion from previous chapters about the three-valued predicate logic used by SQL. Like with the WHERE and HAVING clauses, the ON clause also returns only rows for which the predicate returns TRUE, and does not return rows for which the predicate evaluates to FALSE or UNKNOWN.

In the TSQLFundamentals2008 database all employees have related orders, so all employees show up in the output. However, had there been employees with no related orders, they would have been filtered out by the filter phase.

ANSI SQL-89 Syntax

Similar to cross joins, inner joins can be expressed using the ANSI SQL-89 syntax. You specify a comma between the table names just like in a cross join, and specify the join condition in the query's WHERE clause, like so:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O
WHERE E.empid = O.empid;
```

Note that the ANSI SQL-89 syntax has no ON clause.

Again, both syntaxes are standard, fully supported by SQL Server, and interpreted the same by the engine, so you shouldn't expect any performance difference between the two. But one syntax is safer, as explained in the next section.

Inner Join Safety

I strongly recommend that you stick to the ANSI SQL-92 join syntax because it is safer in several ways. Say you intend to write an inner join query, and by mistake forget to specify the join condition. With the ANSI SQL-92 syntax the query becomes invalid and the parser generates an error. For example, try to run the following code:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
  JOIN Sales.Orders AS O;
```

You get the following error:

```
Msg 102, Level 15, State 1, Line 3
Incorrect syntax near ';'.
```

Even though it might not be obvious immediately that the error involves a missing join condition, you will figure it out eventually and fix the query. However, if you forget to specify the join condition using the ANSI SQL-89 syntax, you get a valid query that performs a cross join:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O;
```

Because the query doesn't fail, the logical error might go unnoticed for a while, and users of your application might end up relying on incorrect results. It is unlikely that a programmer would forget to specify the join condition with such short and simple queries; however, most production queries are much more complicated and have multiple tables, filters, and other query elements. In those cases the likelihood of forgetting to specify a join condition increases.

If I've convinced you that it is important to use the ANSI SQL-92 syntax for inner joins, you might wonder whether the recommendation holds for cross joins. Because no join condition is involved, you might think that both syntaxes are just as good for cross joins. However, I recommend staying with the ANSI SQL-92 syntax with cross joins for a couple of reasons—one being consistency. Also, let's say you do use the ANSI SQL-89 syntax. Even if you intended to write a cross join, when other developers need to review or maintain your code, how will they know whether you intended to write a cross join or intended to write an inner join and forgot to specify the join condition?

Further Join Examples

This section covers a few join examples that are known by specific names, including composite joins, non-equi joins, and multi-table joins.

Composite Joins

A composite join is simply a join based on a predicate that involves more than one attribute from each side. A composite join is commonly required when you need to join two tables based on a primary key–foreign key relationship, and the relationship is composite: that is, based on more than one attribute. For example, suppose you have a foreign key defined on dbo.Table2, columns col1, col2, referencing dbo.Table1, columns col1, col2, and you need to write a query that joins the two based on primary key–foreign key relationship. The FROM clause of the query would look like this:

```
FROM dbo.Table1 AS T1
  JOIN dbo.Table2 AS T2
    ON T1.col1 = T2.col1
    AND T1.col2 = T2.col2
```

For a more tangible example, suppose that you need to audit updates to column values against the OrderDetails table in the TSQLFundamentals2008 database. You create a custom auditing table called OrderDetailsAudit:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Sales.OrderDetailsAudit', 'U') IS NOT NULL
  DROP TABLE Sales.OrderDetailsAudit;
CREATE TABLE Sales.OrderDetailsAudit
(
  lsn        INT NOT NULL IDENTITY,
  orderid    INT NOT NULL,
  productid  INT NOT NULL,
  dt         DATETIME NOT NULL,
  loginname  sysname NOT NULL,
  columnname sysname NOT NULL,
  oldval     SQL_VARIANT,
  newval     SQL_VARIANT,
  CONSTRAINT PK_OrderDetailsAudit PRIMARY KEY(lsn),
  CONSTRAINT FK_OrderDetailsAudit_OrderDetails
    FOREIGN KEY(orderid, productid)
    REFERENCES Sales.OrderDetails(orderid, productid)
);
```

Each audit row stores a log serial number (lsn), the key of the modified row (orderid, productid), the name of the modified column (columnname), the old value (*oldval*), new value (*newval*), when the change took place (dt), and who made the change (loginname). The table has a foreign key defined on the attributes orderid, productid, referencing the primary key of the OrderDetails table, which is defined on the attributes orderid, productid.

Suppose that you already have in place all the required processes that audit column value changes taking place in the OrderDetails table in the OrderDetailsAudit table.

You need to write a query that returns all value changes that took place against the column qty, but in each result row you need to return the current value from the OrderDetails table, and the values before and after the change from the OrderDetailsAudit table. You need to join the two tables based on primary key–foreign key relationship like so:

```
SELECT OD.orderid, OD.productid, OD.qty,
  ODA.dt, ODA.loginname, ODA.oldval, ODA.newval
FROM Sales.OrderDetails AS OD
  JOIN Sales.OrderDetailsAudit AS ODA
    ON OD.orderid = ODA.orderid
    AND OD.productid = ODA.productid
WHERE ODA.columnname = N'qty';
```

Because the relationship is based on multiple attributes, the join condition is composite.

Non-Equi Joins

When the join condition involves only an equality operator, the join is said to be an equi join. When the join condition involves any operator besides equality, the join is said to be a non-equi join. As an example of a non-equi join, the following query joins two instances of the Employees table to produce unique pairs of employees:

```
SELECT
  E1.empid, E1.firstname, E1.lastname,
  E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
  JOIN HR.Employees AS E2
    ON E1.empid < E2.empid;
```

Notice the predicate specified in the ON clause. The purpose of the query is to produce unique pairs of employees. Had you used a cross join, you would have gotten self pairs (for example, 1 with 1), and also mirrored pairs (for example, 1 with 2 and also 2 with 1). Using an inner join with a join condition that says that the key in the left side must be smaller than the key in the right side eliminates the two inapplicable cases. Self pairs are eliminated because both sides are equal. With mirrored pairs, only one of the two cases qualifies because out of the two cases, only one will have a left key that is smaller than the right key. In our case, out of the 81 possible pairs of employees that a cross join would have returned, our query returns the 36 unique pairs shown here:

```
empid firstname  lastname         empid  firstname  lastname
----- ---------- ---------------- ------ ---------- -----------------
1     Sara       Davis            2      Don        Funk
1     Sara       Davis            3      Judy       Lew
2     Don        Funk             3      Judy       Lew
```

```
1       Sara        Davis           4       Yael        Peled
2       Don         Funk            4       Yael        Peled
3       Judy        Lew             4       Yael        Peled
1       Sara        Davis           5       Sven        Buck
2       Don         Funk            5       Sven        Buck
3       Judy        Lew             5       Sven        Buck
4       Yael        Peled           5       Sven        Buck
1       Sara        Davis           6       Paul        Suurs
2       Don         Funk            6       Paul        Suurs
3       Judy        Lew             6       Paul        Suurs
4       Yael        Peled           6       Paul        Suurs
5       Sven        Buck            6       Paul        Suurs
1       Sara        Davis           7       Russell     King
2       Don         Funk            7       Russell     King
3       Judy        Lew             7       Russell     King
4       Yael        Peled           7       Russell     King
5       Sven        Buck            7       Russell     King
6       Paul        Suurs           7       Russell     King
1       Sara        Davis           8       Maria       Cameron
2       Don         Funk            8       Maria       Cameron
3       Judy        Lew             8       Maria       Cameron
4       Yael        Peled           8       Maria       Cameron
5       Sven        Buck            8       Maria       Cameron
6       Paul        Suurs           8       Maria       Cameron
7       Russell     King            8       Maria       Cameron
1       Sara        Davis           9       Zoya        Dolgopyatova
2       Don         Funk            9       Zoya        Dolgopyatova
3       Judy        Lew             9       Zoya        Dolgopyatova
4       Yael        Peled           9       Zoya        Dolgopyatova
5       Sven        Buck            9       Zoya        Dolgopyatova
6       Paul        Suurs           9       Zoya        Dolgopyatova
7       Russell     King            9       Zoya        Dolgopyatova
8       Maria       Cameron         9       Zoya        Dolgopyatova

(36 row(s) affected)
```

If it is still not clear to you what this query does, try to process it one step at a time with a smaller set of employees. For example, suppose the Employees table contained only employees 1, 2, and 3. First, produce the Cartesian product of two instances of the table:

```
E1.empid      E2.empid
------------- -------------
1             1
1             2
1             3
2             1
2             2
2             3
3             1
3             2
3             3
```

Next, filter the rows based on the predicate E1.empid < E2.empid, and you are left with only three rows:

```
E1.empid      E2.empid
------------- -------------
1             2
1             3
2             3
```

Multi-Table Joins

A join table operator operates only on two tables, but a single query can have multiple joins. In general, when more than one table operator appears in the FROM clause, the table operators are logically processed from left to right. That is, the result table of the first table operator is served as the left input to the second table operator; the result of the second table operator is served as the left input to the third table operator and so on. So if there are multiple joins in the FROM clause, logically the first join operates on two base tables, but all other joins get the result of the preceding join as their left input. With cross joins and inner joins, the database engine can (and often does) internally rearrange join ordering for optimization purposes because it won't have an impact on the correctness of the result of the query.

As an example, the following query joins the Customers and Orders tables to match customers with their orders, and joins the result of the first join with the OrderDetails table to match orders with their order lines:

```
SELECT
  C.custid, C.companyname, O.orderid,
  OD.productid, OD.qty
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON C.custid = O.custid
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

This query returns the following output, shown here in abbreviated form:

```
custid      companyname       orderid     productid   qty
----------- ----------------- ----------- ----------- ------
85          Customer ENQZT    10248       11          12
85          Customer ENQZT    10248       42          10
85          Customer ENQZT    10248       72          5
79          Customer FAPSM    10249       14          9
79          Customer FAPSM    10249       51          40
34          Customer IBVRG    10250       41          10
34          Customer IBVRG    10250       51          35
34          Customer IBVRG    10250       65          15
84          Customer NRCSK    10251       22          6
84          Customer NRCSK    10251       57          15
...

(2155 row(s) affected)
```

Outer Joins

Outer joins are usually harder for people to grasp compared to the other types of joins. First I will describe the fundamentals of outer joins. If by the end of the section "Fundamentals of Outer Joins," you feel very comfortable with the material and are ready for more advanced content, you can read an optional section describing aspects of outer joins that are beyond the fundamentals. Otherwise, feel free to skip that part and return to it when you feel comfortable with the material.

Fundamentals of Outer Joins

Outer joins were introduced in ANSI SQL-92 and unlike inner and cross joins, they only have one standard syntax—the one where you specify the *JOIN* keyword between the table names, and the join condition in the ON clause. Outer joins apply the two logical processing phases that inner joins apply (Cartesian product and the ON filter), plus a third phase called Adding Outer Rows that is unique to this type of join.

In an outer join you mark a table as a "preserved" table by using the keywords *LEFT OUTER JOIN*, *RIGHT OUTER JOIN*, or *FULL OUTER JOIN* between the table names. The *OUTER* keyword is optional. The *LEFT* keyword means that the rows of the left table are preserved, the *RIGHT* keyword means that the rows in the right table are preserved, and the *FULL* keyword means that the rows in both the left and right tables are preserved. The third logical query processing phase of an outer join identifies the rows from the preserved table that did not find matches in the other table based on the ON predicate. This phase adds those rows to the result table produced by the first two phases of the join, and uses NULLs as place holders for the attributes from the nonpreserved side of the join in those outer rows.

A good way to understand outer joins is through an example. The following query joins the Customers and Orders tables based on a match between the customer's customer ID and the order's customer ID to return customers and their orders. The join type is a left outer join; therefore, the query also returns customers who did not place any orders in the result:

```
SELECT C.custid, C.companyname, O.orderid
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid;
```

This query returns the following output, shown here in abbreviated form:

```
custid      companyname     orderid
----------- --------------- -----------
1           Customer NRZBB  10643
1           Customer NRZBB  10692
1           Customer NRZBB  10702
1           Customer NRZBB  10835
1           Customer NRZBB  10952
...
```

```
21          Customer KIDPX  10414
21          Customer KIDPX  10512
21          Customer KIDPX  10581
21          Customer KIDPX  10650
21          Customer KIDPX  10725
22          Customer DTDMN  NULL
23          Customer WVFAF  10408
23          Customer WVFAF  10480
23          Customer WVFAF  10634
23          Customer WVFAF  10763
23          Customer WVFAF  10789
...
56          Customer QNIVZ  10684
56          Customer QNIVZ  10766
56          Customer QNIVZ  10833
56          Customer QNIVZ  10999
56          Customer QNIVZ  11020
57          Customer WVAXS  NULL
58          Customer AHXHT  10322
58          Customer AHXHT  10354
58          Customer AHXHT  10474
58          Customer AHXHT  10502
58          Customer AHXHT  10995
...
91          Customer CCFIZ  10792
91          Customer CCFIZ  10870
91          Customer CCFIZ  10906
91          Customer CCFIZ  10998
91          Customer CCFIZ  11044

(832 row(s) affected)
```

Two customers in the Customers table did not place any orders. Their IDs are 22 and 57. Observe that in the output of the query both customers are returned with NULLs in the attributes from the Orders table. Logically, the rows for these two customers were filtered out by the second phase of the join (filter based on the ON predicate), but the third phase added those as outer rows. Had the join been an inner join, these two rows would not have been returned. These two rows are added to preserve all the rows of the left table.

You can consider two kinds of rows in the result of an outer join in respect to the preserved side—inner rows and outer rows. Inner rows are rows that have matches in the other side based on the ON predicate, and outer rows are rows that don't. An inner join returns only inner rows, while an outer join returns both inner and outer rows.

A common question when using outer joins that is the source of a lot of confusion is whether to specify a predicate in the ON or WHERE clauses of a query. You can see that with respect to rows from the preserved side of an outer join, the filter based on the ON predicate is not final. In other words, the ON predicate does not determine whether the row will show up in the output, only whether it will be matched with rows from the other side. So when you need to express a predicate that is not final—meaning a predicate that determines which rows

to match from the nonpreserved side—specify the predicate in the ON clause. When you need a filter to be applied after outer rows are produced, and you want the filter to be final, specify the predicate in the WHERE clause. The WHERE clause is processed after the FROM clause—namely, after all table operators were processed and (in the case of outer joins), after all outer rows were produced. Also, the WHERE clause is final with respect to rows that it filters out, unlike the ON clause.

Suppose that you need to return only customers who did not place any orders, or more technically speaking, you need to return only outer rows. You can use the previous query as your basis, and add a WHERE clause that filters only outer rows. Remember that outer rows are identified by the NULLs in the attributes from the nonpreserved side of the join. So you can filter only the rows where one of the attributes in the nonpreserved side of the join is NULL, like so:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderid IS NULL;
```

This query returns only two rows, with the customers 22 and 57:

```
custid      companyname
----------- ---------------
22          Customer DTDMN
57          Customer WVAXS

(2 row(s) affected)
```

Notice a couple of important things about this query. Recall the discussions about NULLs earlier in the book: When looking for a NULL you should use the operator IS NULL and not an equality operator, because an equality operator comparing something with a NULL always returns UNKNOWN—even when comparing two NULLs. Also, the choice of which attribute from the nonpreserved side of the join to filter is important. You should choose an attribute that can only have a NULL when the row is an outer row and not otherwise (for example, a NULL originating from the base table). For this purpose, three cases are safe to consider—a primary key column, a join column, and a column defined as NOT NULL. A primary key column cannot be NULL; therefore, a NULL in such a column can only mean that the row is an outer row. If a row has a NULL in the join column, that row is filtered out by the second phase of the join, so a NULL in such a column can only mean that it's an outer row. And obviously a NULL in a column that is defined as NOT NULL can only mean that the row is an outer row.

To practice what you've learned and get a better grasp of outer joins, make sure that you perform the exercises for this chapter.

Beyond the Fundamentals of Outer Joins

This section covers more advanced aspects of outer joins and is provided as optional reading for when you feel very comfortable with the fundamentals of outer joins.

Including Missing Values

You can use outer joins to identify and include missing values when querying data. For example, suppose that you need to query all orders from the Orders table in the TSQLFundamentals2008 database. You need to ensure that you get at least one row in the output for each date in the range January 1, 2006 through December 31, 2008. You don't want to do anything special with dates within the range that have orders. But you do want the output to include the dates with no orders, with NULLs as placeholders in the attributes of the order.

To solve the problem, you can first write a query that returns a sequence of all dates in the requested date range. You can then perform a left outer join between that set and the Orders table. This way the result also includes the missing order dates.

To produce a sequence of dates in a given range, I usually use an auxiliary table of numbers. I create a table called Nums with a column called n, and populate it with a sequence of integers (1, 2, 3, and so on). I find that an auxiliary table of numbers is an extremely powerful general-purpose tool that I end up using to solve many problems. You need to create it only once in the database and populate it with as many numbers as you might need. Run the code in Listing 3-1 to create the Nums table in the dbo schema and populate it with 100,000 rows:

LISTING 3-1 Code to Create and Populate the Auxiliary Table Nums

```
SET NOCOUNT ON;
USE TSQLFundamentals2008;
IF OBJECT_ID('dbo.Nums', 'U') IS NOT NULL DROP TABLE dbo.Nums;
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);

DECLARE @i AS INT = 1;
/*
Note:
The ability to declare and initialize variables in one statement
is new in Microsoft SQL Server 2008.
In earlier versions use separate DECLARE and SET statements:

DECLARE @i AS INT;
SET @i = 1;
*/
BEGIN TRAN
  WHILE @i <= 100000
  BEGIN
    INSERT INTO dbo.Nums VALUES(@i);
    SET @i = @i + 1;
  END
COMMIT TRAN
SET NOCOUNT OFF;
```

Note Don't worry if you don't yet understand some parts of the code, such as using variables and loops—those are explained later in the book. For now, it's enough to understand what this code is supposed to do; how it does it is not the focus of discussion here. But in case you're curious and cannot resist, you can find details in Chapter 10, "Programmable Objects." I should point out, however, that declaring and initializing variables in the same statement is new in SQL Server 2008 as the block comment that appears in the code explains. If you're working with an earlier version, you should use separate *DECLARE* and *SET* statements.

As the first step in the solution, you need to produce a sequence of all dates in the requested range. You can achieve this by querying the Nums table, and filtering as many numbers as the number of days in the requested date range. You can use the *DATEDIFF* function to calculate that number. By adding *n* - 1 days to the starting point of the date range (January 1, 2006) you get the actual date in the sequence. Here's the solution query:

```
SELECT DATEADD(day, n-1, '20060101') AS orderdate
FROM dbo.Nums
WHERE n <= DATEDIFF(day, '20060101', '20081231') + 1
ORDER BY orderdate;
```

This query returns a sequence of all dates in the range January 1, 2006 through December 31, 2008, as shown here in abbreviated form:

```
orderdate
-----------------------
2006-01-01 00:00:00.000
2006-01-02 00:00:00.000
2006-01-03 00:00:00.000
2006-01-04 00:00:00.000
2006-01-05 00:00:00.000
...
2008-12-27 00:00:00.000
2008-12-28 00:00:00.000
2008-12-29 00:00:00.000
2008-12-30 00:00:00.000
2008-12-31 00:00:00.000

(1096 row(s) affected)
```

The next step is to extend the previous query, adding a left outer join between Nums and the Orders tables. The join condition compares the order date produced from the Nums table using the expression DATEADD(day, Nums.n - 1, '20060101') and the orderdate from the Orders table like so:

```
SELECT DATEADD(day, Nums.n - 1, '20060101') AS orderdate,
  O.orderid, O.custid, O.empid
FROM dbo.Nums
  LEFT OUTER JOIN Sales.Orders AS O
    ON DATEADD(day, Nums.n - 1, '20060101') = O.orderdate
WHERE Nums.n <= DATEDIFF(day, '20060101', '20081231') + 1
ORDER BY orderdate;
```

This query produces the following output, shown here in abbreviated form:

```
orderdate                orderid     custid      empid
------------------------ ----------- ----------- -----------
2006-01-01 00:00:00.000  NULL        NULL        NULL
2006-01-02 00:00:00.000  NULL        NULL        NULL
2006-01-03 00:00:00.000  NULL        NULL        NULL
2006-01-04 00:00:00.000  NULL        NULL        NULL
2006-01-05 00:00:00.000  NULL        NULL        NULL
...
2006-06-29 00:00:00.000  NULL        NULL        NULL
2006-06-30 00:00:00.000  NULL        NULL        NULL
2006-07-01 00:00:00.000  NULL        NULL        NULL
2006-07-02 00:00:00.000  NULL        NULL        NULL
2006-07-03 00:00:00.000  NULL        NULL        NULL
2006-07-04 00:00:00.000  10248       85          5
2006-07-05 00:00:00.000  10249       79          6
2006-07-06 00:00:00.000  NULL        NULL        NULL
2006-07-07 00:00:00.000  NULL        NULL        NULL
2006-07-08 00:00:00.000  10250       34          4
2006-07-08 00:00:00.000  10251       84          3
2006-07-09 00:00:00.000  10252       76          4
2006-07-10 00:00:00.000  10253       34          3
2006-07-11 00:00:00.000  10254       14          5
2006-07-12 00:00:00.000  10255       68          9
2006-07-13 00:00:00.000  NULL        NULL        NULL
2006-07-14 00:00:00.000  NULL        NULL        NULL
2006-07-15 00:00:00.000  10256       88          3
2006-07-16 00:00:00.000  10257       35          4
...
2008-12-27 00:00:00.000  NULL        NULL        NULL
2008-12-28 00:00:00.000  NULL        NULL        NULL
2008-12-29 00:00:00.000  NULL        NULL        NULL
2008-12-30 00:00:00.000  NULL        NULL        NULL
2008-12-31 00:00:00.000  NULL        NULL        NULL

(1446 row(s) affected)
```

Order dates that do not appear in the Orders table appear in the output of the query with NULLs in the order attributes.

Filtering Attributes from the Nonpreserved Side of an Outer Join

When you need to review code involving outer joins to look for logical bugs, one of the things you should examine is the WHERE clause. If the predicate in the WHERE clause refers to an attribute from the nonpreserved side of the join using an expression in the form <attribute> <operator> <value>, it's usually an indication of a bug. This is because attributes from the nonpreserved side of the join are NULLs in outer rows, and an expression in the form NULL <operator> <value> yields UNKNOWN (unless it's the IS NULL operator explicitly looking for NULLs). Recall that a WHERE clause filters UNKNOWN out. Such a predicate in

the WHERE clause causes all outer rows to be filtered out, effectively nullifying the outer join. In other words, it's as if the join type logically becomes an inner join. So the programmer either made a mistake in the choice of the join type, or made a mistake in the predicate. If this is not clear yet, the following example might help. Consider the following query:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderdate >= '20070101';
```

The query performs a left outer join between the Customers and Orders tables. Prior to applying the WHERE filter, the join operator returns inner rows for customers who placed orders, and outer rows for customers who didn't place orders, with NULLs in the order attributes. The predicate O.orderdate >= '20070101' in the WHERE clause evaluates to UNKNOWN for all outer rows because those have a NULL in the O.orderdate attribute. All outer rows are eliminated by the WHERE filter, as you can see in the output of the query, shown here in abbreviated form:

```
custid      companyname       orderid     orderdate
----------- ----------------- ----------- -----------------------
19          Customer RFNQC    10400       2007-01-01 00:00:00.000
65          Customer NYUHS    10401       2007-01-01 00:00:00.000
20          Customer THHDP    10402       2007-01-02 00:00:00.000
20          Customer THHDP    10403       2007-01-03 00:00:00.000
49          Customer CQRAA    10404       2007-01-03 00:00:00.000
...
58          Customer AHXHT    11073       2008-05-05 00:00:00.000
73          Customer JMIKW    11074       2008-05-06 00:00:00.000
68          Customer CCKOT    11075       2008-05-06 00:00:00.000
9           Customer RTXGC    11076       2008-05-06 00:00:00.000
65          Customer NYUHS    11077       2008-05-06 00:00:00.000

(678 row(s) affected)
```

This means that the use of an outer join here was futile. The programmer either made a mistake in using an outer join or made a mistake in the WHERE predicate.

Using Outer Joins in a Multi-Table Join

Recall the discussion about all-at-once operations in Chapter 2, "Single Table Queries." The concept means that all expressions that appear in the same logical query processing phase are logically evaluated at the same point in time. However, this concept is not applicable to the processing of table operators in the FROM phase. Table operators are logically evaluated from left to right. Rearranging the order in which outer joins are processed might result in different output, so you cannot rearrange them at will.

Some interesting logical bugs have to do with the logical order in which outer joins are processed. For example, a common logical bug involving outer joins could be considered a variation of the bug in the previous section. Suppose that you write a multi-table join query with an outer join between two tables, followed by an inner join with a third table. If the predicate in the inner join's ON clause compares an attribute from the nonpreserved side of the outer join and an attribute from the third table, all outer rows are filtered out. Remember that outer rows have NULLs in the attributes from the nonpreserved side of the join, and comparing a NULL with anything yields UNKNOWN, and UNKNOWN is filtered out by the ON filter. In other words, such a predicate would nullify the outer join and logically it would be as if you specified an inner join. For example, consider the following query:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

The first join is an outer join returning customers and their orders and also customers who did not place any orders. The outer rows representing customers with no orders have NULLs in the order attributes. The second join matches order lines from the OrderDetails table with rows from the result of the first join based on the predicate O.orderid = OD.orderid; however, in the rows representing customers with no orders, the O.orderid attribute is NULL. Therefore, the predicate evaluates to UNKNOWN and those rows are filtered out. The output shown here in abbreviated form doesn't contain the customers 22 and 57, the two customers who did not place orders:

```
custid      orderid     productid   qty
----------- ----------- ----------- ------
85          10248       11          12
85          10248       42          10
85          10248       72          5
79          10249       14          9
79          10249       51          40
...
65          11077       64          2
65          11077       66          1
65          11077       73          2
65          11077       75          4
65          11077       77          2

(2155 row(s) affected)
```

To generalize the problem: outer rows are nullified whenever any kind of outer join (left, right, or full) is followed by a subsequent inner join or right outer join. That's assuming, of course, that the join condition compares the NULLs from the left side with something from the right side.

You have several ways to get around the problem if you want to return customers with no orders in the output. One option is to use a left outer join in the second join as well:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  LEFT OUTER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

This way, the outer rows produced by the first join aren't filtered out, as you can see in the output shown here in abbreviated form:

```
custid      orderid     productid   qty
----------- ----------- ----------- ------
85          10248       11          12
85          10248       42          10
85          10248       72          5
79          10249       14          9
79          10249       51          40
...
65          11077       64          2
65          11077       66          1
65          11077       73          2
65          11077       75          4
65          11077       77          2
22          NULL        NULL        NULL
57          NULL        NULL        NULL

(2157 row(s) affected)
```

A second option is to first join Orders and OrderDetails using an inner join, and then join to the Customers table using a right outer join:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
  RIGHT OUTER JOIN Sales.Customers AS C
     ON O.custid = C.custid;
```

This way, the outer rows are produced by the last join, and are not filtered out.

A third option is to use parentheses to make the inner join between Orders and OrderDetails become an independent logical phase. This way you can apply a left outer join between the Customers table and the result of the inner join between Orders and OrderDetails. The query would look like this:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN
```

```
      (Sales.Orders AS O
         JOIN Sales.OrderDetails AS OD
           ON O.orderid = OD.orderid)
    ON C.custid = O.custid;
```

Using the COUNT Aggregate with Outer Joins

Another common logical bug involves using COUNT with outer joins. When you group the result of an outer join and use the *COUNT(*)* aggregate, the aggregate takes into consideration both inner rows and outer rows because it counts rows regardless of their contents. Usually, you're not supposed to take outer rows into consideration for the purposes of counting. For example, the following query is supposed to return the count of orders for each customer:

```
SELECT C.custid, COUNT(*) AS numorders
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
GROUP BY C.custid;
```

However, the *COUNT(*)* aggregate counts rows regardless of their meaning or contents, and customers who did not place orders—like 22 and 57—each have an outer row in the result of the join. As you can see in the output of the query shown here in abbreviated form, both 22 and 57 show up with a count of 1, while the number of orders they place is actually 0:

```
custid      numorders
----------- -----------
1           6
2           4
3           7
4           13
5           18
...
22          1
...
57          1
...
87          15
88          9
89          14
90          7
91          7

(91 row(s) affected)
```

The *COUNT(*)* aggregate function cannot detect whether a row really represents an order. To fix the problem you should use COUNT(<column>) instead of *COUNT(*)*, and provide a column from the nonpreserved side of the join. This way, the *COUNT()* aggregate ignores

outer rows because they have a NULL in that column. Remember to use a column that can only be NULL in case the row is an outer row—for example, the primary key column orderid:

```
SELECT C.custid, COUNT(O.orderid) AS numorders
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
GROUP BY C.custid;
```

Notice in the output shown here in abbreviated form that the customers 22 and 57 now show up with a count of 0:

```
custid      numorders
----------- -----------
1           6
2           4
3           7
4           13
5           18
...
22          0
...
57          0
...
87          15
88          9
89          14
90          7
91          7

(91 row(s) affected)
```

Conclusion

This chapter covered the join table operator. It described the logical query processing phases involved in the three fundamental types of joins—cross, inner, and outer. The chapter also covered further join examples including composite joins, non-equi joins, and multi-table joins. The chapter concluded with an optional reading section covering more advanced aspects of outer joins. To practice what you've learned, go over the exercises for this chapter.

Exercises

This section provides exercises to help you familiarize yourself with the subjects discussed in this chapter. All exercises involve querying objects in the TSQLFundamentals2008 database.

1-1

Run the following code to create the dbo.Nums auxiliary table in the TSQLFundamentals2008 database:

```
SET NOCOUNT ON;
USE TSQLFundamentals2008;
IF OBJECT_ID('dbo.Nums', 'U') IS NOT NULL DROP TABLE dbo.Nums;
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);

DECLARE @i AS INT = 1;
BEGIN TRAN
  WHILE @i <= 100000
  BEGIN
    INSERT INTO dbo.Nums VALUES(@i);
    SET @i = @i + 1;
  END
COMMIT TRAN
SET NOCOUNT OFF;
```

1-2

Write a query that generates five copies out of each employee row.

Tables involved: HR.Employees, and dbo.Nums tables.

Desired output:

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

```
7           Russell    King               3
8           Maria      Cameron            3
9           Zoya       Dolgopyatova       3
1           Sara       Davis              4
2           Don        Funk               4
3           Judy       Lew                4
4           Yael       Peled              4
5           Sven       Buck               4
6           Paul       Suurs              4
7           Russell    King               4
8           Maria      Cameron            4
9           Zoya       Dolgopyatova       4
1           Sara       Davis              5
2           Don        Funk               5
3           Judy       Lew                5
4           Yael       Peled              5
5           Sven       Buck               5
6           Paul       Suurs              5
7           Russell    King               5
8           Maria      Cameron            5
9           Zoya       Dolgopyatova       5

(45 row(s) affected)
```

1-3 (Optional, Advanced)

Write a query that returns a row for each employee and day in the range June 12, 2009 – June 16, 2009.

Tables involved: HR.Employees, and dbo.Nums tables.

Desired output:

```
empid       dt
----------- -----------------------
1           2009-06-12 00:00:00.000
1           2009-06-13 00:00:00.000
1           2009-06-14 00:00:00.000
1           2009-06-15 00:00:00.000
1           2009-06-16 00:00:00.000
2           2009-06-12 00:00:00.000
2           2009-06-13 00:00:00.000
2           2009-06-14 00:00:00.000
2           2009-06-15 00:00:00.000
2           2009-06-16 00:00:00.000
3           2009-06-12 00:00:00.000
3           2009-06-13 00:00:00.000
3           2009-06-14 00:00:00.000
3           2009-06-15 00:00:00.000
3           2009-06-16 00:00:00.000
4           2009-06-12 00:00:00.000
4           2009-06-13 00:00:00.000
4           2009-06-14 00:00:00.000
4           2009-06-15 00:00:00.000
4           2009-06-16 00:00:00.000
5           2009-06-12 00:00:00.000
5           2009-06-13 00:00:00.000
```

```
5           2009-06-14 00:00:00.000
5           2009-06-15 00:00:00.000
5           2009-06-16 00:00:00.000
6           2009-06-12 00:00:00.000
6           2009-06-13 00:00:00.000
6           2009-06-14 00:00:00.000
6           2009-06-15 00:00:00.000
6           2009-06-16 00:00:00.000
7           2009-06-12 00:00:00.000
7           2009-06-13 00:00:00.000
7           2009-06-14 00:00:00.000
7           2009-06-15 00:00:00.000
7           2009-06-16 00:00:00.000
8           2009-06-12 00:00:00.000
8           2009-06-13 00:00:00.000
8           2009-06-14 00:00:00.000
8           2009-06-15 00:00:00.000
8           2009-06-16 00:00:00.000
9           2009-06-12 00:00:00.000
9           2009-06-13 00:00:00.000
9           2009-06-14 00:00:00.000
9           2009-06-15 00:00:00.000
9           2009-06-16 00:00:00.000

(45 row(s) affected)
```

2

Return U.S. customers, and for each customer return the total number of orders and total quantities.
Tables involved: Sales.Customers, Sales.Orders, and Sales.OrderDetails tables.
Desired output:

```
custid      numorders   totalqty
----------- ----------- -----------
32          11          345
36          5           122
43          2           20
45          4           181
48          8           134
55          10          603
65          18          1383
71          31          4958
75          9           327
77          4           46
78          3           59
82          3           89
89          14          1063

(13 row(s) affected)
```

3

Return customers and their orders including customers who placed no orders.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output (abbreviated):

```
custid      companyname     orderid     orderdate
----------- --------------- ----------- -----------------------
85          Customer ENQZT  10248       2006-07-04 00:00:00.000
79          Customer FAPSM  10249       2006-07-05 00:00:00.000
34          Customer IBVRG  10250       2006-07-08 00:00:00.000
84          Customer NRCSK  10251       2006-07-08 00:00:00.000
...
73          Customer JMIKW  11074       2008-05-06 00:00:00.000
68          Customer CCKOT  11075       2008-05-06 00:00:00.000
9           Customer RTXGC  11076       2008-05-06 00:00:00.000
65          Customer NYUHS  11077       2008-05-06 00:00:00.000
22          Customer DTDMN  NULL        NULL
57          Customer WVAXS  NULL        NULL

(832 row(s) affected)
```

4

Return customers who placed no orders.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output:

```
custid      companyname
----------- ---------------
22          Customer DTDMN
57          Customer WVAXS

(2 row(s) affected)
```

5

Return customers with orders placed on Feb 12, 2007 along with their orders.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output:

```
custid      companyname     orderid     orderdate
----------- --------------- ----------- -----------------------
66          Customer LHANT  10443       2007-02-12 00:00:00.000
5           Customer HGVLZ  10444       2007-02-12 00:00:00.000

(2 row(s) affected)
```

6 (Optional, Advanced)

Return customers with orders placed on Feb 12, 2007 along with their orders. Also return customers who didn't place orders on Feb 12, 2007.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output (abbreviated):

```
custid      companyname       orderid     orderdate
----------- ----------------- ----------- -----------------------
72          Customer AHPOP    NULL        NULL
58          Customer AHXHT    NULL        NULL
25          Customer AZJED    NULL        NULL
18          Customer BSVAR    NULL        NULL
91          Customer CCFIZ    NULL        NULL
...
33          Customer FVXPQ    NULL        NULL
53          Customer GCJSG    NULL        NULL
39          Customer GLLAG    NULL        NULL
16          Customer GYBBY    NULL        NULL
4           Customer HFBZG    NULL        NULL
5           Customer HGVLZ    10444       2007-02-12 00:00:00.000
42          Customer IAIJK    NULL        NULL
34          Customer IBVRG    NULL        NULL
63          Customer IRRVL    NULL        NULL
73          Customer JMIKW    NULL        NULL
15          Customer JUWXK    NULL        NULL
...
21          Customer KIDPX    NULL        NULL
30          Customer KSLQF    NULL        NULL
55          Customer KZQZT    NULL        NULL
71          Customer LCOUJ    NULL        NULL
77          Customer LCYBZ    NULL        NULL
66          Customer LHANT    10443       2007-02-12 00:00:00.000
38          Customer LJUCA    NULL        NULL
59          Customer LOLJO    NULL        NULL
36          Customer LVJSO    NULL        NULL
64          Customer LWGMD    NULL        NULL
29          Customer MDLWA    NULL        NULL
...

(91 row(s) affected)
```

7 (Optional, Advanced)

Return all customers, and for each return a Yes/No value depending on whether the customer placed an order on Feb 12, 2007.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output (abbreviated):

```
custid      companyname       HasOrderOn20070212
----------- ----------------- ------------------
1           Customer NRZBB    No
2           Customer MLTDN    No
3           Customer KBUDE    No
```

```
4           Customer HFBZG    No
5           Customer HGVLZ    Yes
6           Customer XHXJV    No
7           Customer QXVLA    No
8           Customer QUHWH    No
9           Customer RTXGC    No
10          Customer EEALV    No
...

(91 row(s) affected)
```

Solutions

This section provides solutions to the exercises for this chapter.

1-2

Producing multiple copies of rows can be achieved with a fundamental technique that utilizes a cross join. If you need to produce five copies out of each employee row, you need to perform a cross join between the Employees table and a table that has five rows; alternatively, you can perform a cross join between Employees and a table that has more than five rows, but filter only five from that table in the WHERE clause. The Nums table is very convenient for this purpose. Simply cross Employees and Nums, and filter from Nums as many rows as the number of requested copies (five in this case). Here's the solution query:

```
SELECT E.empid, E.FirstName, E.LastName, Nums.n
FROM HR.Employees AS E
  CROSS JOIN dbo.Nums
WHERE Nums.n <= 5
ORDER BY n, empid;
```

1-3

This exercise is an extension of the previous exercise. Instead of being asked to produce a predetermined constant number of copies out of each employee row, you are asked to produce a copy for each day in a certain date range. So here you need to calculate the number of days in the requested date range using the *DATEDIFF* function, and refer to the result of that expression in the query's WHERE clause instead of referring to a constant. To produce the dates, simply add *n* - 1 days to the date that starts the requested range. Here's the solution query:

```
SELECT E.empid,
  DATEADD(day, D.n - 1, '20090612') AS dt
FROM HR.Employees AS E
  CROSS JOIN dbo.Nums AS D
WHERE D.n <= DATEDIFF(day, '20090612', '20090616') + 1
ORDER BY empid, dt;
```

The *DATEDIFF* function returns 4 because there is a four-day difference between June 12, 2009 and June 16, 2009. Add 1 to the result, and you get 5 for the five days in the range. So the WHERE clause filters five rows from Nums where *n* is smaller than or equal to 5. By adding *n* - 1 days to June 12, 2009, you get all dates in the range June 12, 2009 and June 16, 2009.

2

This exercise requires you to write a query that joins three tables: Customers, Orders, and OrderDetails. The query should filter in the WHERE clause only rows where the customer's country is USA. Because you are asked to return aggregates per customer, the query should group the rows by customer ID. You need to resolve a tricky issue here to return the right number of orders for each customer. Because of the join between Orders and OrderDetails, you don't get only one row per order—you get one row per order line. So if you use the *COUNT(*)* function in the SELECT list, you get back the number of order lines for each customer and not the number of orders. To resolve this issue, you need to take each order into consideration only once. You can do this by using COUNT(DISTINCT O.orderid) instead of *COUNT(*)*. The total quantities don't create any special issues because the quantity is associated with the order line and not the order. Here's the solution query:

```
SELECT C.custid, COUNT(DISTINCT O.orderid) AS numorders, SUM(OD.qty) AS totalqty
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON O.custid = C.custid
  JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
WHERE C.country = N'USA'
GROUP BY C.custid;
```

3

To get both customers who placed orders and customers who didn't place orders in the result, you need to use an outer join like so:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid;
```

This query returns 832 rows (including the customers 22 and 57, who didn't place orders). An inner join between the tables would return only 830 rows without these customers.

4

This exercise is an extension of the previous one. To return only customers who didn't place orders, you need to add a WHERE clause to the query that filters only outer rows; namely, rows

that represent customers with no orders. Outer rows have NULLs in the attributes from the nonpreserved side of the join (Orders). But to make sure that the NULL is a placeholder for an outer row and not a NULL that originated from the table, it is recommended that you refer to an attribute that is the primary key, or the join column, or one defined as not allowing NULLs. Here's the solution query referring to the primary key of the Orders table in the WHERE clause:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid
WHERE O.orderid IS NULL;
```

This query returns only two rows for the customers 22 and 57, who didn't place orders.

5

This exercise involves writing a query that performs an inner join between Customers and Orders, and filters only rows where the order date is February 12, 2007:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON O.custid = C.custid
WHERE O.orderdate = '20070212';
```

The WHERE clause filtered out Customers who didn't place orders on February 12, 2007, but that was the request.

6

This exercise builds on the previous one. The trick here is to realize two things. First, you need an outer join because you are supposed to return customers who do not meet a certain criteria. Second, the filter on the order date must appear in the ON clause and not the WHERE clause. Remember that the WHERE filter is applied after outer rows are added and is final. Your goal is to match orders to customers only if the order was placed by the customer and on February 12, 2007. You still want to get customers who didn't place orders on that date in the output; in other words, the filter on the order date should only determine matches and not be considered final in regards to the customer rows. Hence the ON clause should match customers and orders based on both an equality between the customer's customer ID and the order's customer ID, and the order date being February 12, 2007. Here's the solution query:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid
    AND O.orderdate = '20070212';
```

7

This exercise is an extension of the previous exercise. Here, instead of returning matching orders, you just need to return a Yes/No value indicating whether there is a matching order. Remember that in an outer join a nonmatch is identified as an outer row with NULLs in the attributes of the nonpreserved side. So you can use a simple CASE expression that checks whether the current row is an outer one, in which case it returns 'Yes'; otherwise, it returns 'No'. Because technically you can have more than one match per customer, you should add a DISTINCT clause to the SELECT list. This way you get only one row back for each customer. Here's the solution query:

```
SELECT DISTINCT C.custid, C.companyname,
  CASE WHEN O.orderid IS NOT NULL THEN 'Yes' ELSE 'No' END AS [HasOrderOn20070212]
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid
    AND O.orderdate = '20070212';
```