Inside Microsoft SQL Server 2005: T-SQL Querying

by Itzik Ben-Gan, Lubor Kollar and Dejan Sarka
Microsoft Press. (c) 2006.

Chapter 9: Graphs, Trees, Hierarchies, and Recursive Queries

Overview

This chapter covers treatment of specialized data structures called graphs, trees, and hierarchies in Microsoft SQL Server using T-SQL. Of the three, probably the most commonly used among T-SQL programmers is *hierarchy*, and this term is sometimes used even when the data structure involved is not really a hierarchy. I'll start with a terminology section describing each data structure to clear the confusion.

Treatment (representation, maintenance, and manipulation) of graphs, trees, and hierarchies in an RDBMS is far from trivial. I'll discuss two main approaches, one based on iterative/recursive logic, and another based on materializing extra information in the database that describes the data structure.

Interestingly, even though these data structures have been and still are commonly implemented in relational database management systems (RDBMSs), support for recursive queries was only introduced in the standard ANSI SQL:1999. SQL Server 2005 for the first time adopted to some extent the ANSI SQL:1999 recursive querying extensions in T-SQL.

In this chapter, I'll cover solutions that use the new recursive queries in SQL Server 2005, as well as solutions that are applicable in earlier versions of SQL Server.

> **Tip** I also urge you to look up Vadim Tropashko's Nested Intervals model at http://www.dbazine.com. It is a beautiful model, very interesting intellectually, and Vadim covers practical issues such as implementation and performance. However, I find Vadim's model to be substantially more complex than most mere mortals (including me) can grasp in full, so I won't cover it here. The solutions I will cover here, on the other hand, will be fairly simple to understand and implement by experienced T-SQL programmers. Before you make an attempt at reading Vadim's stuff, make sure you have enough coffee and enough hours of sleep.

As promised, I'll start with a terminology section describing graphs, trees, and hierarchies.

Terminology

> **Note** The explanations in this section are based on definitions from the National Institute of Standards and Technology (NIST). I made some revisions and added narrative to the original definitions to make them less formal and keep relevance to the subject area (T-SQL). For more complete and formal definitions of graphs, trees, and related terms, please refer to: http://www.nist.gov/dads/

Graphs

A graph is a set of items connected by *edges*. Each item is called a *vertex* or *node*. An edge is a connection between two vertices of a graph.

A *graph* is a catch-all term for a data structure, and many scenarios can be represented as graphs—for example, employee organizational charts, bills of materials (BOMs), road systems, and so on. To narrow down the type of graph to a more specific case, you need to identify its properties:

- **Directed/Undirected** In a directed graph (also known as a *digraph*), there's a direction or order to the two vertices of an edge. For example, in a BOM graph for coffee-shop products, Latte contains Milk and not the other way around. There's an edge (containment relationship) in the graph for the pair of vertices/items (Latte, Milk), but no edge for the pair (Milk, Latte).

  In an undirected graph, each edge simply connects two vertices, with no particular order. For example, in a road system graph there's a road between Los Angeles and San Francisco. The edge (road) between the vertices (cities) Los Angeles and San Francisco can be expressed as either of the following: {Los Angeles, San Francisco} or {San Francisco, Los Angeles}.

- **Acyclic** An acyclic graph is a graph with no cycle—that is, no *path* that starts and ends at the same vertex—for example, employee organizational charts and BOMs. A directed acyclic graph is also known as a *DAG*.

  If there are paths that start and end at the same vertex—as there usually are in road systems—the graph is not acyclic.

- **Connected** A connected graph is a graph where there's a path between every pair of vertices—for example, employee organizational charts.

Trees

A tree is a special case of a graph—a connected, acyclic graph.

A *rooted tree* is accessed beginning at the *root* node. Each node is either a *leaf* or an *internal node*. An internal node has one or more *child* nodes and is called the *parent* of its child nodes. All children of the same node are *siblings*. Contrary to the appearance in a physical tree, the root is usually depicted at the top of the structure and the leaves are depicted at the bottom. (See Figure 9-1.)

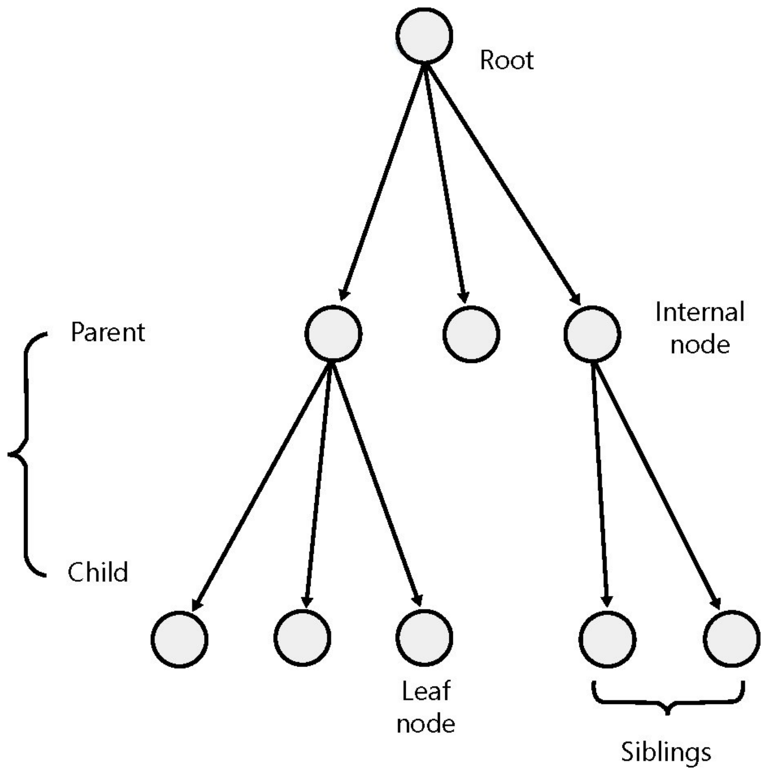

Figure 9-1: A tree

A *forest* is a collection of one or more trees—for example, forum discussions can be represented as a forest where each thread is a tree.

Hierarchies

Some scenarios can be described as a *hierarchy* and modeled as an directed acyclic graph—for example, inheritance among types/classes in object-oriented programming and reports-to relationships in an employee organizational chart. In the former, the edges of the graph locate the inheritance. Classes can inherit methods and properties from other classes (and possibly from multiple classes). In the latter, the edges represent the reports-to relationship between employees. Notice the acyclic, directed nature of these scenarios. The management chain of responsibility in a company cannot go around in circles, for example.

Scenarios

Throughout the chapter, I will use three scenarios: Employee Organizational Chart (tree, hierarchy), Bill Of Materials or BOM (DAG), and Road System (undirected cyclic graph). Note what distinguishes a (directed) tree from a DAG. All trees are DAGs, but not all DAGs are trees. In a tree, an item can have at most one parent; in some management hierarchies, an employee can have more than one manager.

Employee Organizational Chart

The employee organizational chart that I will use is depicted graphically in Figure 9-2.

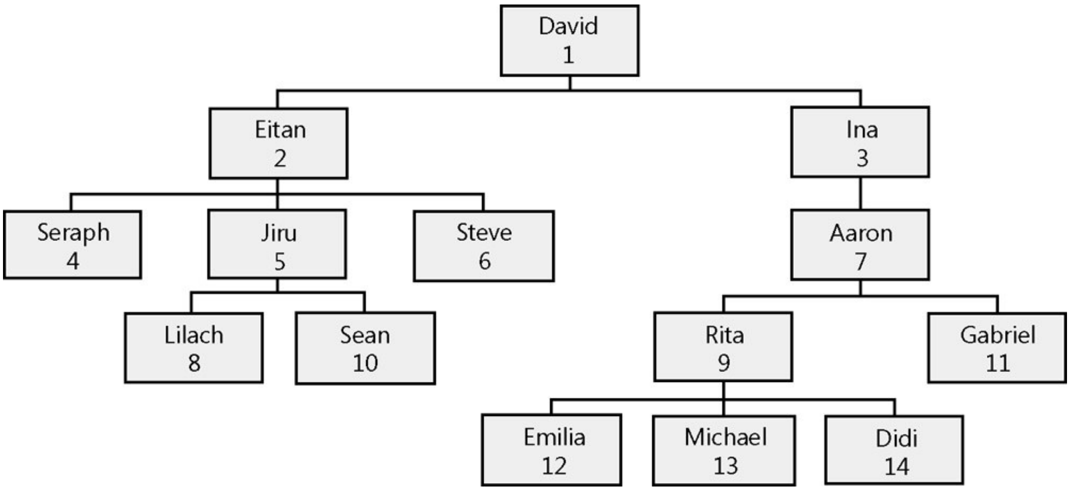

Figure 9-2: Employee Organizational Chart

To create the Employees table and populate it with sample data, run the code in Listing 9-1. The contents of the Employees table are shown in Table 9-1.

Table 9-1: Contents of Employees Table

empid	mgrid	empname	salary
1	NULL	David	10000.0000
2	1	Eitan	7000.0000
3	1	Ina	7500.0000
4	2	Seraph	5000.0000
5	2	Jiru	5500.0000
6	2	Steve	4500.0000
7	3	Aaron	5000.0000
8	5	Lilach	3500.0000
9	7	Rita	3000.0000
10	5	Sean	3000.0000
11	7	Gabriel	3000.0000
12	9	Emilia	2000.0000
13	9	Michael	2000.0000
14	9	Didi	1500.0000

Listing 9-1: Data definition language and sample data for the Employees table

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Employees') IS NOT NULL
  DROP TABLE dbo.Employees;
GO
CREATE TABLE dbo.Employees
(
   empid   INT         NOT NULL PRIMARY KEY,
   mgrid   INT         NULL REFERENCES dbo.Employees,
   empname VARCHAR(25) NOT NULL,
   salary  MONEY       NOT NULL,
   CHECK (empid <> mgrid)
);

INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
```

```
  VALUES(1, NULL, 'David', $10000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(2, 1, 'Eitan', $7000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(3, 1, 'Ina', $7500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(4, 2, 'Seraph', $5000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(5, 2, 'Jiru', $5500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(6, 2, 'Steve', $4500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(7, 3, 'Aaron', $5000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(8, 5, 'Lilach', $3500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(9, 7, 'Rita', $3000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(10, 5, 'Sean', $3000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(11, 7, 'Gabriel', $3000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(12, 9, 'Emilia' , $2000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(13, 9, 'Michael', $2000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(14, 9, 'Didi', $1500.00);

CREATE UNIQUE INDEX idx_unc_mgrid_empid ON dbo.Employees(mgrid, empid);
```

The Employees table represents a management hierarchy as an adjacency list, where the manager and employee represent the parent and child nodes, respectively.

Bill of Materials (BOM)

I will use a BOM of coffee shop products, which is depicted graphically in Figure 9-3.

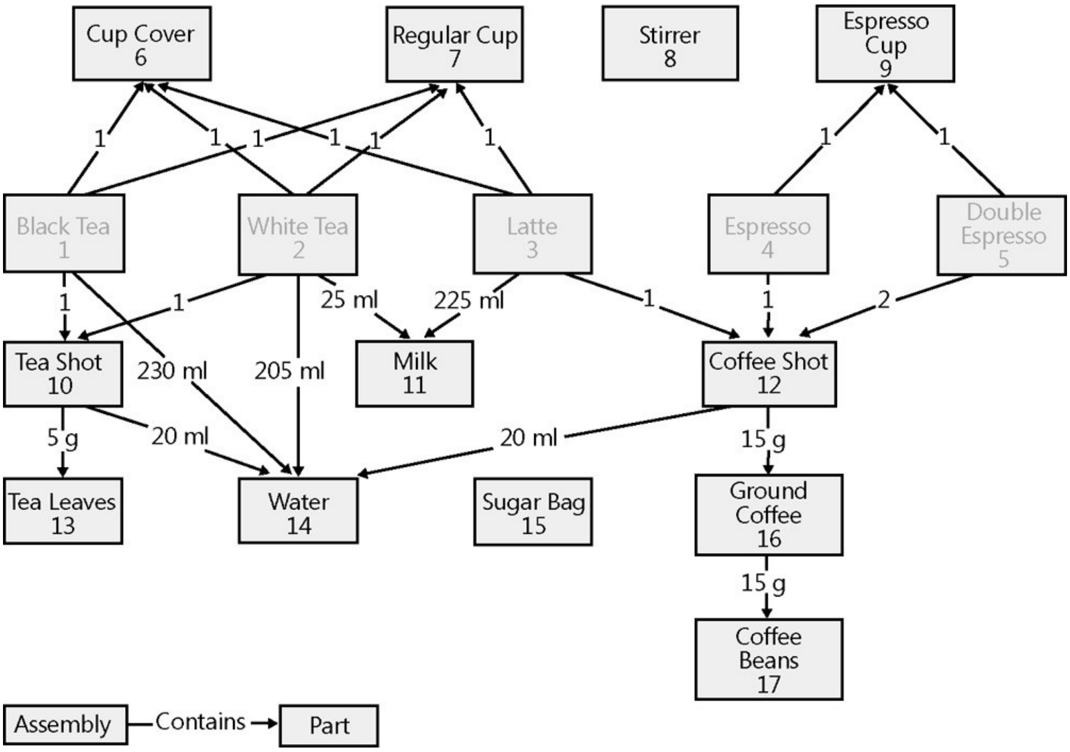

Figure 9-3: Bill of Materials (BOM)

To create the Parts and BOM tables and populate them with sample data, run the code in Listing 9-2. The contents of the Parts and BOM tables are shown in Tables 9-2 and 9-3.

Table 9-2: Contents of Parts Table

partid	partname
1	Black Tea
2	White Tea
3	Latte
4	Espresso
5	Double Espresso
6	Cup Cover
7	Regular Cup
8	Stirrer
9	Espresso Cup
10	Tea Shot
11	Milk
12	Coffee Shot
13	Tea Leaves
14	Water
15	Sugar Bag
16	Ground Coffee
17	Coffee Beans

Table 9-3: Contents of BOM Table

partid	assemblyid	unit	qty
1	NULL	EA	1.00
2	NULL	EA	1.00
3	NULL	EA	1.00
4	NULL	EA	1.00
5	NULL	EA	1.00
6	1	EA	1.00
7	1	EA	1.00
10	1	EA	1.00
14	1	mL	230.00
6	2	EA	1.00
7	2	EA	1.00
10	2	EA	1.00
14	2	mL	205.00
11	2	mL	25.00
6	3	EA	1.00
7	3	EA	1.00
11	3	mL	225.00
12	3	EA	1.00
9	4	EA	1.00
12	4	EA	1.00

9	5	EA	1.00
12	5	EA	2.00
13	10	g	5.00
14	10	mL	20.00
14	12	mL	20.00
16	12	g	15.00
17	16	g	15.00

Notice that the first scenario (employee organizational chart) requires only one table because it is modeled as a tree; both an edge (manager, employee) and a vertex (employee) can be represented by the same row. The BOM scenario requires two tables because it is modeled as a DAG, where multiple paths can lead to each node; an edge (assembly, part) is represented by a row in the BOM table, and a vertex (part) is represented by a row in the Parts table.

Listing 9-2: Data definition language and sample data for the Parts and BOM tables

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.BOM') IS NOT NULL
  DROP TABLE dbo.BOM;
GO
IF OBJECT_ID('dbo.Parts') IS NOT NULL
  DROP TABLE dbo.Parts;

GO
CREATE TABLE dbo.Parts
(
    partid   INT         NOT NULL PRIMARY KEY,
    partname VARCHAR(25) NOT NULL
);

INSERT INTO dbo.Parts(partid, partname) VALUES( 1, 'Black Tea');
INSERT INTO dbo.Parts(partid, partname) VALUES( 2, 'White Tea');
INSERT INTO dbo.Parts(partid, partname) VALUES( 3, 'Latte');
INSERT INTO dbo.Parts(partid, partname) VALUES( 4, 'Espresso');
INSERT INTO dbo.Parts(partid, partname) VALUES( 5, 'Double Espresso');
INSERT INTO dbo.Parts(partid, partname) VALUES( 6, 'Cup Cover');
INSERT INTO dbo.Parts(partid, partname) VALUES( 7, 'Regular Cup');
INSERT INTO dbo.Parts(partid, partname) VALUES( 8, 'Stirrer');
INSERT INTO dbo.Parts(partid, partname) VALUES( 9, 'Espresso Cup');
INSERT INTO dbo.Parts(partid, partname) VALUES(10, 'Tea Shot');
INSERT INTO dbo.Parts(partid, partname) VALUES(11, 'Milk');
INSERT INTO dbo.Parts(partid, partname) VALUES(12, 'Coffee Shot');
INSERT INTO dbo.Parts(partid, partname) VALUES(13, 'Tea Leaves');
INSERT INTO dbo.Parts(partid, partname) VALUES(14, 'Water');
INSERT INTO dbo.Parts(partid, partname) VALUES(15, 'Sugar Bag');
INSERT INTO dbo.Parts(partid, partname) VALUES(16, 'Ground Coffee');
INSERT INTO dbo.Parts(partid, partname) VALUES(17, 'Coffee Beans');

CREATE TABLE dbo.BOM
(
    partid     INT           NOT NULL REFERENCES dbo.Parts,
    assemblyid INT           NULL REFERENCES dbo.Parts,
    unit       VARCHAR(3)    NOT NULL,
    qty        DECIMAL(8, 2) NOT NULL,
    UNIQUE(partid, assemblyid),
    CHECK (partid <> assemblyid)
);

INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 1, NULL, 'EA',  1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 2, NULL, 'EA',  1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
```

```
  VALUES( 3, NULL, 'EA',  1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 4, NULL, 'EA',  1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 5, NULL, 'EA',  1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 6, 1,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 7, 1,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(10, 1,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14, 1,   'mL',   230.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 6, 2,   'EA',   1.00);

INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 7, 2,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(10, 2,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14, 2,   'mL',   205.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(11, 2,   'mL',   25.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 6, 3,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 7, 3,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(11, 3,   'mL',   225.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(12, 3,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 9, 4,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(12, 4,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 9, 5,   'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(12, 5,   'EA',   2.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(13, 10,  'g' ,   5.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14, 10,  'mL',   20.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14, 12,  'mL',   20.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(16, 12,  'g' ,   15.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(17, 16,  'g' ,   15.00);
```

BOM represents a directed acyclic graph (DAG). It holds the parent and child node IDs in the *assemblyid* and *partid* attributes, respectively. BOM also represents a *weighted* graph, where a weight/number is associated with each edge. In our case, that weight is the *qty* attribute that holds the quantity of the part within the assembly (assembly of sub-parts). The unit attribute holds the unit of the *qty* (EA for each, g for gram, mL for milliliter, and so on).

Road System

The Road System that I will use is that of several major cities in the United States, and it is depicted graphically in Figure 9-4.

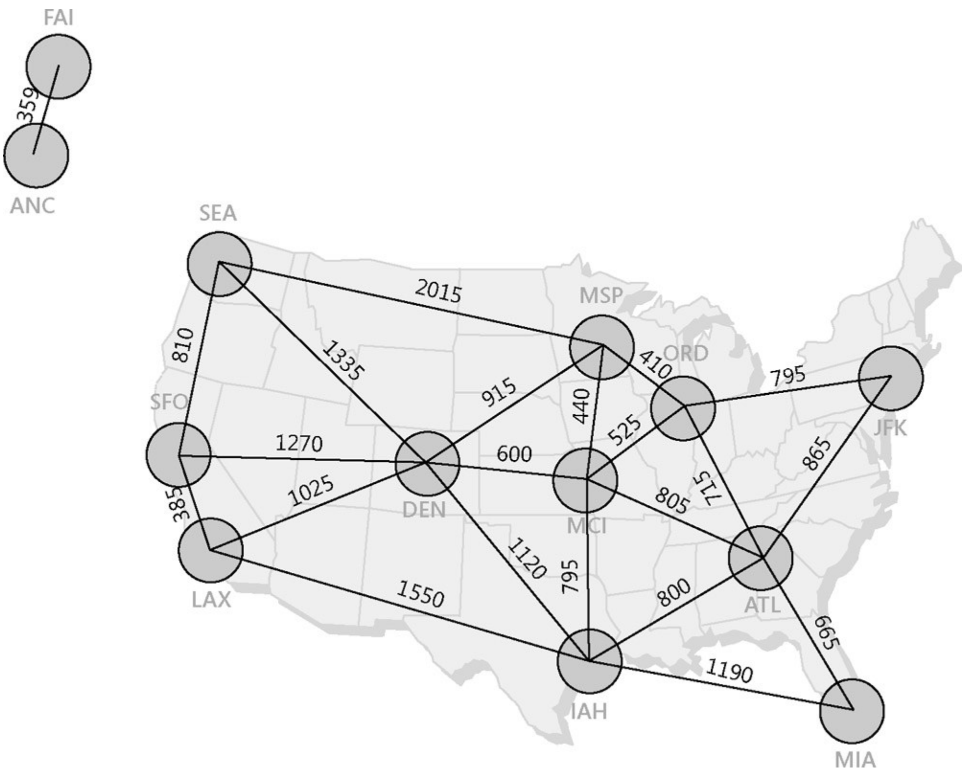

Figure 9-4: Road System

To create the Cities and Roads tables and populate them with sample data, run the code in Listing 9-3. The contents of the Cities and Roads tables are shown in Tables 9-4 and 9-5.

Table 9-4: Contents of Cities Table

cityid	city	region	country
ANC	Anchorage	AK	USA
ATL	Atlanta	GA	USA
DEN	Denver	CO	USA
FAI	Fairbanks	AK	USA
IAH	Houston	TX	USA
JFK	New York	NY	USA
LAX	Los Angeles	CA	USA
MCI	Kansas City	KS	USA
MIA	Miami	FL	USA
MSP	Minneapolis	MN	USA
ORD	Chicago	IL	USA
SEA	Seattle	WA	USA
SFO	San Francisco	CA	USA

Table 9-5: Contents of Roads Table

city1	city2	distance
ANC	FAI	359
ATL	IAH	800
ATL	JFK	865
ATL	MCI	805

ATL	MIA	665
ATL	ORD	715
DEN	IAH	1120
DEN	LAX	1025
DEN	MCI	600
DEN	MSP	915
DEN	SEA	1335
DEN	SFO	1270
IAH	LAX	1550
IAH	MCI	795
IAH	MIA	1190
JFK	ORD	795
LAX	SFO	385
MCI	MSP	440
MCI	ORD	525
MSP	ORD	410
MSP	SEA	2015
SEA	SFO	815

Listing 9-3: Data definition language and sample data for the Cities and Roads tables

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Roads') IS NOT NULL
  DROP TABLE dbo.Roads;

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

CREATE TABLE dbo.Cities
(
    cityid  CHAR(3)     NOT NULL PRIMARY KEY,
    city    VARCHAR(30) NOT NULL,
    region  VARCHAR(30) NULL,
    country VARCHAR(30) NOT NULL
);

INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('ATL', 'Atlanta', 'GA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('ORD', 'Chicago', 'IL', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('DEN', 'Denver', 'CO', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('IAH', 'Houston', 'TX', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('MCI', 'Kansas City', 'KS', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('LAX', 'Los Angeles', 'CA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('MIA', 'Miami', 'FL', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('MSP', 'Minneapolis', 'MN', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('JFK', 'New York', 'NY', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
```

```
   VALUE('SEA', 'Seattle', 'WA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('SFO', 'San Francisco', 'CA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('ANC', 'Anchorage', 'AK', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
   VALUE('FAI', 'Fairbanks', 'AK', 'USA');

CREATE TABLE dbo.Roads
(
  city1        CHAR(3) NOT NULL REFERENCES dbo.Cities,
  city2        CHAR(3) NOT NULL REFERENCES dbo.Cities,
  distance INT      NOT NULL,
  PRIMARY KEY(city1, city2),
  CHECK(city1 < city2),
  CHECK(distance > 0)
);

INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ANC', 'FAI', 359);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'ORD', 715);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'IAH', 800);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'MCI', 805);

INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'MIA', 665);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'JFK', 865);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'IAH', 1120);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'MCI', 600);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'LAX', 1025);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'MSP', 915);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'SEA', 1335);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'SFO', 1270);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('IAH', 'MCI', 795);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('IAH', 'LAX', 1550);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('IAH', 'MIA', 1190);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('JFK', 'ORD', 795);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('LAX', 'SFO', 385);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MCI', 'ORD', 525);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MCI', 'MSP', 440);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MSP', 'ORD', 410);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MSP', 'SEA', 2015);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('SEA', 'SFO', 815);
```

The Roads table represents an undirected cyclic weighted graph. Each edge (road) is represented by a row in the table. The attributes *city1 and city2* are two city IDs representing the nodes of the edge. The weight in this case is the distance attribute, which holds the distance between the cities in miles. Note that the Roads table has a CHECK constraint (*city1 < city2*) as part of its schema definition to reject attempts to enter the same edge twice (for example, {SEA, SFO} and {SFO, SEA}).

Having all the scenarios and sample data in place, let's go over the approaches to treatment of graphs, trees, and hierarchies. I'll cover three main approaches: iterative/recursive, materialized path, and nested sets.

Iteration/Recursion

Iterative approaches apply some form of loops or recursion. There are many iterative algorithms that traverse graphs. Some traverse graphs a node at a time and are usually implemented with cursors, but these are typically very slow. I will focus on algorithms that traverse graphs a level at a time using a combination of iterative or recursive logic and set-based queries. Given a set of nodes *U*, the *next level of subordinates* refers to the set *V*, which consists of the direct subordinates (children) of the nodes in U. In my experience, implementations of iterative algorithms that traverse a graph a level at a time perform much better than the ones that traverse a graph a node at a time.

There are several advantages to using iterative solutions rather than the other methods. First, you don't need to materialize any extra information describing the graph to the database besides the node IDs in the edges. In other words, there's no need to redesign your tables. The solutions traverse the graph by relying solely on the stored edge information—for example, (*mgrid, empid*), (*assemblyid, partid*), (*city1, city2*), and so on.

Second, most of the solutions that apply to trees also apply to the more generic digraphs. In other words, most solutions that apply to graphs where only one path can lead to a given node also apply to graphs where multiple paths may lead to a given node.

Finally, most of the solutions that I will describe in this section support a virtually unlimited number of levels.

I will use two main tools to implement solutions in my examples: user defined functions (UDFs) and recursive common table expressions (CTEs). UDFs have been available since SQL Server 2000, while CTEs were introduced in SQL Server 2005. When using UDFs, I'll rely on SQL Server 2000–compatible features only so that you will be able to implement the solutions in SQL Server 2000. Because CTEs are new to SQL Server 2005, I felt free to rely on other new T-SQL features (for example, using the ROW_NUMBER function). The core algorithms will be similar in both versions.

In my solutions, I focused on UDFs and CTEs, but note that in some cases when performance of a UDF or CTE is not satisfactory, you might get better performance by implementing a solution with a stored procedure. Stored procedures give you more control—for example, you can materialize and index interim sets in temporary tables, and so on.

However, I used UDFs and CTEs because I wanted to focus on the algorithms and the clarity of the solutions.

Subordinates

Let's start with a classical request to return subordinates; for example, return all subordinates of a given employee. More technically, you're after a subgraph/subtree of a given root in a directed graph (digraph). The iterative algorithm is very simple:

Input: *@root*

Algorithm:

- set @*lvl* = 0; insert into table @Subs row for *@root*

-; while there were rows in the previous level of employees:

- set @*lvl* = @*lvl* + *1*; insert into table @Subs rows for the next level (*mgrid* in (*empid* values in previous level))

- return @Subs

Run the code in Listing 9-4 to create the *fn_subordinates1* function, which implements this algorithm as a UDF.

Listing 9-4: Creation script for the *fn_subordinates1* function

```
---------------------------------------------------------------------
-- Function: fn_subordinates1, Descendants
--
-- Input   : @root INT: Manager id
--
-- Output  : @Subs Table: id and level of subordinates of
--                        input manager (empid = @root) in all levels
--
-- Process : * Insert into @Subs row of input manager
--           * In a loop, while previous insert loaded more than 0 rows
--             insert into @Subs next level of subordinates
---------------------------------------------------------------------
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_subordinates1') IS NOT NULL
  DROP FUNCTION dbo.fn_subordinates1;
GO
CREATE FUNCTION dbo.fn_subordinates1(@root AS INT) RETURNS @Subs TABLE
(
   empid INT NOT NULL PRIMARY KEY NONCLUSTERED,
   lvl   INT NOT NULL,
   UNIQUE CLUSTERED(lvl, empid) -- Index will be used to filter level
)
AS
BEGIN
```

```
  DECLARE @lvl AS INT;
  SET @lvl = 0;                 -- Initialize level counter with 0

  -- Insert root node into @Subs
  INSERT INTO @Subs(empid, lvl)
    SELECT empid, @lvl FROM dbo.Employees WHERE empid = @root;

  WHILE @@rowcount > 0          -- while previous level had rows
BEGIN
  SET @lvl = @lvl + 1;                 -- Increment level counter

  -- Insert next level of subordinates to @Subs
  INSERT INTO @Subs(empid, lvl)
     SELECT C.empid, @lvl
     FROM @Subs AS P                   -- P = Parent
       JOIN dbo.Employees AS C         -- C = Child
         ON P.lvl = @lvl - 1             -- Filter parents from previous level
         AND C.mgrid = P.empid;
END

  RETURN;
END
GO
```

The function accepts the *@root* input parameter, which is the ID of the requested subtree's root employee. The function returns the @Subs table variable, with all subordinates of employee with ID = *@root* in all levels. Besides containing the employee attributes, @Subs also has a column called *lvl* that keeps track of the level in the subtree (0 for the subtree's root, and increasing from there by 1 in each iteration).

The function's code keeps track of the current level being handled in the *@lvl* local variable, which is initialized with zero.

The function's code first inserts into @Subs the row from Employees where *empid* = *@root*.

Then in a loop, while the last insert affects more than zero rows, the code increments the *@lvl* variable's value by one and loads to @Subs the next level of employees—in other words, direct subordinates of the managers loaded in the previous level.

To load the next level of employees to @Subs, the query in the loop joins @Subs (representing managers) with Employees (representing subordinates).

The *lvl* column is important because it allows you to isolate the managers that were inserted into @Subs in the last iteration. To return only subordinates of the previously inserted managers, the join condition filters from @Subs only rows where the *lvl* column is equal to the previous level (*@lvl* − *1*).

To test the function, run the following code, which returns the subordinates of employee 3, as shown in Table 9-6:

```
SELECT empid, lvl FROM dbo.fn_subordinates1(3) AS S;
```

Table 9-6: Subtree of Employee 3, IDs Only

empid	lvl
3	0
7	1
9	2
11	2
12	3
13	3
14	3

You can verify that the output is correct by examining Figure 9-2 and following the subtree of the root employee (ID = 3).

To get other attributes of the employees besides just the employee ID, you can either rewrite the function and add those attributes to the @Subs table, or simply join the function with the Employees table like so:

```
SELECT E.empid, E.empname, S.lvl
FROM dbo.fn_subordinates1(3) AS S
  JOIN dbo.Employees AS E
    ON E.empid = S.empid;
```

You will get the output shown in Table 9-7.

Table 9-7: Subordinates of Employee 3, Including Employee Names

empid	empname	lvl
3	Ina	0
7	Aaron	1
9	Rita	2
11	Gabriel	2
12	Emilia	3
13	Michael	3
14	Didi	3

To limit the result set to leaf employees under the given root, simply add a filter with a NOT EXISTS predicate to select only employees that are not managers of other employees:

```
SELECT empid
FROM dbo.fn_subordinates1(3) AS P
WHERE NOT EXISTS
  (SELECT * FROM dbo.Employees AS C
    WHERE c.mgrid = P.empid);
```

This query returns employee IDs 11, 12, 13, and 14.

So far, you've seen the UDF implementation of a subtree under a given root. Listing 9-5 has the CTE solution (SQL Server 2005 only).

Listing 9-5: Subtree of a given root, CTE solution

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

WITH SubsCTE
AS
(
  -- Anchor member returns root node
  SELECT empid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  -- Recursive member returns next level of children
  SELECT C.empid, C.empname, P.lvl + 1
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
    ON C.mgrid = P.empid
)
SELECT * FROM SubsCTE;
```

Running the code in Listing 9-5 gives the same results shown in Table 9-7.

The solution applies very similar logic to the UDF implementation. It's simpler in the sense that you don't need to explicitly define the returned table or to filter the previous level's managers.

The first query in the CTE's body returns the row from Employees for the given root employee. It also returns zero as the level of the root employee. In a recursive CTE, a query that doesn't have any recursive references is known as an *anchor member*.

The second query in the CTE's body (following the UNION ALL set operation) has a recursive reference to the CTE's name. This makes it a *recursive member*, and it is treated in a special manner. The recursive reference to the CTE's name (SubsCTE) represents the result set returned previously. The recursive member query joins the previous result set, which represents the managers in the previous level, with the Employees table to return the next level of employees. The recursive query also calculates the level value as the employee's manager level plus one. The first time that the recursive member is invoked, SubsCTE stands for the result set returned by the anchor member (root employee). There's no explicit termination check for the recursive member; rather, it is invoked repeatedly until it returns an empty set. Thus, the first time it is invoked, it returns direct subordinates of the subtree's root employee. The second time it is invoked, SubsCTE represents the result set of the first invocation of the recursive member (first level of subordinates), so it returns the second level of subordinates. The recursive member is invoked repeatedly until there are no more subordinates, in which case it will return an empty set and recursion will stop.

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

As I mentioned earlier, using iterative logic to return a subgraph of a digraph where multiple paths might exist to a node is similar to returning a subtree. Run the code in Listing 9-6 to create the *fn_partsexplosion* function. The function accepts a part ID representing an assembly in a BOM, and it returns the parts explosion (direct and indirect subitems) of the assembly.

Listing 9-6: Creation script for the *fn_partsexplosion* function

```
---------------------------------------------------------------------
-- Function: fn_partsexplosion, Parts Explosion
--
-- Input   : @root INT: assembly id
--
-- Output  : @PartsExplosion Table:
--             id and level of contained parts of input part
--             in all levels
--
-- Process : * Insert into @PartsExplosion row of input root part
--           * In a loop, while previous insert loaded more than 0 rows
--             insert into @PartsExplosion next level of parts
---------------------------------------------------------------------

USE tempdb;
GO
IF OBJECT_ID('dbo.fn_partsexplosion') IS NOT NULL
  DROP FUNCTION dbo.fn_partsexplosion;
GO
CREATE FUNCTION dbo.fn_partsexplosion(@root AS INT)
  RETURNS @PartsExplosion Table
(
  partid INT           NOT NULL,
  qty    DECIMAL(8, 2) NOT NULL,
  unit   VARCHAR(3)    NOT NULL,
  lvl    INT           NOT NULL,
  n      INT           NOT NULL IDENTITY, -- surrogate key
  UNIQUE CLUSTERED(lvl, n) -- Index will be used to filter lvl
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;                -- Initialize level counter with 0

  -- Insert root node to @PartsExplosion
  INSERT INTO @PartsExplosion(partid, qty, unit, lvl)
```

```
     SELECT partid, qty, unit, @lvl
     FROM dbo.BOM
     WHERE partid = @root;

  WHILE @@rowcount < 0         -- while previous level had rows
  BEGIN
    SET @lvl = @lvl + 1;              -- Increment level counter

    -- Insert next level of subordinates to @PartsExplosion
    INSERT INTO @PartsExplosion(partid, qty, unit, lvl)
      SELECT C.partid, P.qty * C.qty, C.unit, @lvl
      FROM @PartsExplosion AS P -- P = Parent
        JOIN dbo.BOM AS C       -- C = Child
          ON P.lvl = @lvl - 1   -- Filter parents from previous level
          AND C.assemblyid = P.partid;
  END

  RETURN;
END
GO
```

The implementation of the *fn_partsexplosion* function is similar to the implementation of the function *fn_subordinates1*. The row for the root part is loaded to the @PartsExplosion table variable (the function's output parameter). And then in a loop, while the previous insert loaded more than zero rows, the next level parts are loaded into @PartsExplosion. There is a small addition here that is specific to a BOM—calculating the quantity. The root part's quantity is simply the one stored in the part's row. The contained (child) part's quantity is the quantity of its containing (parent) item multiplied by its own quantity.

Run the following code to test the function, returning the part explosion of *partid* 2 (White Tea):

```
SELECT P.partid, P.partname, PE.qty, PE.unit, PE.lvl
FROM dbo.fn_partsexplosion(2) AS PE
  JOIN dbo.Parts AS P
    ON P.partid = PE.partid;
```

You can check the correctness of the output shown in Table 9-8 by examining Figure 9-3.

Table 9-8: Explosion of Part 2

partid	partname	qty	unit	lvl
2	White Tea	1.00	EA	0
6	Cup Cover	1.00	EA	1
7	Regular Cup	1.00	EA	1
10	Tea Shot	1.00	EA	1
11	Milk	25.00	mL	1
14	Water	205.00	mL	1
13	Tea Leaves	5.00	g	2
14	Water	20.00	mL	2

Listing 9-7 has the CTE solution for the parts explosion, which, again, is similar to the subtree solution with the addition of the quantity calculation.

Listing 9-7: CTE solution for the parts explosion

```
DECLARE @root AS INT;
SET @root = 2;

WITH PartsExplosionCTE
AS
(
  -- Anchor member returns root part
  SELECT partid, qty, unit, 0 AS lvl
  FROM dbo.BOM
```

```
  WHERE partid = @root

  UNION ALL

  -- Recursive member returns next level of parts
  SELECT C.partid, CAST(P.qty * C.qty AS DECIMAL(8, 2)),
    C.unit, P.lvl + 1
  FROM PartsExplosionCTE AS P
    JOIN dbo.BOM AS C
    ON C.assemblyid = P.partid
)
SELECT P.partid, P.partname, PE.qty, PE.unit, PE.lvl
FROM PartsExplosionCTE AS PE
  JOIN dbo.Parts AS P
  ON P.partid = PE.partid;
```

A parts explosion might contain more than one occurrence of the same part because different parts in the assembly might contain the same subpart. For example, you can notice in Table 9-8 that water appears twice because white tea contains 205 milliliters of water directly, and it also contains a tea shot, which in turn contains 20 milliliters of water. You might want to aggregate the result set by part and unit as follows, generating the output shown in Table 9-9:

```
SELECT P.partid, P.partname, PES.qty, PES.unit
FROM (SELECT partid, unit, SUM(qty) AS qty
      FROM dbo.fn_partsexplosion(2) AS PE
      GROUP BY partid, unit) AS PES
  JOIN dbo.Parts AS P
    ON P.partid = PES.partid;
```

Table 9-9: Explosion of Part 2, with Aggregated Parts

partid	partname	qty	unit
2	White Tea	1.00	EA
6	Cup Cover	1.00	EA
7	Regular Cup	1.00	EA
10	Tea Shot	1.00	EA
13	Tea Leaves	5.00	g
11	Milk	25.00	mL
14	Water	225.00	mL

I won't get into issues with grouping of parts that might contain different units of measurements here. Obviously, you'll need to deal with those by applying conversion factors.

As another example, the following code explodes part 5 (Double Espresso), returning the output shown in Table 9-10:

```
SELECT P.partid, P.partname, PES.qty, PES.unit
FROM (SELECT partid, unit, SUM(qty) AS qty
      FROM dbo.fn_partsexplosion(5) AS PE
      GROUP BY partid, unit) AS PES
  JOIN dbo.Parts AS P
     ON P.partid = PES.partid;
```

Table 9-10: Explosion of Part 5, with Aggregated Parts

partid	partname	qty	unit
5	Double Espresso	1.00	EA
9	Espresso Cup	1.00	EA
12	Coffee Shot	2.00	EA
16	Ground Coffee	30.00	g
17	Coffee Beans	450.00	g

14	Water	40.00	mL

Going back to returning a subtree of a given employee, you might need in some cases to limit the number of returned levels. To achieve this, there's a minor addition you need to make to the original algorithm:

Input: *@root*, *@maxlevels* (besides root)

Algorithm:

- set @*lvl* = 0; insert into table @Subs row for *@root*

- while there were rows in the previous level, and *@lvl* < *@maxlevels:*

- set @*lvl* = @*lvl* + *1*; insert into table @Subs rows for the next level (*mgrid* in (*empid* values in previous level))

- return @Subs

Run the code in Listing 9-8 to create the *fn_subordinates2* function, which is a revision of *fn_subordinates2* that also supports a level limit.

Listing 9-8: Creation script for the *fn_subordinates2* function

```
---------------------------------------------------------------------
-- Function: fn_subordinates2,
--           Descendants with optional level limit
--
-- Input   : @root INT: Manager id
--           @maxlevels INT: Max number of levels to return
--
-- Output  : @Subs TABLE: id and level of subordinates of
--           input manager in all levels <= @maxlevels
--
-- Process : * Insert into @Subs row of input manager
--           * In a loop, while previous insert loaded more than 0 rows
--             and previous level is smaller than @maxlevels
--             insert into @Subs next level of subordinates
---------------------------------------------------------------------
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_subordinates2') IS NOT NULL
  DROP FUNCTION dbo.fn_subordinates2;
GO
CREATE FUNCTION dbo.fn_subordinates2
  (@root AS INT, @maxlevels AS INT = NULL) RETURNS @Subs TABLE
(
  empid INT NOT NULL PRIMARY KEY NONCLUSTERED,
  lvl   INT NOT NULL,
  UNIQUE CLUSTERED(lvl, empid) -- Index will be used to filter level
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;                -- Initialize level counter with 0
  -- If input @maxlevels is NULL, set it to maximum integer

  -- to virtually have no limit on levels
  SET @maxlevels = COALESCE(@maxlevels, 2147483647);

  -- Insert root node to @Subs
  INSERT INTO @Subs(empid, lvl)
    SELECT empid, @lvl FROM dbo.Employees WHERE empid = @root;

  WHILE @@rowcount > 0     -- while previous level had rows
    AND @lvl < @maxlevels  -- and previous level < @maxlevels
  BEGIN
    SET @lvl = @lvl + 1;   -- Increment level counter
```

```
  -- Insert next level of subordinates to @Subs
  INSERT INTO @Subs(empid, lvl)
    SELECT C.empid, @lvl
    FROM @Subs AS P             -- P = Parent
      JOIN dbo.Employees AS C   -- C = Child
        ON P.lvl = @lvl - 1     -- Filter parents from previous level
        AND C.mgrid = P.empid;
  END

  RETURN;
END
GO
```

In addition to the original input, *fn_subordinates2* also accepts the *@maxlevels* input that indicates the maximum number of requested levels under *@root* to return. For no limit on levels, a NULL should be specified in *@maxlevels*. Notice that if *@maxlevels* is NULL, the function substitutes the NULL with the maximum possible integer value to practically have no limit.

The loop's condition, besides checking that the previous insert affected more than zero rows, also checks that the *@lvl* variable is smaller than *@maxlevels*. Except for these minor revisions, the function's implementation is the same as *fn_subordinates1*.

To test the function, run the following code that requests the subordinates of employee 3 in all levels (*@maxlevels* is NULL) and generates the output shown in Table 9-11:

```
SELECT empid, lvl
FROM dbo.fn_subordinates2(3, NULL) AS S;
```

Table 9-11: Subtree of Employee 3, with No Level Limit

empid	lvl
3	0
7	1
9	2
11	2
12	3
13	3
14	3

To get only two levels of subordinates under employee 3, run the following code, which generates the output shown in Table 9-12:

```
SELECT empid, lvl
FROM dbo.fn_subordinates2(3, 2) AS S;
```

Table 9-12: Subtree of Employee 3, up to 2 Levels

empid	lvl
3	0
7	1
9	2
11	2

To get only the second level employees under employee 3, add a filter on the level, which will generate the output shown in Table 9-13:

```
SELECT empid
FROM dbo.fn_subordinates2(3, 2) AS S
WHERE lvl = 2;
```

Table 9-13: Subtree of Employee 3, Only Level 2

empid
9
11

Caution To limit levels using a CTE, you might be tempted to use the hint called MAXRECURSION, which raises an error and aborts when the number of invocations of the recursive member exceeds the input. However, MAXRECURSION was designed as a safety measure to avoid infinite recursion in cases of problems in the data or bugs in the code. When not specified, MAXRECURSION defaults to 100. You can specify MAXRECURSION 0 to have no limit, but be aware of the implications.

To test this approach, run the code in Listing 9-9, which generates the output shown in Table 9-14. It's the same subtree CTE shown earlier, with the addition of the MAXRECURSION hint, limiting recursive invocations to 2.

Table 9-14: Subtree of Employee 3, Levels Limited with MAXRECURSION

empid	empname	lvl
3	Ina	0
7	Aaron	1
11	Gabriel	2
9	Rita	2

Listing 9-9: Subtree with level limit, CTE solution with MAXRECURSION

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

WITH SubsCTE
AS
(

   SELECT empid, empname, 0 AS lvl
   FROM dbo.Employees
   WHERE empid = @root

   UNION ALL

   SELECT C.empid, C.empname, P.lvl + 1
   FROM SubsCTE AS P
    JOIN dbo.Employees AS C
    ON C.mgrid = P.empid
)
SELECT * FROM SubsCTE
OPTION (MAXRECURSION 2);
```

```
Server: Msg 530, Level 16, State 1, Line 4
```

Caution The statement terminated. The maximum recursion 2 has been exhausted before statement completion.

The code breaks as soon as the recursive member is invoked the third time. It's not recommended to use the MAXRECURSION hint to logically limit the number of levels for two reasons. First, an error is generated even though there's no logical error here. Second, SQL Server does not guarantee to return any result set if an error is generated. In this particular case, a result set was returned, but there's no guarantee that will happen in other cases.

To logically limit the number of levels, simply add a filter on the parent's level column in the recursive member's join condition, as shown in Listing 9-10.

Listing 9-10: Subtree with level limit, CTE solution, with level column

```
DECLARE @root AS INT, @maxlevels AS INT;
SET @root = 3;
SET @maxlevels = 2;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
      AND P.lvl < @maxlevels -- limit parent's level
)
SELECT * FROM SubsCTE;
```

Ancestors

Requests for ancestors of a given node are also common—for example, returning the chain of management for a given employee. Not surprisingly, the algorithms for returning ancestors using iterative logic are similar to those for returning subordinates. Simply, instead of traversing the graph starting with a given node and proceeding "downwards" to child nodes, you start with a given node and proceed "upwards" to parent nodes.

Run the code in Listing 9-11 to create the *fn_managers* function. The function accepts an input employee ID (*@empid*) and, optionally, a level limit (*@maxlevels*), and it returns managers up to the requested number of levels away from the input employee (if a limit was specified).

Listing 9-11: Creation script for the *fn_managers* function

```
---------------------------------------------------------------------
-- Function: fn_managers, Ancestors with optional level limit
--
-- Input   : @empid INT : Employee id
--           @maxlevels : Max number of levels to return
--
-- Output  : @Mgrs Table: id and level of managers of
--                        input employee in all levels <= @maxlevels
--
-- Process : * In a loop, while current manager is not null
--             and previous level is smaller than @maxlevels
--             insert into @Mgrs current manager,
--             and get next level manager
---------------------------------------------------------------------
USE tempdb;
GO
```

```
IF OBJECT_ID('dbo.fn_managers') IS NOT NULL
  DROP FUNCTION dbo.fn_managers;
GO
CREATE FUNCTION dbo.fn_managers
  (@empid AS INT, @maxlevels AS INT = NULL) RETURNS @Mgrs TABLE
(
  empid INT NOT NULL PRIMARY KEY,
  lvl   INT NOT NULL
)
AS

BEGIN
  IF NOT EXISTS(SELECT * FROM dbo.Employees WHERE empid = @empid)
    RETURN;

  DECLARE @lvl AS INT;
  SET @lvl = 0;                    -- Initialize level counter with 0
  -- If input @maxlevels is NULL, set it to maximum integer
  -- to virtually have no limit on levels
  SET @maxlevels = COALESCE(@maxlevels, 2147483647);

  WHILE @empid IS NOT NULL         -- while current employee has a manager
    AND @lvl <= @maxlevels         -- and previous level < @maxlevels
  BEGIN
    -- Insert current manager to @Mgrs
    INSERT INTO @Mgrs(empid, lvl) VALUES(@empid, @lvl);
    SET @lvl = @lvl + 1;           -- Increment level counter
    -- Get next level manager
    SET @empid = (SELECT mgrid FROM dbo.Employees
                  WHERE empid = @empid);
  END

  RETURN;
END
GO
```

The function first checks whether the input node ID exists, and then breaks if it doesn't. It then initializes the *@lvl* counter to zero, and it assigns the maximum possible integer to the *@maxlevels* variable if a NULL was specified in it to practically have no level limit.

The function then enters a loop that iterates as long as *@empid* is not null (because null represents the root's manager ID) and the current level is smaller than or equal to the requested number of levels. The loop's body inserts the current employee ID along with the level counter into the @Mgrs output table variable, increments the level counter, and assigns the current employee's manager's ID to the *@empid* variable.

I should point out a couple of differences between this function and the subordinates function. This function uses a scalar subquery to get the manager ID in the next level, unlike the subordinates function, which used a join to get the next level of subordinates. The reason for the difference is that there can be only one manager for a given employee, while there can be multiple subordinates for a given manager. Also, this function uses the expression *@lvl* <= *@maxlevels* to limit the number of levels, while the subordinates function used the expression *@lvl* < *@maxlevels*. The reason for the discrepancy is that this function doesn't have a separate INSERT statement to get the root employee and a separate one to get the next level of employees; rather, it has only one INSERT statement in the loop. Consequently the *@lvl* counter here is incremented after the INSERT, while in the subordinates function it was incremented before the INSERT.

To test the function, run the following code, which returns managers in all levels of employee 8 and generates the output shown in Table 9-15:

```
SELECT empid, lvl
FROM dbo.fn_managers(8, NULL) AS M;
```

Table 9-15: Management Chain of Employee 8, No Level

Limit

empid	lvl
1	3
2	2
5	1
8	0

The CTE solution to returning ancestors is almost identical to the CTE solution returning a subtree. The minor difference is that here the recursive member treats the CTE as the child part of the join and the Employees table as the parent part, while in the subtree solution the roles were opposite. Run the code in Listing 9-12 to get the management chain of employee 8 using a CTE and generate the output shown in Table 9-16.

Table 9-16: CTE Output for Management Chain of Employee 8

empid	mgrid	empname	lvl
8	5	Lilach	0
5	2	Jiru	1
2	1	Eitan	2
1	NULL	David	3

Listing 9-12: Management chain of employee 8, CTE solution

```
DECLARE @empid AS INT;
SET @empid = 8;

WITH MgrsCTE
AS
(
  SELECT empid, mgrid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @empid

  UNION ALL

  SELECT P.empid, P.mgrid, P.empname, C.lvl + 1
  FROM MgrsCTE AS C
   JOIN dbo.Employees AS P
     ON C.mgrid = P.empid
)
SELECT * FROM MgrsCTE;
```

To get only two levels of managers of employee 8 using the *fn_managers* function, run the following code, which generates the output shown in Table 9-17:

```
SELECT empid, lvl
FROM dbo.fn_managers(8, 2) AS M;
```

Table 9-17: Management Chain of Employee 8, 2 Level Limit, CTE Solution

empid	lvl
2	2

5	1
8	0

And to return only the second-level manager, simply add a filter in the outer query, returning employee ID 2:

```
SELECT empid
FROM dbo.fn_managers(8, 2) AS M
WHERE lvl = 2;
```

To return two levels of managers of employee 8 with a CTE, simply add a filter on the child's level in the join condition of the recursive member as shown in Listing 9-13.

Listing 9-13: Ancestors with level limit, CTE solution

```
DECLARE @empid AS INT, @maxlevels AS INT;
SET @empid = 8;
SET @maxlevels = 2;

WITH MgrsCTE
AS
(
  SELECT empid, mgrid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @empid

  UNION ALL

  SELECT P.empid, P.mgrid, P.empname, C.lvl + 1
  FROM MgrsCTE AS C
   JOIN dbo.Employees AS P
     ON C.mgrid = P.empid
     AND C.lvl < @maxlevels -- limit child's level
)
SELECT * FROM MgrsCTE;
```

Subgraph/Subtree with Path Enumeration

In the subgraph/subtree solutions, you might also want to generate for each node an enumerated path consisting of all node IDs in the path leading to the node with some separator (for example, '.'). For example, the enumerated path for employee 8 in the Organization Chart scenario is '.1.2.5.8.' because employee 5 is the manager of employee 8, employee 2 is the manager of 5, employee 1 is the manager of 2, and employee 1 is the root employee.

The enumerated path has many uses—for example, to sort the nodes from the hierarchy in the output, to detect cycles, and other uses that I'll describe later in the "Materialized Path" section.

Fortunately, you can make minor additions to the solutions I provided for returning a subgraph/subtree to calculate the enumerated path without any additional I/O.

The algorithm starts with the subtree's root node, and in a loop or recursive call returns the next level. For the root node, the path is simply: *'.' + node id + '.'*. For successive level nodes, the path is: *parent's path + node id + '.'*.

Run the code in Listing 9-14 to create the *fn_subordinates3* function, which is the same as *fn_subordinates2* except for the addition of the enumerated path calculation.

Listing 9-14: Creation script for the *fn_subordinates3* function

```
---------------------------------------------------------------------
-- Function: fn_subordinates3,
--           Descendants with optional level limit,
--           and path enumeration
--
-- Input   : @root      INT: Manager id
--           @maxlevels INT: Max number of levels to return
--
-- Output  : @Subs TABLE: id, level and materialized ancestors path
```

```
--                          of subordinates of input manager
--                          in all levels <= @maxlevels
--
-- Process : * Insert into @Subs row of input manager
--           * In a loop, while previous insert loaded more than 0 rows
--             and previous level is smaller than @maxlevels:
--             - insert into @Subs next level of subordinates
--             - calculate a materialized ancestors path for each
--               by concatenating current node id to parent's path
---------------------------------------------------------------------
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_subordinates3') IS NOT NULL
  DROP FUNCTION dbo.fn_subordinates3;
GO
CREATE FUNCTION dbo.fn_subordinates3
  (@root AS INT, @maxlevels AS INT = NULL) RETURNS @Subs TABLE
(
  empid  INT          NOT NULL PRIMARY KEY NONCLUSTERED,
  lvl    INT          NOT NULL,
  path   VARCHAR(900) NOT NULL
  UNIQUE CLUSTERED(lvl, empid) -- Index will be used to filter level
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;                -- Initialize level counter with 0
  -- If input @maxlevels is NULL, set it to maximum integer
  -- to virtually have no limit on levels
  SET @maxlevels = COALESCE(@maxlevels, 2147483647);

  -- Insert root node to @Subs
  INSERT INTO @Subs(empid, lvl, path)

  SELECT empid, @lvl, '.' + CAST(empid AS VARCHAR(10)) + '.'
  FROM dbo.Employees WHERE empid = @root;

WHILE @@rowcount > 0            -- while previous level had rows
  AND @lvl < @maxlevels         -- and previous level < @maxlevels
BEGIN
  SET @lvl = @lvl + 1;          -- Increment level counter

  -- Insert next level of subordinates to @Subs
  INSERT INTO @Subs(empid, lvl, path)
     SELECT C.empid, @lvl,
       P.path + CAST(C.empid AS VARCHAR(10)) + '.'
     FROM @Subs AS P -- P = Parent
       JOIN dbo.Employees AS C -- C = Child
         ON P.lvl = @lvl - 1 -- Filter parents from previous level
         AND C.mgrid = P.empid;
  END

  RETURN;
END
GO
```

To test the function, run the following code, which returns all subordinates of employee 1 and their paths, as shown in Table 9-18:

```
SELECT empid, lvl, path
FROM dbo.fn_subordinates3(1, NULL) AS S;
```

Table 9-18: Subtree with Enumerated Path

empid	lvl	path
1	0	.1

2	1	.1.2
3	1	.1.3
4	2	.1.2.4
5	2	.1.2.5
6	2	.1.2.6
7	2	.1.3.7
8	3	.1.2.5.8
9	3	.1.3.7.9
10	3	.1.2.5.10
11	3	.1.3.7.11
12	4	.1.3.7.9.12
13	4	.1.3.7.9.13
14	4	.1.3.7.9.14

With both the *lvl* and *path* values, you can easily return output that graphically shows the hierarchical relationships of the employees in the subtree:

```
SELECT E.empid, REPLICATE(' | ', lvl) + empname AS empname
FROM dbo.fn_subordinates3(1, NULL) AS S
  JOIN dbo.Employees AS E
    ON E.empid = S.empid
ORDER BY path;
```

The query joins the subtree returned from the *fn_subordinates3* function with the Employees table based on employee ID match. From the function, you get the *lvl* and *path* values, and from the table you get other employee attributes of interest, such as the employee name. You generate indentation before the employee name by replicating a string (in this case, ' | ') *lvl* times and concatenating the employee name to it. Sorting the employees by the *path* column produces a correct hierarchical sort, which requires that a child node will appear later than its parent node—or in other words, that a child node will have a higher sort value than its parent node. By definition, a child's path is greater than a parent's path because it's prefixed with the parent's path. The output of this query is shown in Table 9-19.

Table 9-19: Subtree, Sorted by Path and Indented by Level

empid	empname
1	David
2	| Eitan
4	| | Seraph
5	| | Jiru
10	| | | Sean
8	| | | Lilach
6	| | Steve
3	| Ina
7	| | Aaron
11	| | | Gabriel
9	| | | Rita
12	| | | | Emilia
13	| | | | Michael
14	| | | | Didi

Similarly, you can add path calculation to the subtree CTE as shown in Listing 9-15.

Listing 9-15: Subtree with path enumeration, CTE solution

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS

(
  SELECT empid, empname, 0 AS lvl,
    -- Path of root = '.' + empid + '.'
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
  -- Path of child = parent's path + child empid + '.'
  CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
       AS VARCHAR(MAX)) AS path
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, REPLICATE(' | ', lvl) + empname AS empname
FROM SubsCTE
ORDER BY path;
```

Note Corresponding columns between an anchor member and a recursive member of a CTE must match in both data type and size. That's the reason I converted the path strings in both to the same datatype and size—VARCHAR (MAX).

Sorting

Sorting is a presentation request and usually is used by the client rather than the server. This means that you might want the sorting of hierarchies to take place on the client. In this section, however, I'll present server-side sorting techniques with T-SQL that you can use when you prefer to handle sorting on the server.

A *topological sort* of a DAG is defined as one that provides a child with a higher sort value than its parent. Occasionally, I will refer to a topological sort informally as "correct hierarchical sort." More than one way of ordering the items in a DAG may qualify as correct. You might or might not care about the order among siblings. If the order among siblings doesn't matter to you, you can achieve sorting by constructing an enumerated path for each node, as described in the previous section, and sort the nodes by that path.

Remember that the enumerated path is a character string made of the IDs of the ancestors leading to the node, using some separator. This means that siblings are sorted by their node IDs. Because the path is character based, you get character-based sorting of IDs, which might be different than the integer sorting. For example, employee ID 11 will sort lower than its sibling with ID 9 (*'.1.3.7.11.'* < *'.1.3.7.9.'*), even though 9 < 11. You can guarantee that sorting by the enumerated path will produce a correct hierarchical sort, but it will not guarantee the order of siblings. If you need such a guarantee, you need a different solution.

For optimal sorting flexibility, you might want to guarantee the following:

1. A correct topological sort—that is, a sort in which a child will have a higher sort value than its parent's.
2. Siblings are sorted in a requested order (for example, by *empname* or by *salary*).
3. Integer sort values are generated, as opposed to lengthy strings.

In the enumerated path solution, requirement 1 is met. Requirement 2 is not met because the path is made of node IDs and is character based; comparison and sorting among characters is based on collation properties, yielding different comparison and sorting behavior than with integers. Requirement 3 is not met because the solution orders the results by the path, which is lengthy compared to an integer value. To meet all three requirements, we can still make use of a path for each node, but with several differences:

- Instead of node IDs, the path will be constructed from values that represent a position (row number) among nodes based on a requested order (for example, *empname* or *salary*).
- Instead of using a character string with varying lengths for each level in the path, use a binary string with a fixed length for each level.
- Once the binary paths are constructed, calculate integer values representing path order (row numbers) and ultimately use those to sort the hierarchy.

The core algorithm to traverse the subtree is maintained. It's just that the paths are constructed differently, and you need to figure out how to calculate row numbers. In SQL Server 2000, to calculate row numbers based on a requested order you can insert the rows into a table with an identity column using INSERT…SELECT…ORDER BY. (See Knowledge Base article 273586 at http://www.support.microsoft.com/default.aspx?scid=kb; en-us;273586.)

In SQL Server 2005, you can use the ROW_NUMBER function, which is much simpler and faster than the SQL Server 2000 alternative.

Run the code in Listing 9-16 to create the SQL Server 2000–compatible stored procedure usp_sortsubs, which implements this logic.

Listing 9-16: Creation script for the usp_sortsubs procedure

```
---------------------------------------------------------------------
-- Stored Procedure: usp_sortsubs,
--   Descendants with optional level limit and sort values
--
-- Input : @root      INT: Manager id
--         @maxlevels INT: Max number of levels to return
--         @orderby   sysname: determines sort order
--
-- Output : Rowset: id, level and sort values
--                  of subordinates of input manager
--                  in all levels <= @maxlevels
--

-- Process : * Use a loop to load the desired subtree into #SubsPath
--           * For each node, construct a binary sort path
--           * The row number represents the node's position among
--             its siblings based on the input ORDER BY list
--           * Insert the contents of #SubPath into #SubsSort sorted
--             by the binary sortpath
--           * IDENTITY values representing the global sort value
--             in the subtree will be generated in the target
--             #SubsSort table
--           * Return all rows from #SubsSort sorted by the
--             sort value
---------------------------------------------------------------------
USE tempdb;
GO
IF OBJECT_ID('dbo.usp_sortsubs') IS NOT NULL
  DROP PROC dbo.usp_sortsubs;
GO
CREATE PROC dbo.usp_sortsubs
  @root      AS INT     = NULL,
  @maxlevels AS INT     = NULL,
  @orderby   AS sysname = N'empid'
AS

SET NOCOUNT ON;

-- #SubsPath is a temp table that will hold binary sort paths
CREATE TABLE #SubsPath
(
  rownum   INT NOT NULL IDENTITY,
```

```
    nodeid   INT NOT NULL,
    lvl      INT NOT NULL,
    sortpath VARBINARY(900) NULL
  );
  CREATE UNIQUE CLUSTERED INDEX idx_uc_lvl_empid ON #SubsPath(lvl, nodeid);

  -- #SubsPath is a temp table that will hold the final
  -- integer sort values
  CREATE TABLE #SubsSort
  (
    nodeid  INT NOT NULL,
    lvl     INT NOT NULL,
    sortval INT NOT NULL IDENTITY
  );
  CREATE UNIQUE CLUSTERED INDEX idx_uc_sortval ON #SubsSort(sortval);

  -- If @root is not specified, set it to root of the tree
  IF @root IS NULL
    SET @root = (SELECT empid FROM dbo.Employees WHERE mgrid IS NULL);
  -- If @maxlevels is not specified, set it maximum integer
  IF @maxlevels IS NULL
    SET @maxlevels = 2147483647;

  DECLARE @lvl AS INT, @sql AS NVARCHAR(4000);
  SET @lvl = 0;

  -- Load row for input root to #SubsPath
  -- The root's sort path is simply 1 converted to binary
  INSERT INTO #SubsPath(nodeid, lvl, sortpath)
    SELECT empid, @lvl, CAST(1 AS BINARY(4))
    FROM dbo.Employees
    WHERE empid = @root;

  -- Form a loop to load the next level of subordinates
  -- to #SubsPath in each iteration
  WHILE @@rowcount > 0 AND @lvl < @maxlevels
  BEGIN
    SET @lvl = @lvl + 1;

    -- Insert next level of subordinates
    -- Initially, just copy parent's path to child
    -- Note that IDENTITY values will be generated in #SubsPath
    -- based on input order by list
    --
    -- Then update the path of the employees in the current level
    -- to their parent's path + their rownum converted to binary
    INSERT INTO #SubsPath(nodeid, lvl, sortpath)
      SELECT C.empid, @lvl, P.sortpath
      FROM #SubsPath AS P
       JOIN dbo.Employees AS C
        ON P.lvl = @lvl - 1
        AND C.mgrid = P.nodeid
      ORDER BY -- determines order of siblings
        CASE WHEN @orderby = N'empid' THEN empid END,
        CASE WHEN @orderby = N'empname' THEN empname END,
        CASE WHEN @orderby = N'salary' THEN salary END;

    UPDATE #SubsPath
      SET sortpath = sortpath + CAST(rownum AS BINARY(4))
    WHERE lvl = @lvl;
  END

  -- Load the rows from #SubsPath to @SubsSort sorted by the binary
  -- sort path
  -- The target identity values in the sortval column will represent
  -- the global sort value of the nodes within the result subtree
  INSERT INTO #SubsSort(nodeid, lvl)
    SELECT nodeid, lvl FROM #SubsPath ORDER BY sortpath;
```

```
-- Return for each node the id, level and sort value
SELECT nodeid AS empid, lvl, sortval FROM #SubsSort
ORDER BY sortval;
GO
```

The input parameters *@root* and *@maxlevels* are similar to the ones used in the previous subtree routines I discussed. In addition, the stored procedure accepts the *@orderby* parameter, where you specify a column name by which you want siblings sorted. The stored procedure uses a series of CASE expressions to determine which column's values to sort by. The stored procedure returns a result set with the node IDs in the requested subtree, along with a level and an integer sort value for each node.

The stored procedure traverses the subtree in a similar fashion to the previous iterative implementations I discussed—that is, a level at a time.

First, the root employee is loaded into the #SubsPath temporary table. Then, in each iteration of the loop, the next level of employees is inserted into #SubsPath.

The #SubsPath table has an identity column (*rownum*) that will represent the position of an employee among siblings based on the desired sort (the ORDER BY section of the INSERT SELECT statement). The root's path is set to 1 converted to BINARY(4). For each level of employees that is inserted into the #SubsPath table, the parent's path is copied to the child's path, and then an UPDATE statement concatenates to the child's path the *rownum* value converted to BINARY(4).

At the end of the loop, #SubsPath contains the complete binary sort path for each node.

This process will probably be better explained by following an example. Say you're after the subtree of employee 1 (David) with no level limit, sorting siblings by *empname*. Table 9-20 shows the identity values that are generated for the employees in each level.

Table 9-20: Identity Values Generated for Employees in Each Level

Level 0	Level 1	Level 2	Level 3	Level 4
1 - David	2 - Eitan	4 - Aaron	8 - Gabriel	12 - Didi
	3 - Ina	5 - Jiru	9 - Lilach	13 - Emilia
		6 - Seraph	10 - Rita	14 - Michael
		7 - Steve	11 - Sean	

Table 9-21 shows the binary sort paths constructed for each employee, made of the position values of the ancestors leading to the node.

Table 9-21: Binary Sort Paths Constructed for Each Employee

Lvl	Manager	Employee	Sort Path				
0	NULL	David (1)	1				
1	David	Eitan (2)	1	2			
1	David	Ina (3)	1	3			
2	Eitan	Jiru (5)	1	2	5		
2	Eitan	Seraph (6)	1	2	6		
2	Eitan	Steve (7)	1	2	7		
2	Ina	Aaron (4)	1	3	4		
3	Jiru	Lilach (9)	1	2	5	9	
3	Jiru	Sean (11)	1	2	5	11	
3	Aaron	Gabriel (8)	1	3	4	8	
3	Aaron	Rita (10)	1	3	4	10	

4	Rita	Didi (12)	1	3	4	10	12
4	Rita	Emilia (13)	1	3	4	10	13
4	Rita	Michael (14)	1	3	4	10	14

The next step in the stored procedure is to insert the contents of #SubsPath into #SubsSort in *sortpath* order. #SubsSort also has an identity column (*sortval*), which will represent the employees' final sort values. Table 9-22 will help you visualize how the sort values are calculated in #SubsSort based on *sortpath* order.

Table 9-22: Integer Sort Values Calculated Based on *sortpath* Order

sortval	lvl	Manager	Employee	sortpath				
1	0	NULL	David (1)	1				
2	1	David	Eitan (2)	1	2			
3	2	Eitan	Jiru (5)	1	2	5		
4	3	Jiru	Lilach (9)	1	2	5	9	
5	3	Jiru	Sean (11)	1	2	5	11	
6	2	Eitan	Seraph (6)	1	2	6		
7	2	Eitan	Steve (7)	1	2	7		
8	1	David	Ina (3)	1	3			
9	2	Ina	Aaron (4)	1	3	4		
10	3	Aaron	Gabriel (8)	1	3	4	8	
11	3	Aaron	Rita (10)	1	3	4	10	
12	4	Rita	Didi (12)	1	3	4	10	12
13	4	Rita	Emilia (13)	1	3	4	10	13
14	4	Rita	Michael (14)	1	3	4	10	14

Finally, the stored procedure returns for each node the node ID, level, and integer sort value. To test the procedure, run the following code, specifying *empname* as the sort columns. The code generates the output shown in Table 9-23.

```
EXEC dbo.usp_sortsubs @orderby = N'empname';
```

Table 9-23: All Employee IDs with Sort Values Based on *empname*

empid	lvl	sortval
1	0	1
2	1	2
5	2	3
8	3	4
10	3	5
4	2	6
6	2	7
3	1	8
7	2	9
11	3	10
9	3	11
14	4	12
12	4	13

13	4	14

To get three levels of subordinates underneath employee 1 having siblings sorted by *empname*, run the following code, which generates the output shown in Table 9-24:

```
EXEC dbo.usp_sortsubs
  @root = 1,
  @maxlevels = 3,
  @orderby = N'empname';
```

Table 9-24: Subtree with Levels Limit, and Sort Based on *empname*

empid	lvl	sortval
1	0	1
2	1	2
5	2	3
8	3	4
10	3	5
4	2	6
6	2	7
3	1	8
7	2	9
11	3	10
9	3	11

To return attributes other than the employee ID (for example, the employee name), you need to first produce the result set of the stored procedure, and then join it with the Employees table. For example, the code in Listing 9-17 returns all employees, having siblings sorted by *empname*, with indentation, and generates the output shown in Table 9-25:

Table 9-25: All Employees with Sort of Siblings Based on *empname*

empid	empname
1	David
2	| Eitan
5	| | Jiru
8	| | | Lilach
10	| | | Sean
4	| | Seraph
6	| | Steve
3	| Ina
7	| | Aaron
11	| | | Gabriel
9	| | | Rita
14	| | | | Didi
12	| | | | Emilia
13	| | | | Michael

Listing 9-17: Script returning all employees, having siblings sorted by *empname*

```
CREATE TABLE #Subs
(
  empid   INT NULL,
  lvl     INT NULL,
  sortval INT NULL
);
CREATE UNIQUE CLUSTERED INDEX idx_uc_sortval ON #Subs(sortval);

-- By empname
INSERT INTO #Subs(empid, lvl, sortval)
    EXEC dbo.usp_sortsubs
      @orderby = N'empname';

SELECT E.empid, REPLICATE(' | ', lvl) + E.empname AS empname
FROM #Subs AS S
  JOIN dbo.Employees AS E
    ON S.empid = E.empid
  ORDER BY sortval;
```

Similarly, the code in Listing 9-18 returns all employees, having siblings sorted by *salary*, with indentation, and generates the output shown in Table 9-26:

Listing 9-18: Script returning all employees, with siblings sorted by *salary*

```
TRUNCATE TABLE #Subs;

INSERT INTO #Subs(empid, lvl, sortval)
   EXEC dbo.usp_sortsubs
     @orderby = N'salary';

SELECT E.empid, salary, REPLICATE(' | ', lvl) + E.empname AS empname
FROM #Subs AS S
  JOIN dbo.Employees AS E
    ON S.empid = E.empid
ORDER BY sortval;
```

Table 9-26: All Employees with Siblings Sorted by *salary*

empid	salary	empname
1	10000.00	David
2	7000.00	|Eitan
6	4500.00	| | Steve
4	5000.00	| | Seraph
5	5500.00	| | Jiru
10	3000.00	| | | Sean
8	3500.00	| | | Lilach
3	7500.00	| Ina
7	5000.00	| | Aaron
9	3000.00	| | | Rita
14	1500.00	| | | | Didi
12	2000.00	| | | | Emilia
13	2000.00	| | | | Michael
11	3000.00	| | | Gabriel

Make sure you drop the temporary table #Subs once you're finished:

```
DROP TABLE #Subs
```

The implementation of a similar algorithm in SQL Server 2005 is dramatically simpler and faster, mainly because it uses CTEs and the ROW_NUMBER function.

Run the code in Listing 9-19 to return the subtree of employee 1, with siblings sorted by *empname* with indentation, and generate the output shown in Table 9-27.

Table 9-27: Employees with Sort Based on *empname*, CTE Output

empid	sortval	empname
1	1	David
2	2	| Eitan
5	3	| | Jiru
8	4	| | | Lilach
10	5	| | | Sean
4	6	| | Seraph
6	7	| | Steve
3	8	| Ina
7	9	| | Aaron
11	10	| | | Gabriel
9	11	| | | Rita
14	12	| | | | Didi
12	13	| | | | Emilia
13	14	| | | | Michael

Listing 9-19: Returning all employees in the hierarchy with siblings sorted by *empname*, CTE solution

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    -- Path of root is 1 (binary)
    CAST(1 AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    -- Path of child = parent's path + child row number (binary)
    P.sortpath + CAST(
      ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        ORDER BY C.empname) -- sort col(s)
      AS BINARY(4))
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval,
  REPLICATE(' | ', lvl) + empname AS empname
FROM SubsCTE
ORDER BY sortval;
```

The anchor member query returns the root, with 1 as the binary path. The recursive member query calculates the row number of an employee among siblings based on *empname* ordering and concatenates that row number converted to binary(4) to the parent's path.

The outer query simply calculates row numbers to generate the sort values based on the binary path order, and it sorts the subtree by those sort values, adding indentation based on the calculated level.

If you want siblings sorted in a different way, you need to change only the ORDER BY list of the ROW_NUMBER function in the recursive member query. Listing 9-20 has the revision that sorts siblings by *salary*, generating the output shown in Table 9-28.

Table 9-28: Employees with Sort Based on *salary*, CTE Output

empid	salary	sortval	empname
1	10000.00	1	David
2	7000.00	2	| Eitan
6	4500.00	3	| | Steve
4	5000.00	4	| | Seraph
5	5500.00	5	| | Jiru
10	3000.00	6	| | | Sean
8	3500.00	7	| | | Lilach
3	7500.00	8	|Ina
7	5000.00	9	| | Aaron
9	3000.00	10	| | | Rita
14	1500.00	11	| | | | Didi
12	2000.00	12	| | | | Emilia
13	2000.00	13	| | | | Michael
11	3000.00	14	| | | Gabriel

Listing 9-20: Returning all employees in the hierarchy with siblings sorted by *salary*, CTE solution

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, salary, 0 AS lvl,
    -- Path of root = 1 (binary)
    CAST(1 AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, C.salary, P.lvl + 1,
    -- Path of child = parent's path + child row number (binary)
    P.sortpath + CAST(
      ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        ORDER BY C.salary) -- sort col(s)
      AS BINARY(4))
    FROM SubsCTE AS P
      JOIN dbo.Employees AS C
        ON C.mgrid = P.empid
)
SELECT empid, salary, ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval,
  REPLICATE(' | ', lvl) + empname AS empname
```

```
FROM SubsCTE
ORDER BY sortval;
```

Note If you need to sort siblings by a single integer sort column (for example, by *empid*), you can construct the binary sort path from the sort column values themselves instead of row numbers based on that column.

Cycles

Cycles in graphs are paths that begin and end at the same node. In some scenarios, cycles are natural (for example, road systems). If you have a cycle in what's supposed to be an acyclic graph, it might indicate that there's a problem in your data. Either way, you need a way to identify them. If a cycle indicates a problem in the data, you need to identify the problem and fix it. If cycles are natural, while traversing the graph you don't want to endlessly keep returning to the same point.

Cycle detection with T-SQL can be a very complex and expensive task. However, I'll show you how to detect cycles with a fairly simple technique with reasonable performance, relying on path enumeration, which I discussed earlier. For demonstration purposes, I'll use this technique to detect cycles in the tree represented by the Employees table, but you can apply this technique to forests as well and also to more generic graphs, as I will demonstrate later.

Suppose that Didi (*empid* 14) is unhappy with her location in the company's management hierarchy. Didi also happens to be the database administrator and has full access to the Employees table. Didi runs the following code, making her the manager of the CEO and introducing a cycle:

```
UPDATE dbo.Employees SET mgrid = 14 WHERE empid = 1;
```

The Employees table currently contains the following cycle of employee IDs:

1→3→7→9→14→1.

As a baseline, I'll use one of the solutions I covered earlier, which constructs an enumerated path. In my examples, I'll use a CTE solution, but of course you can apply the same logic to the UDF solution in SQL Server 2000.

Simply put, a cycle is detected when you follow a path leading to a given node if its parent's path already contains the child node ID. You can keep track of cycles by maintaining a *cycle* column, which will contain 0 if no cycle was detected and 1 if one was detected. In the anchor member of the solution CTE, the *cycle* column value is simply the constant 0, because obviously there's no cycle at the root level. In the recursive member's query, use a LIKE predicate to check whether the parent's path contains the child node ID. Return 1 if it does and 0 otherwise. Note the importance of the dots at both the beginning and end of both the path and the pattern—without the dots, you will get an unwanted match for employee ID *n* (for example *n = 3*) if the path contains employee ID *nm* (for example *m = 15, nm = 315*). Listing 9-21 shows the code that returns a subtree with an enumerated path calculation and has the addition of the *cycle* column calculation. If you run the code in Listing 9-21, it will always break after 100 levels (the default MAXRECURSION value) because cycles are detected but not avoided.

Listing 9-21: Detecting cycles, CTE solution

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Obviously root has no cycle
    0 AS cycle
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
```

```
          AS VARCHAR(MAX)) AS path,
  -- Cycle detected if parent's path contains child's id
  CASE WHEN P.path LIKE '%.' + CAST(C.empid AS VARCHAR(10)) + '.%'
    THEN 1 ELSE 0 END
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, empname, cycle, path
FROM SubsCTE;
```

You need to avoid cycles, or in other words, not pursue paths for which cycles are detected. To achieve this, simply add a filter to the recursive member that returns a child only if its parent's *cycle* value is 0. The code in Listing 9-22 includes this cycle avoidance logic, generating the output shown in Table 9-29.

Listing 9-22: Not pursuing cycles, CTE solution

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
          AS VARCHAR(MAX)) AS path,
    -- Obviously root has no cycle
    0 AS cycle
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
          AS VARCHAR(MAX)) AS path,
    -- Cycle detected if parent's path contains child's id
    CASE WHEN P.path LIKE '%.' + CAST(C.empid AS VARCHAR(10)) + '.%'
      THEN 1 ELSE 0 END
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
      AND P.cycle = 0 -- do not pursue branch for parent with cycle
)
SELECT empid, empname, cycle, path
FROM SubsCTE;
```

Table 9-29: Employees with Cycles not Pursued

empid	empname	cycle	path
1	David	0	.1
2	Eitan	0	.1.2
3	Ina	0	.1.3
7	Aaron	0	.1.3.7
11	Gabriel	0	.1.3.7.11
9	Rita	0	.1.3.7.9
12	Emilia	0	.1.3.7.9.12
13	Michael	0	.1.3.7.9.13
14	Didi	0	.1.3.7.9.14
1	David	1	.1.3.7.9.14.1

4	Seraph	0	.1.2.4
5	Jiru	0	.1.2.5
6	Steve	0	.1.2.6
10	Sean	0	.1.2.5.10
8	Lilach	0	.1.2.5.8

Notice in the output that the second time employee 1 was reached, a cycle was detected for it, and the path was not pursued any further. In a cyclic graph, that's all the logic you usually need to add. In our case, the cycle indicates a problem with the data that needs to be fixed. To isolate only the cyclic path (in our case, .1.3.7.9.14.1.), simply add the filter *cycle = 1* to the outer query as shown in Listing 9-23.

Listing 9-23: Isolating cyclic paths, CTE solution

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS

(
  SELECT empid, empname, 0 AS lvl,
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Obviously root has no cycle
    0 AS cycle
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Cycle detected if parent's path contains child's id
    CASE WHEN P.path LIKE '%.' + CAST(C.empid AS VARCHAR(10)) + '.%'
      THEN 1 ELSE 0 END
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
      AND P.cycle = 0
)
SELECT path FROM SubsCTE WHERE cycle = 1;
```

Now that the cyclic path has been identified, you can fix the data by running the following code:

```
UPDATE dbo.Employees SET mgrid = NULL WHERE empid = 1;
```

Didi will probably find herself unemployed.

Materialized Path

So far I presented solutions where paths were computed when the code was executed. In the materialized path solution, the paths will be stored so that they need not be computed repeatedly. You basically store an enumerated path and a level for each node of the tree in two additional columns. The solution applies only to trees—possibly forests.

There are two main advantages of this approach over the iterative/recursive approach. Queries are simpler and set-based (without relying on recursive CTEs). Also, queries typically perform much faster, as they can rely on indexing of the path.

However, now that you have two additional attributes in the table, you need to keep them in sync with the tree as it undergoes changes. The cost of modifications will determine whether it's reasonable to synchronize the path and level values with every change in the tree. For example, what is the effect of adding a new leaf to the tree? I like to refer to the effect of such a modification informally as the "shake effect." Fortunately, as I will elaborate shortly, the shake effect of adding new leaves is minor. Also, the effect of dropping or moving a small subtree is typically not very significant.

The enumerated path can get lengthy when the tree is deep—in other words, when there are many levels of managers. SQL Server limits the size of index keys to 900 bytes. To achieve the performance benefits of an index on the path column, you will limit it to 900 bytes. Before getting concerned by this fact, try thinking in practical terms: 900 bytes can contain trees with hundreds of levels. Will your tree ever reach more than hundreds of levels? I'll admit that I never had to model a hierarchy with hundreds of levels. In short, apply common sense and think in practical terms.

Maintaining Data

First run the code in Listing 9-24 to create the Employees table with the new *lvl* and *path* columns.

Listing 9-24: Data definition language for employees with materialized paths

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Employees') IS NOT NULL
  DROP TABLE dbo.Employees;
GO
CREATE TABLE dbo.Employees
(
   empid   INT          NOT NULL PRIMARY KEY NONCLUSTERED,
   mgrid   INT          NULL REFERENCES dbo.Employees,
   empname VARCHAR(25)  NOT NULL,
   salary  MONEY        NOT NULL,
   lvl     INT          NOT NULL,
   path    VARCHAR(900) NOT NULL UNIQUE CLUSTERED
);
CREATE UNIQUE INDEX idx_unc_mgrid_empid ON dbo.Employees(mgrid, empid);
GO
```

To handle modifications in a tree, it's recommended to use stored procedures that will also take care of the *lvl* and *path* values. Alternatively, you can use triggers, and their logic will be very similar to that in the stored procedures below.

Adding Employees Who Manage No One (Leaves)

Let's start with handling inserts. The logic of the insert procedure is simple. If the new employee is a root employee (that is, the manager ID is null), its level is 0 and its path is *'.'* + *employee id* + *'.'*. Otherwise, its level is the parent's level plus 1, and its path is: *parent path* + *'employee id* + *'.'*. As you can figure out, the shake effect here is minor. There's no need to make any changes to other employees, and to calculate the new employee's *lvl* and *path* values, you only need to query the employee's parent.

Run the code in Listing 9-25 to create the usp_insertemp stored procedure, and run the code in Listing 9-26 to populate the Employees table with sample data.

Listing 9-25: Creation script for the usp_insertemp procedure

```
---------------------------------------------------------------------
-- Stored Procedure: usp_insertemp,
--    Inserts new employee who manages no one into the table
---------------------------------------------------------------------
USE tempdb;
GO
IF OBJECT_ID('dbo.usp_insertemp') IS NOT NULL
  DROP PROC dbo.usp_insertemp;
GO
CREATE PROC dbo.usp_insertemp
   @empid   INT,
   @mgrid   INT,
   @empname VARCHAR(25),
   @salary  MONEY
AS

SET NOCOUNT ON;
```

```
-- Handle case where the new employee has no manager (root)
IF @mgrid IS NULL
  INSERT INTO dbo.Employees(empid, mgrid, empname, salary, lvl, path)
    VALUES(@empid, @mgrid, @empname, @salary,
      0, '.' + CAST(@empid AS VARCHAR(10)) + '.');
-- Handle subordinate case (non-root)
ELSE
   INSERT INTO dbo.Employees(empid, mgrid, empname, salary, lvl, path)
     SELECT @empid, @mgrid, @empname, @salary,
       lvl + 1, path + CAST(@empid AS VARCHAR(10)) + '.'
     FROM dbo.Employees
     WHERE empid = @mgrid;
GO
```

Listing 9-26: Sample data for employees with path

```
EXEC dbo.usp_insertemp
  @empid = 1, @mgrid = NULL, @empname = 'David', @salary = $10000.00;
EXEC dbo.usp_insertemp
  @empid = 2, @mgrid = 1, @empname = 'Eitan', @salary = $7000.00;
EXEC dbo.usp_insertemp
  @empid = 3, @mgrid = 1, @empname = 'Ina', @salary = $7500.00;
EXEC dbo.usp_insertemp
  @empid = 4, @mgrid = 2, @empname = 'Seraph', @salary = $5000.00;
EXEC dbo.usp_insertemp
  @empid = 5, @mgrid = 2, @empname = 'Jiru', @salary = $5500.00;
EXEC dbo.usp_insertemp
  @empid = 6, @mgrid = 2, @empname = 'Steve', @salary = $4500.00;
EXEC dbo.usp_insertemp
  @empid = 7, @mgrid = 3, @empname = 'Aaron', @salary = $5000.00;
EXEC dbo.usp_insertemp
  @empid = 8, @mgrid = 5, @empname = 'Lilach', @salary = $3500.00;

EXEC dbo.usp_insertemp
  @empid = 9, @mgrid = 7, @empname = 'Rita', @salary = $3000.00;
EXEC dbo.usp_insertemp
  @empid = 10, @mgrid = 5, @empname = 'Sean', @salary = $3000.00;
EXEC dbo.usp_insertemp
  @empid = 11, @mgrid = 7, @empname = 'Gabriel', @salary = $3000.00;
EXEC dbo.usp_insertemp
  @empid = 12, @mgrid = 9, @empname = 'Emilia', @salary = $2000.00;
EXEC dbo.usp_insertemp
  @empid = 13, @mgrid = 9, @empname = 'Michael', @salary = $2000.00;
EXEC dbo.usp_insertemp
  @empid = 14, @mgrid = 9, @empname = 'Didi', @salary = $1500.00;
```

Run the following query to examine the resulting contents of Employees, as shown in Table 9-30:

```
SELECT empid, mgrid, empname, salary, lvl, path
FROM dbo.Employees
ORDER BY path;
```

Table 9-30: Employees with Materialized Path

empid	mgrid	empname	salary	lvl	path
1	NULL	David	10000.0000	0	.1
2	1	Eitan	7000.0000	1	.1.2
4	2	Seraph	5000.0000	2	.1.2.4
5	2	Jiru	5500.0000	2	.1.2.5
10	5	Sean	3000.0000	3	.1.2.5.10
8	5	Lilach	3500.0000	3	.1.2.5.8
6	2	Steve	4500.0000	2	.1.2.6
3	1	Ina	7500.0000	1	.1.3

7	3	Aaron	5000.0000	2	.1.3.7
11	7	Gabriel	3000.0000	3	.1.3.7.11
9	7	Rita	3000.0000	3	.1.3.7.9
12	9	Emilia	2000.0000	4	.1.3.7.9.12
13	9	Michael	2000.0000	4	.1.3.7.9.13
14	9	Didi	1500.0000	4	.1.3.7.9.14

Moving a Subtree

Moving a subtree is a bit tricky. A change in someone's manager affects the row for that employee and for all of his or her subordinates. The inputs are the root of the subtree and the new parent (manager) of that root. The level and path values of all employees in the subtree are going to be affected. So you need to be able to isolate that subtree and also figure out how to revise the level and path values of all the subtree's members. To isolate the affected subtree, you join the row for the root (R) with the Employees table (E) based on *E.path LIKE R.path* + *'%'*. To calculate the revisions in level and path, you need access to the rows of both the old manager of the root (OM) and the new one (NM). The new level value for all nodes is their current level value plus the difference in levels between the new manager's level and the old manager's level. For example, if you move a subtree to a new location so that the difference in levels between the new manager and the old one is 2, you need to add 2 to the level value of all employees in the affected subtree. Similarly, to amend the path value of all nodes in the subtree, you need to remove the prefix containing the root's old manager's path and substitute it with the new manager's path. This can be achieved simply by using the STUFF function.

Run the code in Listing 9-27 to create the usp_movesubtree stored procedure, which implements the logic I just described.

Listing 9-27: Creation script for the usp_movesubtree procedure

```
---------------------------------------------------------------------
-- Stored Procedure: usp_movesubtree,
--   Moves a whole subtree of a given root to a new location
--   under a given manager
---------------------------------------------------------------------
USE tempdb;
GO
IF OBJECT_ID('dbo.usp_movesubtree') IS NOT NULL
  DROP PROC dbo.usp_movesubtree;
GO
CREATE PROC dbo.usp_movesubtree
  @root INT,
  @mgrid INT
AS

SET NOCOUNT ON;

BEGIN TRAN;
  -- Update level and path of all employees in the subtree (E)
  -- Set level =
  --   current level + new manager's level - old manager's level
  -- Set path =
  --   in current path remove old manager's path
  --   and substitute with new manager's path
UPDATE E
  SET lvl = E.lvl + NM.lvl - OM.lvl,
      path = STUFF(E.path, 1, LEN(OM.path), NM.path)
  FROM dbo.Employees AS E          -- E = Employees    (subtree)
    JOIN dbo.Employees AS R        -- R = Root         (one row)
      ON R.empid = @root
      AND E.path LIKE R.path + '%'
    JOIN dbo.Employees AS OM       -- OM = Old Manager (one row)
      ON OM.empid = R.mgrid
    JOIN dbo.Employees AS NM       -- NM = New Manager (one row)
      ON NM.empid = @mgrid;

  -- Update root's new manager
  UPDATE dbo.Employees SET mgrid = @mgrid WHERE empid = @root;
COMMIT TRAN;
```

```
GO
```

The implementation of this stored procedure is simplistic and is provided for demonstration purposes. Good behavior is not guaranteed for invalid parameter choices. To make this procedure more robust, you should also check the inputs to make sure that attempts to make someone his or her own manager or to generate cycles are rejected. For example, this can be achieved by using an EXISTS predicate with a SELECT statement that first generates a result set with the new paths, and checking that the employee IDs do not appear in their managers' path.

To test the procedure, first examine the tree shown in Table 9-31 before moving the subtree:

```
SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
FROM dbo.Employees
ORDER BY path;
```

Table 9-31: Employees before Moving Subtree

empid	empname	lvl	path
1	David	0	.1
2	| Eitan	1	.1.2
4	| | Seraph	2	.1.2.4
5	| | Jiru	2	.1.2.5
10	| | | Sean	3	.1.2.5.10
8	| | | Lilach	3	.1.2.5.8
6	| | Steve	2	.1.2.6
3	| Ina	1	.1.3
7	| | Aaron	2	.1.3.7
11	| | | Gabriel	3	.1.3.7.11
9	| | | Rita	3	.1.3.7.9
12	| | | | Emilia	4	.1.3.7.9.12
13	| | | | Michael	4	.1.3.7.9.13
14	| | | | Didi	4	.1.3.7.9.14

Then run the following code to move Aaron's subtree under Sean, and examine the result tree shown in Table 9-32 to verify that the subtree moved correctly:

```
BEGIN TRAN;

  EXEC dbo.usp_movesubtree
  @root = 7,
  @mgrid = 10;

  -- After moving subtree
  SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
  FROM dbo.Employees
  ORDER BY path;

ROLLBACK TRAN; -- rollback used in order not to apply the change
```

Table 9-32: Employees After Moving Subtree

empid	empname	lvl	path
1	David	0	.1
2	| Eitan	1	.1.2
4	| | Seraph	2	.1.2.4
5	| | Jiru	2	.1.2.5
10	| | | Sean	3	.1.2.5.10

7	| | | | Aaron	4	.1.2.5.10.7
11	| | | | | Gabriel	5	.1.2.5.10.7.11
9	| | | | | Rita	5	.1.2.5.10.7.9
12	| | | | | | Emilia	6	.1.2.5.10.7.9.12
13	| | | | | | Michael	6	.1.2.5.10.7.9.13
14	| | | | | | Didi	6	.1.2.5.10.7.9.14
8	| | | Lilach	3	.1.2.5.8
6	| | Steve	2	.1.2.6
3	| Ina	1	.1.3

Note The change is rolled back for demonstration only, so the data is the same at the start of each test script.

Removing a Subtree

Removing a subtree is a simple task. You just delete all employees whose path value has the subtree's root path as a prefix.

To test this solution, first examine the current state of the tree shown in Table 9-33 by running the following query:

```
SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
FROM dbo.Employees
ORDER BY path;
```

Table 9-33: Employees Before Deleting Subtree

empid	empname	lvl	path
1	David	0	.1
2	| Eitan	1	.1.2
4	| | Seraph	2	.1.2.4
5	| | Jiru	2	.1.2.5
10	| | | Sean	3	.1.2.5.10
8	| | | Lilach	3	.1.2.5.8
6	| | Steve	2	.1.2.6
3	| Ina	1	.1.3
7	| | Aaron	2	.1.3.7
11	| | | Gabriel	3	.1.3.7.11
9	| | | Rita	3	.1.3.7.9
12	| | | | Emilia	4	.1.3.7.9.12
13	| | | | Michael	4	.1.3.7.9.13
14	| | | | Didi	4	.1.3.7.9.14

Issue the following code, which first removes Aaron and his subordinates and then displays the resulting tree shown in Table 9-34:

```
BEGIN TRAN;

  DELETE FROM dbo.Employees
  WHERE path LIKE
    (SELECT M.path + '%'
     FROM dbo.Employees as M
     WHERE M.empid = 7);

  -- After deleting subtree
  SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
  FROM dbo.Employees
  ORDER BY path;
```

```
ROLLBACK TRAN; -- rollback used in order not to apply the change
```

Table 9-34: Employees After Deleting a Subtree

empid	empname	lvl	path
1	David	0	.1
2	| Eitan	1	.1.2
4	| | Seraph	2	.1.2.4
5	| | Jiru	2	.1.2.5
10	| | | Sean	3	.1.2.5.10
8	| | | Lilach	3	.1.2.5.8
6	| | Steve	2	.1.2.6
3	| Ina	1	.1.3

Querying

Querying data in the materialized path solution is simple and elegant. For subtree-related requests, the optimizer can always use a clustered or covering index that you create on the *path* column. If you create a nonclustered, noncovering index on the *path* column, the optimizer still will be able to use it if the query is selective enough.

Let's review typical requests from a tree. For each request, I'll provide a sample query followed by its output (shown in Table 9-35).

Return the subtree with a given root:

```
SELECT REPLICATE(' | ', E.lvl - M.lvl) + E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3 -- root
    AND E.path LIKE M.path + '%'
      ORDER BY E.path;
```

Table 9-35: Subtree with a Given Root

Ina
| Aaron
| | Gabriel
| | Rita
| | | Emilia
| | | Michael
| | | Didi

The query joins two instances of Employees. One represents the managers (*M*) and is filtered by the given root employee. The other represents the employees in the subtree (*E*). The subtree is identified using the following logical expression in the join condition: *E.path LIKE M.path + '%'*, which identifies a subordinate if it contains the root's path as a prefix. Indentation is achieved by replicating a string ('|') as many times as the employee's level within the subtree. The output is sorted by the path of the employee.

To exclude the subtree's root (top level manager) from the output, simply add an underscore before the percent sign in the LIKE pattern:

```
SELECT REPLICATE(' | ', E.lvl - M.lvl - 1) + E.empname
FROM dbo.Employees AS E
```

```
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '_%'
ORDER BY E.path;
```

You will get the output shown in Table 9-36.

Table 9-36: Subtree of a Given Root, Excluding Root

Aaron
| Gabriel
| Rita
| | Emilia
| | Michael
| | Didi

With the additional underscore in the LIKE condition, an employee is returned only if its path starts with the root's path and has at least one subsequent character.

To return leaf nodes under a given root (including the root itself if it is a leaf), add a NOT EXISTS predicate to identify only employees that are not managers of another employee:

```
SELECT E.empid, E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '%'
WHERE NOT EXISTS
  (SELECT *
   FROM dbo.Employees AS E2
   WHERE E2.mgrid = E.empid);
```

You will get the output shown in Table 9-37.

Table 9-37: Leaf Nodes Under a Given Root

empid	empname
11	Gabriel
12	Emilia
13	Michael
14	Didi

To return a subtree with a given root, limiting the number of levels under the root, add a filter in the join condition that limits the level difference between the employee and the root:

```
SELECT REPLICATE(' | ', E.lvl - M.lvl) + E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '%'
    AND E.lvl - M.lvl <= 2
ORDER BY E.path;
```

You will get the output shown in Table 9-38.

Table 9-38: Subtree with a Given Root, Limiting Levels

Ina
| Aaron
| | Gabriel
| | Rita

To return only the nodes exactly *n* levels under a given root, use an equal to operator (=) to identify the specific level difference instead of a less than or equal to (<=) operator:

```
SELECT E.empid, E.empname
FROM dbo.Employees AS E

  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '%'
    AND E.lvl - M.lvl = 2;
```

You will get the output shown in Table 9-39.

Table 9-39: Nodes that Are Exactly *n* Levels Under a Given Root

empid	empname
11	Gabriel
9	Rita

To return management chain of a given node, you use a query similar to the subtree query, with one small difference—you filter a specific employee ID, as opposed to filtering a specific manager ID:

```
SELECT REPLICATE(' | ', M.lvl) + M.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON E.empid = 14
    AND E.path LIKE M.path + '%'
ORDER BY E.path;
```

You will get the output shown in Table 9-40.

Table 9-40: Management Chain of Employee 14

David
|Ina
| | Aaron
| | | Rita
| | | | Didi

You get all managers whose paths are a prefix of the given employee's path.

Note that there's an important difference in performance between requesting a subtree and requesting the ancestors, even though they look very similar. For each query, either *M.path* or *E.path* is a constant. If *M.path* is constant, *E.path LIKE M.path + '%'* uses an index, because it asks for all paths with a given prefix. If *E.path* is constant, it does not use an index, because it asks for all prefixes of a given path. The subtree query can seek within an index to the first path that meets the filter, and it can scan to the right until it gets to the last path that meets the filter. In other words, only the relevant paths in the index are accessed. While in the ancestors query, ALL paths must be scanned to check whether they match the filter. This means performing a full table/index scan. In large tables, this translates to a slow query. To handle ancestor requests more efficiently, you can create a function that accepts an employee ID as input, splits its path, and returns a table with the path's node IDs in separate rows. You can join this table with the tree and use index seek operations for the specific employee IDs in the path. The split function uses an auxiliary table of numbers, which I covered in Chapter 4 under the section "Auxiliary Table of Numbers." If you currently don't have a Nums table in tempdb, first create it by running the code in Listing 9-28.

Listing 9-28: Creating and populating auxiliary table of numbers

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Nums') IS NOT NULL
  DROP TABLE dbo.Nums;
GO
CREATE TABLE Nums(n INT NOT NULL PRIMARY KEY);
DECLARE @max AS INT, @rc AS INT;
SET @max = 8000;
SET @rc = 1;

INSERT INTO Nums VALUES(1);
WHILE @rc * 2 <= @max
BEGIN
  INSERT INTO dbo.Nums SELECT n + @rc FROM dbo.Nums;
  SET @rc = @rc * 2;
END

INSERT INTO dbo.Nums
  SELECT n + @rc FROM dbo.Nums WHERE n + @rc <= @max;
```

Run the code in Listing 9-29 to create the *fn_splitpath* function.

Listing 9-29: Creation script for the *fn_splitpath* function

```
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_splitpath') IS NOT NULL
  DROP FUNCTION dbo.fn_splitpath;
GO
CREATE FUNCTION dbo.fn_splitpath(@empid AS INT) RETURNS TABLE
AS
RETURN
  SELECT
    n - LEN(REPLACE(LEFT(path, n), '.', '')) AS pos,
    CAST(SUBSTRING(path, n + 1,
           CHARINDEX('.', path, n+1) - n - 1) AS INT) AS empid
    FROM dbo.Employees
      JOIN dbo.Nums
        ON empid = @empid
        AND n < LEN(path)
        AND SUBSTRING(path, n, 1) = '.'
GO
```

You can find details on the logic behind the split technique that the function implements in Chapter 5 under the section "Separating Elements." To test the function, run the following code, which splits employee 14's path and generates the output shown in Table 9-41:

```
SELECT pos, empid FROM dbo.fn_splitpath(14);
```

Table 9-41: Output of the *fn_splitpath* Function

pos	empid
1	1
2	3
3	7
4	9
5	14

Now to get the management chain of a given employee, simply join the table returned by the function with the Employees table:

```
SELECT REPLICATE(' | ', lvl) + empname
FROM dbo.fn_splitpath(14) AS SP
  JOIN dbo.Employees AS E
    ON E.empid = SP.empid
ORDER BY path;
```

Nested Sets

Nested sets is one of the most beautiful and intellectually stimulating solutions I've ever seen for modeling trees.

More Info Joe Celko has extensive coverage of the Nested Sets model in his writings. You can find Joe Celko's coverage of nested sets in his book, *Joe Celko's Trees and Hierarchies in SQL for Smarties* (Morgan-Kaufmann, 2004).

Here I will cover T-SQL applications of the model, which for the most part work in SQL Server 2005 only because they use new features such as recursive CTEs and the ROW_NUMBER function.

The main advantages of the nested sets solution are simple and fast queries, which I'll describe later, and no level limit. However, alas, with large data sets, the solution's practicality is usually limited to static trees. For dynamic environments, the solution is limited to small trees (possibly large forests, but ones that consist of small trees).

Instead of representing a tree as an adjacency list (parent/child relationship), this solution models the tree relationships as nested sets. A parent is represented in the nested sets model as a containing set and a child as a contained set. Set containment relationships are represented with two integer values assigned to each set: left and right. For all sets: a set's left value is smaller than all contained sets' left values, and a set's right value is higher than all contained sets' right values. Naturally, this containment relationship is transitive in terms of *n*-level relationships (ancestor/descendant). The queries are based on these nested sets relationships. Logically, it's as if a set spreads two arms around all its contained sets.

Assigning Left and Right Values

Figure 9-5 provides a graphical visualization of the Employees hierarchy with the left and right values assigned to each employee.

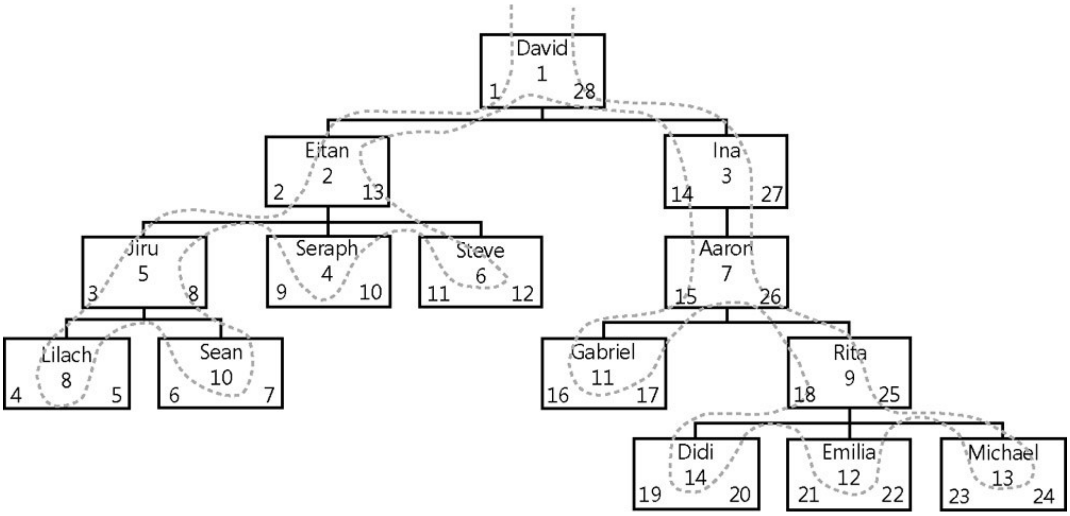

Figure 9-5: Employees hierarchy as nested sets

The curved line that walks the tree represents the order of assignment of the left and right values. Note that the model allows you to choose in which order you assign values to siblings. In this particular case, I chose to traverse siblings by employee name order.

You start with the root, traversing the tree counterclockwise. Every time you enter a node, you increment a counter and set it as the node's left value. Every time you leave a node, you increment the counter and set it as the node's right value. This algorithm can be implemented to the letter as an iterative/recursive routine that assigns each node with left and right values. However, such an implementation requires traversing the tree a node at a time, which can be very slow. I'll show an algorithm that traverses the tree a level at a time, which is faster. The core algorithm is based on logic I discussed earlier in the chapter, traversing the tree a level at a time and calculating binary sort paths. To understand this algorithm, it will help to examine Figure 9-6.

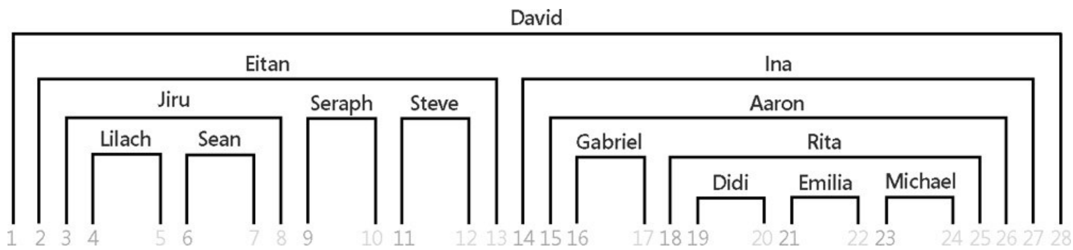

Figure 9-6: Illustration of nested sets model

The figure illustrates each employee as spreading two arms around its subordinates. Left and right values can now be assigned to the different arms by simply incrementing a counter from left to right. Keep this illustration in mind, as it's the key to understanding the solution that I will present.

Again, the baseline is the original algorithm that traverses a subtree a level at a time and constructs a binary sort path based on desired sibling sorting (for example, *empname, empid*).

Note To get good performance, you should create an index on the parent ID and sort columns—for example, *(mgrid, empname, empid)*.

Instead of generating one row for each node (as was the case in the earlier solutions for generating sort values based on a binary path), you generate two rows by cross-joining each level with an auxiliary table that has two numbers: *n=1* representing the left arm, and *n=2* representing the right arm. Still, the binary paths are constructed from row numbers, but in this case the arm number is taken into consideration besides the other sort elements (for example, *empname, empid, n*). The query that returns the next level of subordinates returns the subordinates of the left arm only—again, cross-joined with two numbers (*n=1, n=2*) to generate two arms for each node.

The code in Listing 9-30 has the CTE implementation of this algorithm and generates the output shown in Table 9-42. The purpose of this code is to generate two binary sort paths for each employee, which will later be used to calculate left and right values. Before you run this code, make sure you have the original Employees table in the tempdb database. If you don't, rerun the code in Listing 9-1 first.

Table 9-42: Binary Sort Paths Representing Nested Sets Relationships

empid	lvl	n	sortpath
1	0	1	0x00000001
2	1	1	0x0000000100000001
5	2	1	0x000000010000000100000001
8	3	1	0x00000001000000010000000100000001
8	3	2	0x00000001000000010000000100000002
10	3	1	0x00000001000000010000000100000003
10	3	2	0x00000001000000010000000100000004
5	2	2	0x000000010000000100000002
4	2	1	0x000000010000000100000003
4	2	2	0x000000010000000100000004
6	2	1	0x000000010000000100000005
6	2	2	0x000000010000000100000006
2	1	2	0x0000000100000002
3	1	1	0x0000000100000003
7	2	1	0x000000010000000300000001
11	3	1	0x00000001000000030000000100000001
11	3	2	0x00000001000000030000000100000002
9	3	1	0x00000001000000030000000100000003
14	4	1	0x0000000100000003000000010000000300000001
14	4	2	0x0000000100000003000000010000000300000002
12	4	1	0x0000000100000003000000010000000300000003
12	4	2	0x0000000100000003000000010000000300000004
13	4	1	0x0000000100000003000000010000000300000005
13	4	2	0x0000000100000003000000010000000300000006
9	3	2	0x00000001000000030000000100000004
7	2	2	0x000000010000000300000002
3	1	2	0x0000000100000004
1	0	2	0x00000002

Listing 9-30: Producing binary sort paths representing nested sets relationships

```
USE tempdb;
GO
-- Create index to speed sorting siblings by empname, empid
CREATE UNIQUE INDEX idx_unc_mgrid_empname_empid
ON dbo.Employees(mgrid, empname, empid);
GO

DECLARE @root AS INT;
SET @root = 1;

-- CTE with two numbers: 1 and 2
WITH TwoNumsCTE
AS
(
  SELECT 1 AS n UNION ALL SELECT 2
),
```

```
-- CTE with two binary sort paths for each node:
--   One smaller than descendants sort paths
--   One greater than descendants sort paths
SortPathCTE

AS
(
  SELECT empid, 0 AS lvl, n,
    CAST(n AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees CROSS JOIN TwoNumsCTE
  WHERE empid = @root

  UNION ALL

SELECT C.empid, P.lvl + 1, TN.n,
  P.sortpath + CAST(
    (-1+ROW_NUMBER() OVER(PARTITION BY C.mgrid
                     -- *** determines order of siblings ***
                     ORDER BY C.empname, C.empid))/2*2+TN.n
    AS BINARY(4))
  FROM SortPathCTE AS P
    JOIN dbo.Employees AS C
      ON P.n = 1
      AND C.mgrid = P.empid
    CROSS JOIN TwoNumsCTE AS TN
)
SELECT * FROM SortPathCTE
ORDER BY sortpath;
```

TwoNumsCTE is the auxiliary table with two numbers representing the two arms. Of course, you could use a real Nums table if you wanted, instead of generating a virtual one.

Two sort paths are generated for each node. The left one is represented by *n=1*, and the right one by *n=2*. Notice that for a given node, the left sort path is smaller than all left sort paths of subordinates, and the right sort path is greater than all right sort paths of subordinates. The sort paths will be used to generate the left and right values in Figure 9-6. You need to generate left and right integer values to represent the nested sets relationships between the employees. To assign the integer values to the arms (*sortval*), simply use the ROW_NUMBER function based on *sortpath* order. Finally, to return one row for each employee containing the left and right integer values, group the rows by employee and level, and return the *MIN(sortval)* as the left value and *MAX(sortval)* as the right value. The complete solution to generate left and right values is shown in Listing 9-31 and generates the output shown in Table 9-43.

Table 9-43: Left and Right Values Generated with a CTE

empid	lvl	lft	rgt
1	0	1	28
2	1	2	13
5	2	3	8
8	3	4	5
10	3	6	7
4	2	9	10
6	2	11	12
3	1	14	27
7	2	15	26
11	3	16	17
9	3	18	25
14	4	19	20
12	4	21	22

13	4	23	24

Listing 9-31: CTE code that creates nested sets relationships

```
DECLARE @root AS INT;
SET @root = 1;

-- CTE with two numbers: 1 and 2
WITH TwoNumsCTE
AS
(
  SELECT 1 AS n UNION ALL SELECT 2
),
-- CTE with two binary sort paths for each node:
--   One smaller than descendants sort paths
--   One greater than descendants sort paths
SortPathCTE
AS
(
  SELECT empid, 0 AS lvl, n,
    CAST(n AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees CROSS JOIN TwoNumsCTE
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, P.lvl + 1, TN.n,
    P.sortpath + CAST(
      (-1+ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        -- *** determines order of siblings ***
                        ORDER BY C.empname, C.empid))/2*2+TN.n
      AS BINARY(4))
    FROM SortPathCTE AS P
      JOIN dbo.Employees AS C
        ON P.n = 1
        AND C.mgrid = P.empid
      CROSS JOIN TwoNumsCTE AS TN
),
-- CTE with Row Numbers Representing sortpath Order
SortCTE
AS
(
  SELECT empid, lvl,
    ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval
  FROM SortPathCTE
),
-- CTE with Left and Right Values Representing
-- Nested Sets Relationships
NestedSetsCTE
AS
(
  SELECT empid, lvl, MIN(sortval) AS lft, MAX(sortval) AS rgt
  FROM SortCTE
  GROUP BY empid, lvl
)
SELECT * FROM NestedSetsCTE
ORDER BY lft;
```

The implementation of this algorithm in SQL Server 2000 is similar, but it's lengthier and slower, mainly because of the calculation of row numbers using identity values instead of the ROW_NUMBER function. You have to materialize interim results in a table to generate the identity values. For simplicity's sake, I'll show a solution with a UDF, where siblings are ordered by *empname, empid*. To create the *fn_empsnestedsets* UDF, run the code in Listing 9-32.

Listing 9-32: Creation script for the *fn_empsnestedsets* function

```
---------------------------------------------------------------------
-- Function: fn_empsnestedsets, Nested Sets Relationships
```

```
--
-- Input   : @root INT: Root of subtree
--
-- Output  : @NestedSets Table: employee id, level in the subtree,
--                              left and right values representing
--                              nested sets relationships
--
-- Process : * Loads subtree into @SortPath,
--             first root, then a level at a time.
--             Note: two instances of each employee are loaded;
--                   one representing left arm (n = 1),
--                   and one representing right (n = 2).
--             For each employee and arm, a binary path is constructed,
--             representing the nested sets position.
--             The binary path has 4 bytes for each of the employee's
--             ancestors. For each ancestor, the 4 bytes represent
--             its position in the level (calculated with identity).
--             Finally @SortPath will contain a pair of rows for each
--             employee along with a sort path representing the arm's
--             nested sets position.
--           * Next, the rows from @SortPath are loaded
--             into @SortVals, sorted by sortpath. After the load,
--             an integer identity column sortval holds sort values
--             representing the nested sets position of each arm.
--           * The data from @SortVals is grouped by employee,
--             generating the left and right values for each employee
--             in one row. The result set is loaded into the
--             @NestedSets table, which is the function's output.
--
---------------------------------------------------------------------
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_empsnestedsets') IS NOT NULL
  DROP FUNCTION dbo.fn_empsnestedsets;
GO
CREATE FUNCTION dbo.fn_empsnestedsets(@root AS INT)
  RETURNS @NestedSets TABLE
(
   empid INT NOT NULL PRIMARY KEY,
   lvl   INT NOT NULL,
   lft   INT NOT NULL,
   rgt   INT NOT NULL
)

AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;

-- @TwoNums: Table Variable with two numbers: 1 and 2
DECLARE @TwoNums TABLE(n INT NOT NULL PRIMARY KEY);
INSERT INTO @TwoNums(n) SELECT 1 AS n UNION ALL SELECT 2;

-- @SortPath: Table Variable with two binary sort paths
-- for each node:
--   One smaller than descendants sort paths
--   One greater than descendants sort paths
DECLARE @SortPath TABLE
(
   empid    INT            NOT NULL,
   lvl      INT            NOT NULL,
   n        INT            NOT NULL,
   sortpath VARBINARY(900) NOT NULL,
   rownum   INT            NOT NULL IDENTITY,
   UNIQUE(lvl, n, empid)
);

-- Load root into @SortPath
```

```
INSERT INTO @SortPath(empid, lvl, n, sortpath)
  SELECT empid, @lvl, n,
    CAST(n AS BINARY(4)) AS sortpath
  FROM dbo.Employees CROSS JOIN @TwoNums
  WHERE empid = @root

WHILE @@rowcount > 0
BEGIN
  SET @lvl = @lvl + 1;

-- Load next level into @SortPath
INSERT INTO @SortPath(empid, lvl, n, sortpath)
  SELECT C.empid, @lvl, TN.n, P.sortpath
  FROM @SortPath AS P
    JOIN dbo.Employees AS C
      ON P.lvl = @lvl - 1
      AND P.n = 1
      AND C.mgrid = P.empid
    CROSS JOIN @TwoNums AS TN
  -- *** Determines order of siblings ***
  ORDER BY C.empname, C.empid, TN.n;

  -- Update sort path to include child's position
  UPDATE @SortPath
    SET sortpath = sortpath + CAST(rownum AS BINARY(4))
  WHERE lvl = @lvl;
END

-- @SortVals: Table Variable with row numbers
-- representing sortpath order
DECLARE @SortVals TABLE

(
   empid   INT NOT NULL,
   lvl     INT NOT NULL,
   sortval INT NOT NULL IDENTITY
)

-- Load data from @SortPath sorted by sortpath
-- to generate sort values
INSERT INTO @SortVals(empid, lvl)
  SELECT empid, lvl FROM @SortPath ORDER BY sortpath;

-- Load data into @NestedSets, generating left and right
-- values representing nested sets relationships
INSERT INTO @NestedSets(empid, lvl, lft, rgt)
   SELECT empid, lvl, MIN(sortval), MAX(sortval)
   FROM @SortVals
   GROUP BY empid, lvl

   RETURN;
END
GO
```

To test the function, run the following code, which generates the output shown in Table 9-44:

```
SELECT * FROM dbo.fn_empsnestedsets(1)
ORDER BY lft;
```

Table 9-44: Left and Right Values Generated with a UDF

empid	lvl	lft	rgt
1	0	1	28
2	1	2	13
5	2	3	8

8	3	4	5
10	3	6	7
4	2	9	10
6	2	11	12
3	1	14	27
7	2	15	26
11	3	16	17
9	3	18	25
14	4	19	20
12	4	21	22
13	4	23	24

In the opening paragraph of the "Nested Sets" section, I mentioned that this solution is not adequate for large dynamic trees (trees that incur frequent changes). Suppose you stored left and right values in two additional columns in the Employees table. Note that you won't need the *mgrid* column in the table anymore, as the two additional columns with the left and right values are sufficient to answer requests for subordinates, ancestors, and so on. Consider the shake effect of adding a node to the tree. For example, take a look at Figures 9-5 and 9-6, and try to figure out the effect of adding a new subordinate to Steve. Steve has left and right values 11 and 12, respectively. The new node should get left and right values of 12 and 13, respectively. Steve's right value, and in fact all left and right values in the tree that were greater than or equal to 14, should be increased by two. On average, at least half the nodes in the tree must be updated every time a new node is inserted. As you can see here, the shake effect is very dramatic. That's why the nested sets solution is adequate for a large tree only if it's static, or if you need to run queries against a static snapshot of the tree periodically.

Nested sets can provide reasonably good performance with dynamic trees that are small (or forests with small trees)—for example, when maintaining forum discussions where each thread is a small independent tree in a forest. You can implement a solution that synchronizes the left and right values of the tree with every change. You can achieve this by using stored procedures, or even triggers, as long as the cost of modification is small enough to be bearable. I won't even get into variations of the nested sets model that maintain gaps between the values (that is, leave room to insert new leaves without as much work), as they are all ultimately limited.

To generate a table of employees (EmployeesNS) with the employee ID, employee name, salary, level, left, and right values, join the outer query of either the CTE or the UDF solution and use a SELECT INTO statement. Run the code in Listing 9-33 to create this as the EmployeesNS table with siblings ordered by *empname, empid*.

Listing 9-33: Materializing nested sets relationships in a table

```
SET NOCOUNT ON;
USE tempdb;
GO

DECLARE @root AS INT;
SET @root = 1;

WITH TwoNumsCTE
AS
(
  SELECT 1 AS n UNION ALL SELECT 2
),
SortPathCTE
AS
(
  SELECT empid, 0 AS lvl, n,
    CAST(n AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees CROSS JOIN TwoNumsCTE
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, P.lvl + 1, TN.n,
    P.sortpath + CAST(
```

```
      ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        -- *** determines order of siblings ***
                        ORDER BY C.empname, C.empid, TN.n)
      AS BINARY(4))
  FROM SortPathCTE AS P
    JOIN dbo.Employees AS C
      ON P.n = 1
      AND C.mgrid = P.empid
    CROSS JOIN TwoNumsCTE AS TN
),
SortCTE
AS
(
  SELECT empid, lvl,
    ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval
  FROM SortPathCTE
),
NestedSetsCTE
AS
(
  SELECT empid, lvl, MIN(sortval) AS lft, MAX(sortval) AS rgt
  FROM SortCTE
  GROUP BY empid, lvl
)
SELECT E.empid, E.empname, E.salary, NS.lvl, NS.lft, NS.rgt
INTO dbo.EmployeesNS
FROM NestedSetsCTE AS NS
  JOIN dbo.Employees AS E
    ON E.empid = NS.empid;

ALTER TABLE dbo.EmployeesNS ADD PRIMARY KEY NONCLUSTERED(empid);
CREATE UNIQUE CLUSTERED INDEX idx_unc_lft_rgt ON dbo.EmployeesNS(lft, rgt);
GO
```

Querying

The EmployeesNS table models a tree of employees as nested sets. Querying is simple, elegant, and fast with the index on left and right values.

In the following section, I'll present common requests against a tree and the query solution for each, followed by the output of the query.

Return the subtree of a given root, generating the output shown in Table 9-45:

```
SELECT C.empid, REPLICATE(' | ', C.lvl - P.lvl) + C.empname AS empname
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON P.empid = 3
    AND C.lft >= P.lft AND C.rgt <= P.rgt
ORDER BY C.lft;
```

Table 9-45: Subtree of a Given Root

empid	empname
3	Ina
7	| Aaron
11	| | Gabriel
9	| | Rita
14	| | | Didi
12	| | | Emilia
13	| | | Michael

The query joins two instances of EmployeesNS. One represents the parent (*P*) and is filtered by the given root. The other represents the child (*C*). The two are joined based on the child's left being greater than or equal to the parent's left, and the child's right being smaller than or equal to the parent's right. Indentation of the output is achieved by replicating a string (' | ') child level minus parent level times. The output is sorted by the child's left value, which by definition represents correct hierarchical sorting, and the desired sort of siblings. This subtree query is used as the baseline for most of the following queries.

If you want to exclude the subtree's root node from the output, simply use greater than (>) and less than (<) operators instead of greater than or equal to (>=) and less than or equal to (<=) operators. To the subtree query, add a filter in the join condition that returns only nodes where the child's level minus the parent's level is smaller than or equal to the requested number of levels under the root.

Return the subtree of a given root, limiting 2 levels of subordinates under the root, generating the output shown in Table 9-46:

```
SELECT C.empid, REPLICATE(' | ', C.lvl - P.lvl) + C.empname AS empname
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON P.empid = 3
    AND C.lft >= P.lft AND C.rgt <= P.rgt
    AND C.lvl - P.lvl <= 2
ORDER BY C.lft;
```

Table 9-46: Subtree of a Given Root, with Level Limit

empid	empname
3	Ina
7	| Aaron
11	| | Gabriel
9	| | Rita

Return leaf nodes under a given root, generating the output shown in Table 9-47:

```
SELECT C.empid, C.empname
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON P.empid = 3
    AND C.lft >= P.lft AND C.rgt <= P.rgt
    AND C.rgt - C.lft = 1;
```

Table 9-47: Leaf Nodes Under a Given Root

empid	empname
11	Gabriel
12	Emilia
13	Michael
14	Didi

A leaf node is a node for which the right value is greater than the left value by 1 (no subordinates). Add this filter to the join condition of the subtree query. As you can see, the nested sets solution allows for dramatically faster identification of leaf nodes than other solutions using a NOT EXISTS predicate.

Return the count of subordinates of each node, generating the output shown in Table 9-48:

```
SELECT empid, (rgt - lft - 1) / 2 AS cnt,
  REPLICATE(' | ', lvl) + empname AS empname
FROM dbo.EmployeesNS
ORDER BY lft;
```

Table 9-48: Count of

Subordinates of Each Node

empid	cnt	empname
1	13	David
2	5	| Eitan
5	2	| | Jiru
8	0	| | | Lilach
10	0	| | | Sean
4	0	| | Seraph
6	0	| | Steve
3	6	|Ina
7	5	| | Aaron
11	0	| | | Gabriel
9	3	| | | Rita
14	0	| | | | Didi
12	0	| | | | Emilia
13	0	| | | | Michael

Because each node accounts for exactly two *lft* and *rgt* values, and in our implementation no gaps exist, you can calculate the count of subordinates by accessing the subtree's root alone. The count is: *(rgt − lft − 1) / 2*.

Return all ancestors of a given node, generating the output shown in Table 9-49:

```
SELECT P.empid, P.empname, P.lvl
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON C.empid = 14
    AND C.lft >= P.lft AND C.rgt <= P.rgt;
```

Table 9-49: Ancestors of a Given Node

empid	empname	lvl
1	David	0
3	Ina	1
7	Aaron	2
9	Rita	3
14	Didi	4

The ancestors query is almost identical to the subtree query. The nested sets relationships remain the same. The only difference is that here you filter a specific child node ID, while in the subtree query you filtered a specific parent node ID.

When you're done querying the EmployeesNS table, don't forget to get rid of it:

```
DROP TABLE dbo.EmployeesNS;
```

Transitive Closure

The transitive closure of a directed graph G is the graph with the same vertices as G, and with an edge connecting each pair of nodes that are connected by a path (not necessarily containing just one edge) in G. The transitive closure helps answer a number of questions immediately, without the need to explore paths in the graph. For example, is David a manager of Aaron (directly or indirectly)? If the transitive closure of the Employees graph contains an edge from David to Aaron, he is. Does Double Espresso contain water? Can I drive from Los Angeles to New York? If the input graph contains the edges (a, b) and (b, c), there's a transitive relationship between a and c. The transitive closure will contain the edges (a, b), (b, c), and also (a, c). If David is the direct manager of Ina, and Ina is the direct manager of Aaron, David transitively is a manager of Aaron, or Aaron transitively is a subordinate of David.

There are problems related to transitive closure that deal with specialized cases of transitive relationships. An example is the "shortest path" problem, where you're trying to determine the shortest path between two nodes. For example, what's the shortest path between Los Angeles and New York?

In this section, I will describe iterative/recursive solutions for transitive closure and shortest path problems. In some of my examples, I will use CTEs that apply to SQL Server 2005. As with examples I presented earlier in the chapter, you can make adjustments and implement similar algorithms in SQL Server 2000 by using UDFs or stored procedures.

Note The performance of some of the solutions that I will show (specifically those that use recursive CTEs) degrades exponentially as the input graph grows. I'll present them for demonstration purposes because they are fairly simple and natural. They are adequate for fairly small graphs. There are efficient algorithms for transitive closure–related problems (for example, Floyd's and Warshall's algorithms) that can be implemented as "level at a time" (breadth-first) iterations. For details on those, please refer to http://www.nist.gov/dads/. I'll show efficient solutions provided by Steve Kass that can be applied to larger graphs.

Directed Acyclic Graph

The first problem that I will discuss is generating a transitive closure of a directed acyclic graph (DAG). Later I'll show you how to deal with undirected and cyclic graphs as well. Whether the graph is directed or undirected doesn't really complicate the solution significantly, while dealing with cyclic graphs does. The input DAG that I will use in my example is the BOM I used earlier in the chapter, which you create by running the code in Listing 9-2.

The code that generates the transitive closure of BOM is somewhat similar to solutions for the subgraph problem (that is, the parts explosion). Specifically, you traverse the graph a level at a time (or more accurately, you are using "breadth-first" search techniques). However, instead of returning only a root node here, the anchor member returns all first-level relationships in BOM. In most graphs, this simply means all existing source/target pairs. In our case, this means all assembly/part pairs where the assembly is not NULL. The recursive member joins the CTE representing the previous level or parent (*P*) with BOM representing the next level or child (*C*). It returns the original product id (*P*) as the source, and the child product id (*C*) as the target. The outer query returns the distinct assembly/part pairs. Keep in mind that multiple paths may lead to a part in BOM, but you need to return each unique pair only once.

Run the code in Listing 9-34 to generate the transitive closure of BOM shown in Table 9-50.

Listing 9-34: Transitive closure of BOM (DAG)

```
WITH BOMTC
AS
(
   -- Return all first-level containment relationships
   SELECT assemblyid, partid
   FROM dbo.BOM
   WHERE assemblyid IS NOT NULL

   UNION ALL

   -- Return next-level containment relationships
   SELECT P.assemblyid, C.partid
   FROM BOMTC AS P
     JOIN dbo.BOM AS C
       ON C.assemblyid = P.partid
)
-- Return distinct pairs that have
-- transitive containment relationships
SELECT DISTINCT assemblyid, partid
FROM BOMTC;
```

Table 9-50: Transitive Closure of BOM (DAG)

assemblyid	partid
1	6

1	7
1	10
1	13
1	14
2	6
2	7
2	10
2	11
2	13
2	14
3	6
3	7
3	11
3	12
3	14
3	16
3	17
4	9
4	12
4	14
4	16
4	17
5	9
5	12
5	14
5	16
5	17
10	13
10	14
12	14
12	16
12	17
16	17

This solution eliminates duplicate edges found in the BOMCTE by applying a DISTINCT clause in the outer query. A more efficient solution would be to avoid getting duplicates altogether by using a NOT EXISTS predicate in the query that runs repeatedly; such a predicate would filter newly found edges that do not appear in the set of edges that were already found. However, such an implementation will not be able to use a CTE because the recursive member in the CTE has access only to the "immediate previous level," as opposed to "all previous levels" obtained thus far. Instead, you can use a UDF that invokes the query that runs repeatedly in a loop and inserts each obtained level of nodes into a table variable. Run the code in Listing 9-35 to create the *fn_BOMTC* UDF, which implements this logic.

Listing 9-35: Creation script for the *fn_BOMTC* UDF

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

CREATE FUNCTION fn_BOMTC() RETURNS @BOMTC TABLE
(
```

```
   assemblyid INT NOT NULL,
   partid     INT NOT NULL,
   PRIMARY KEY (assemblyid, partid)
)
AS
BEGIN
  INSERT INTO @BOMTC(assemblyid, partid)
    SELECT assemblyid, partid
    FROM dbo.BOM
    WHERE assemblyid IS NOT NULL

  WHILE @@rowcount > 0
    INSERT INTO @BOMTC
    SELECT P.assemblyid, C.partid
    FROM @BOMTC AS P
      JOIN dbo.BOM AS C
        ON C.assemblyid = P.partid
      WHERE NOT EXISTS
        (SELECT * FROM @BOMTC AS P2
        WHERE P2.assemblyid = P.assemblyid
        AND P2.partid = C.partid);

    RETURN;
END
GO
```

Run the following code to query the function and you will get the output shown in Table 9-50:

```
SELECT assemblyid, partid FROM fn_BOMTC();
```

If you want to return all paths in BOM, along with the distance in levels between the parts, you use a similar algorithm with a few additions and revisions. You calculate the distance the same way you calculated the level value in the subgraph/subtree solutions. That is, the anchor assigns a constant distance of 1 for the first level, and the recursive member simply adds one in each iteration. Also, the path calculation is similar to the one used in the subgraph/subtree solutions. The anchor generates a path made of *'.'* + *source_id* + *'.'* + *target_id* + *'.'*. The recursive member generates it as: *parent's path* + *target_id* + *'.'*. Finally, the outer query simply returns all paths (without applying DISTINCT in this case).

Run the code in Listing 9-36 to generate all possible paths in BOM and their distances.

Listing 9-36: All paths in BOM

```
WITH BOMPaths
AS
(
  SELECT assemblyid, partid,
    1 AS distance, -- distance in first level is 1
    -- path in first level is .assemblyid.partid.
    '.' + CAST(assemblyid AS VARCHAR(MAX)) +
    '.' + CAST(partid AS VARCHAR(MAX)) + '.' AS path
  FROM dbo.BOM
  WHERE assemblyid IS NOT NULL

  UNION ALL

  SELECT P.assemblyid, C.partid,
    -- distance in next level is parent's distance + 1
    P.distance + 1,
    -- path in next level is parent_path.child_partid.
    P.path + CAST(C.partid AS VARCHAR(MAX)) + '.'
  FROM BOMPaths AS P
    JOIN dbo.BOM AS C
      ON C.assemblyid = P.partid
)
-- Return all paths
SELECT * FROM BOMPaths;
```

You will get the output shown in Table 9-51.

Table 9-51: All Paths in BOM

assemblyid	partid	distance	path
1	6	1	.1.6.
2	6	1	.2.6
3	6	1	.3.6.
1	7	1	.1.7.
2	7	1	.2.7.
3	7	1	.3.7.
4	9	1	.4.9.
5	9	1	.5.9.
1	10	1	.1.10.
2	10	1	.2.10.
2	11	1	.2.11.
3	11	1	.3.11.
3	12	1	.3.12.
4	12	1	.4.12.
5	12	1	.5.12.
10	13	1	.10.13.
1	14	1	.1.14.
2	14	1	.2.14.
10	14	1	.10.14.
12	14	1	.12.14.
12	16	1	.12.16.
16	17	1	.16.17.
12	17	2	.12.16.17.
5	14	2	.5.12.14.
5	16	2	.5.12.16.
5	17	3	.5.12.16.17.
4	14	2	.4.12.14.
4	16	2	.4.12.16.
4	17	3	.4.12.16.17.
3	14	2	.3.12.14.
3	16	2	.3.12.16.
3	17	3	.3.12.16.17.
2	13	2	.2.10.13.
2	14	2	.2.10.14.
1	13	2	.1.10.13.
1	14	2	.1.10.14.

To isolate only the shortest paths, add a second CTE (BOMMinDist) that groups all paths by assembly and part, returning the minimum distance for each group. And in the outer query, join the first CTE (BOMPaths) with BOMMinDist, based on *assembly*, *part*, and *distance* match to return the actual paths.

Run the code in Listing 9-37 to produce the shortest paths in BOM as shown in Table 9-52.

Listing 9-37: Shortest paths in BOM

```
WITH BOMPaths -- All paths
AS
(
  SELECT assemblyid, partid,
    1 AS distance,
    '.' + CAST(assemblyid AS VARCHAR(MAX)) +
    '.' + CAST(partid AS VARCHAR(MAX)) + '.' AS path
  FROM dbo.BOM
  WHERE assemblyid IS NOT NULL

  UNION ALL

  SELECT P.assemblyid, C.partid,
    P.distance + 1,
    P.path + CAST(C.partid AS VARCHAR(MAX)) + '.'
  FROM BOMPaths AS P
      JOIN dbo.BOM AS C
        ON C.assemblyid = P.partid
),
BOMMinDist AS -- Minimum distance for each pair
(
  SELECT assemblyid, partid, MIN(distance) AS mindist
  FROM BOMPaths
  GROUP BY assemblyid, partid
)
-- Shortest path for each pair
SELECT BP.*
FROM BOMMinDist AS BMD
  JOIN BOMPaths AS BP
    ON BMD.assemblyid = BP.assemblyid
    AND BMD.partid = BP.partid
    AND BMD.mindist = BP.distance;
```

Table 9-52: Shortest Paths in BOM

assemblyid	partid	distance	path
1	6	1	.1.6.
2	6	1	.2.6.
3	6	1	.3.6.
1	7	1	.1.7.
2	7	1	.2.7.
3	7	1	.3.7.
4	9	1	.4.9.
5	9	1	.5.9.
1	10	1	.1.10.
2	10	1	.2.10.
2	11	1	.2.11.
3	11	1	.3.11.
3	12	1	.3.12.
4	12	1	.4.12.
5	12	1	.5.12.
10	13	1	.10.13.
1	14	1	.1.14.
2	14	1	.2.14.
10	14	1	.10.14.

12	14	1	.12.14.
12	16	1	.12.16.
16	17	1	.16.17.
12	17	2	.12.16.17.
5	14	2	.5.12.14.
5	16	2	.5.12.16.
5	17	3	.5.12.16.17.
4	14	2	.4.12.14.
4	16	2	.4.12.16.
4	17	3	.4.12.16.17.
3	14	2	.3.12.14.
3	16	2	.3.12.16.
3	17	3	.3.12.16.17.
2	13	2	.2.10.13.
1	13	2	.1.10.13.

Undirected Cyclic Graph

Even though transitive closure is defined for a directed graph, you can also define and generate it for undirected graphs where each edge represents a two-way relationship. In my examples, I will use the Roads graph, which you create and populate by running the code in Listing 9-3. To see a visual representation of Roads, examine Figure 9-4. To apply the transitive closure and shortest path solutions to Roads, first convert it to a digraph by generating two directed edges from each existing edge:

```
SELECT city1 AS from_city, city2 AS to_city FROM dbo.Roads
UNION ALL
SELECT city2, city1 FROM dbo.Roads
```

For example, the edge (*JFK, ATL*) in the undirected graph will appear as the edges (*JFK, ATL*) and (*ATL, JFK*) in the digraph. The former represents the road from New York to Atlanta, and the latter represents the road from Atlanta to New York.

Because Roads is a cyclic graph, you also need to use the cycle-detection logic I described earlier in the chapter to avoid traversing cyclic paths. Armed with the techniques to generate a digraph out of an undirected graph and to detect cycles, you have all the tools you need to produce the transitive closure of roads.

Run the code in Listing 9-38 to generate the transitive closure of roads shown in Table 9-53.

Listing 9-38: Transitive closure of Roads (undirected cyclic graph)

```
WITH Roads2 -- Two rows for each pair (from-->to, to-->from)
AS
(
  SELECT city1 AS from_city, city2 AS to_city FROM dbo.Roads
  UNION ALL
  SELECT city2, city1 FROM dbo.Roads
),
RoadPaths AS
(
  -- Return all first-level reachability pairs
  SELECT from_city, to_city,
    -- path is needed to identify cycles
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  -- Return next-level reachability pairs
  SELECT F.from_city, T.to_city,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
```

```
    FROM RoadPaths AS F
      JOIN Roads2 AS T
        -- if to_city appears in from_city's path, cycle detected
        ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
                THEN 1 ELSE 0 END = 0
        AND F.to_city = T.from_city
)
-- Return Transitive Closure of Roads
SELECT DISTINCT from_city, to_city
FROM RoadPaths;
```

Table 9-53: Transitive Closure of Roads

from to	from to	from to	from to	from to
ANC FAI	IAH DEN	LAX MCI	MIA ORD	SEA ATL
ATL DEN	IAH JFK	LAX MIA	MIA SEA	SEA DEN
ATL IAH	IAH LAX	LAX MSP	MIA SFO	SEA IAH
ATL JFK	IAH MCI	LAX ORD	MSP ATL	SEA JFK
ATL LAX	IAH MIA	LAX SEA	MSP DEN	SEA LAX
ATL MCI	IAH MSP	LAX SFO	MSP IAH	SEA MCI
ATL MIA	IAH ORD	MCI ATL	MSP JFK	SEA MIA
ATL MSP	IAH SEA	MCI DEN	MSP LAX	SEA MSP
ATL ORD	IAH SFO	MCI IAH	MSP MCI	SEA ORD
ATL SEA	JFK ATL	MCI JFK	MSP MIA	SEA SFO
ATL SFO	JFK DEN	MCI LAX	MSP ORD	SFO ATL
DEN ATL	JFK IAH	MCI MIA	MSP SEA	SFO DEN
DEN IAH	JFK LAX	MCI MSP	MSP SFO	SFO IAH
DEN JFK	JFK MCI	MCI ORD	ORD ATL	SFO JFK
DEN LAX	JFK MIA	MCI SEA	ORD DEN	SFO LAX
DEN MCI	JFK MSP	MCI SFO	ORD IAH	SFO MCI
DEN MIA	JFK ORD	MIA ATL	ORD JFK	SFO MIA
DEN MSP	JFK SEA	MIA DEN	ORD LAX	SFO MSP
DEN ORD	JFK SFO	MIA IAH	ORD MCI	SFO ORD
DEN SEA	LAX ATL	MIA JFK	ORD MIA	SFO SEA
DEN SFO	LAX DEN	MIA LAX	ORD MSP	
FAI ANC	LAX IAH	MIA MCI	ORD SEA	
IAH ATL	LAX JFK	MIA MSP	ORD SFO	

The Roads2 CTE creates the digraph out of Roads. The RoadPaths CTE returns all possible source/target pairs (which has a big performance penalty), and it avoids returning and pursuing a path for which a cycle is detected. The outer query returns all distinct source/target pairs.

Here as well you can use loops instead of a recursive CTE to optimize the solution, as demonstrated earlier with the BOM scenario in Listing 9-35. Run the code in Listing 9-39 to create the *fn_RoadsTC* UDF, which returns the transitive closure of Roads using loops.

Listing 9-39: Creation script for the *fn_RoadsTC* UDF

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

CREATE FUNCTION dbo.fn_RoadsTC() RETURNS @RoadsTC TABLE (
   from_city VARCHAR(3) NOT NULL,
```

```
   to_city   VARCHAR(3) NOT NULL,
   PRIMARY KEY (from_city, to_city)
)
AS

BEGIN
  DECLARE @added as INT;

  INSERT INTO @RoadsTC(from_city, to_city)
    SELECT city1, city2 FROM dbo.Roads;

  SET @added = @@rowcount;

  INSERT INTO @RoadsTC
    SELECT city2, city1 FROM dbo.Roads

  SET @added = @added + @@rowcount;

  WHILE @added > 0 BEGIN

    INSERT INTO @RoadsTC
      SELECT DISTINCT TC.from_city, R.city2
      FROM @RoadsTC AS TC
        JOIN dbo.Roads AS R
          ON R.city1 = TC.to_city
      WHERE NOT EXISTS
        (SELECT * FROM @RoadsTC AS TC2
         WHERE TC2.from_city = TC.from_city
           AND TC2.to_city = R.city2)
        AND TC.from_city <> R.city2;

    SET @added = @@rowcount;

    INSERT INTO @RoadsTC
      SELECT DISTINCT TC.from_city, R.city1
      FROM @RoadsTC AS TC
        JOIN dbo.Roads AS R
          ON R.city2 = TC.to_city
      WHERE NOT EXISTS
        (SELECT * FROM @RoadsTC AS TC2
         WHERE TC2.from_city = TC.from_city
           AND TC2.to_city = R.city1)
        AND TC.from_city <> R.city1;

    SET @added = @added + @@rowcount;
  END
  RETURN;
END
GO

-- Use the fn_RoadsTC UDF
SELECT * FROM dbo.fn_RoadsTC();
GO
```

Run the following query to get the transitive closure of Roads shown in Table 9-53:

```
SELECT * FROM dbo.fn_RoadsTC();
```

To return all paths and distances, use similar logic to the one used in the digraph solution in the previous section. The difference here is that the distance is not just a level counter; it is the sum of the distances along the route from one city to the other.

Run the code in Listing 9-40 to return all paths and distances in Roads.

Listing 9-40: All paths and distances in Roads

```
WITH Roads2
AS
```

```
(
  SELECT city1 AS from_city, city2 AS to_city, distance FROM dbo.Roads
  UNION ALL
  SELECT city2, city1, distance FROM dbo.Roads
),
RoadPaths AS
(
  SELECT from_city, to_city, distance,
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  SELECT F.from_city, T.to_city, F.distance + T.distance,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
  FROM RoadPaths AS F
    JOIN Roads2 AS T
      ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
              THEN 1 ELSE 0 END = 0
      AND F.to_city = T.from_city
)
-- Return all paths and distances
SELECT * FROM RoadPaths;
```

Finally, to return shortest paths in Roads, use the same logic as the digraph shortest paths solution. Run the code in Listing 9-41 to return shortest paths in Roads as shown in Table 9-54.

Listing 9-41: Shortest paths in Roads

```
WITH Roads2
AS
(
  SELECT city1 AS from_city, city2 AS to_city, distance FROM dbo.Roads
  UNION ALL
  SELECT city2, city1, distance FROM dbo.Roads
),
RoadPaths AS
(
  SELECT from_city, to_city, distance,
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  SELECT F.from_city, T.to_city, F.distance + T.distance,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
  FROM RoadPaths AS F
    JOIN Roads2 AS T
      ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
              THEN 1 ELSE 0 END = 0
      AND F.to_city = T.from_city
),
RoadsMinDist -- Min distance for each pair in TC
AS
(
  SELECT from_city, to_city, MIN(distance) AS mindist
  FROM RoadPaths
  GROUP BY from_city, to_city
)
-- Return shortest paths and distances
SELECT RP.*
FROM RoadsMinDist AS RMD
  JOIN RoadPaths AS RP
    ON RMD.from_city = RP.from_city
    AND RMD.to_city = RP.to_city
    AND RMD.mindist = RP.distance;
```

Table 9-54: Shortest Paths in Roads

from_city	to_city	distance	path
ANC	FAI	359	.ANC.FAI.
ATL	IAH	800	.ATL.IAH.
ATL	JFK	865	.ATL.JFK.
ATL	MCI	805	.ATL.MCI.
ATL	MIA	665	.ATL.MIA.
ATL	ORD	715	.ATL.ORD.
DEN	IAH	1120	.DEN.IAH.
DEN	LAX	1025	.DEN.LAX.
DEN	MCI	600	.DEN.MCI.
DEN	MSP	915	.DEN.MSP.
DEN	SEA	1335	.DEN.SEA.
DEN	SFO	1270	.DEN.SFO.
IAH	LAX	1550	.IAH.LAX.
IAH	MCI	795	.IAH.MCI.
IAH	MIA	1190	.IAH.MIA.
JFK	ORD	795	.JFK.ORD.
LAX	SFO	385	.LAX.SFO.
MCI	MSP	440	.MCI.MSP.
MCI	ORD	525	.MCI.ORD.
MSP	ORD	410	.MSP.ORD.
MSP	SEA	2015	.MSP.SEA.
SEA	SFO	815	.SEA.SFO.
FAI	ANC	359	.FAI.ANC.
IAH	ATL	800	.IAH.ATL.
JFK	ATL	865	.JFK.ATL.
MCI	ATL	805	.MCI.ATL.
MIA	ATL	665	.MIA.ATL.
ORD	ATL	715	.ORD.ATL.
IAH	DEN	1120	.IAH.DEN.
LAX	DEN	1025	.LAX.DEN.
MCI	DEN	600	.MCI.DEN.
MSP	DEN	915	.MSP.DEN.
SEA	DEN	1335	.SEA.DEN.
SFO	DEN	1270	.SFO.DEN.
LAX	IAH	1550	.LAX.IAH.
MCI	IAH	795	.MCI.IAH.
MIA	IAH	1190	.MIA.IAH.
ORD	JFK	795	.ORD.JFK.
SFO	LAX	385	.SFO.LAX.
MSP	MCI	440	.MSP.MCI.
ORD	MCI	525	.ORD.MCI.
ORD	MSP	410	.ORD.MSP.

SEA	MSP	2015	.SEA.MSP.
SFO	SEA	815	.SFO.SEA.
SEA	ORD	2425	.SEA.MSP.ORD.
SEA	JFK	3220	.SEA.MSP.ORD.JFK.
ORD	SEA	2425	.ORD.MSP.SEA.
ORD	DEN	1125	.ORD.MCI.DEN.
ORD	IAH	1320	.ORD.MCI.IAH.
ORD	LAX	2150	.ORD.MCI.DEN.LAX.
ORD	SFO	2395	.ORD.MCI.DEN.SFO.
MSP	IAH	1235	.MSP.MCI.IAH.
SFO	IAH	1935	.SFO.LAX.IAH.
SFO	MIA	3125	.SFO.LAX.IAH.MIA.
MIA	LAX	2740	.MIA.IAH.LAX.
MIA	SFO	3125	.MIA.IAH.LAX.SFO.
LAX	MIA	2740	.LAX.IAH.MIA.
LAX	ATL	2350	.LAX.IAH.ATL.
SFO	MCI	1870	.SFO.DEN.MCI.
SFO	MSP	2185	.SFO.DEN.MSP.
SFO	ORD	2395	.SFO.DEN.MCI.ORD.
SFO	ATL	2675	.SFO.DEN.MCI.ATL.
SFO	JFK	3190	.SFO.DEN.MCI.ORD.JFK.
SEA	IAH	2455	.SEA.DEN.IAH.
SEA	MCI	1935	.SEA.DEN.MCI.
SEA	ATL	2740	.SEA.DEN.MCI.ATL.
SEA	MIA	3405	.SEA.DEN.MCI.ATL.MIA.
MSP	LAX	1940	.MSP.DEN.LAX.
MSP	SFO	2185	.MSP.DEN.SFO.
MCI	LAX	1625	.MCI.DEN.LAX.
MCI	SEA	1935	.MCI.DEN.SEA.
MCI	SFO	1870	.MCI.DEN.SFO.
LAX	MCI	1625	.LAX.DEN.MCI.
LAX	MSP	1940	.LAX.DEN.MSP.
LAX	ORD	2150	.LAX.DEN.MCI.ORD.
LAX	JFK	2945	.LAX.DEN.MCI.ORD.JFK.
IAH	SEA	2455	.IAH.DEN.SEA.
ORD	MIA	1380	.ORD.ATL.MIA.
MIA	JFK	1530	.MIA.ATL.JFK.
MIA	MCI	1470	.MIA.ATL.MCI.
MIA	ORD	1380	.MIA.ATL.ORD.
MIA	MSP	1790	.MIA.ATL.ORD.MSP.
MIA	DEN	2070	.MIA.ATL.MCI.DEN.
MIA	SEA	3405	.MIA.ATL.MCI.DEN.SEA.
MCI	MIA	1470	.MCI.ATL.MIA.
JFK	IAH	1665	.JFK.ATL.IAH.
JFK	MIA	1530	.JFK.ATL.MIA.

IAH	JFK	1665	.IAH.ATL.JFK.
SEA	LAX	1200	.SEA.SFO.LAX.
MSP	ATL	1125	.MSP.ORD.ATL.
MSP	JFK	1205	.MSP.ORD.JFK.
MSP	MIA	1790	.MSP.ORD.ATL.MIA.
MCI	JFK	1320	.MCI.ORD.JFK.
LAX	SEA	1200	.LAX.SFO.SEA.
JFK	MCI	1320	.JFK.ORD.MCI.
JFK	MSP	1205	.JFK.ORD.MSP.
JFK	SEA	3220	.JFK.ORD.MSP.SEA.
JFK	DEN	1920	.JFK.ORD.MCI.DEN.
JFK	LAX	2945	.JFK.ORD.MCI.DEN.LAX.
JFK	SFO	3190	.JFK.ORD.MCI.DEN.SFO.
IAH	MSP	1235	.IAH.MCI.MSP.
IAH	ORD	1320	.IAH.MCI.ORD.
IAH	SFO	1935	.IAH.LAX.SFO.
DEN	ORD	1125	.DEN.MCI.ORD.
DEN	ATL	1405	.DEN.MCI.ATL.
DEN	MIA	2070	.DEN.MCI.ATL.MIA.
DEN	JFK	1920	.DEN.MCI.ORD.JFK.
ATL	MSP	1125	.ATL.ORD.MSP.
ATL	DEN	1405	.ATL.MCI.DEN.
ATL	SEA	2740	.ATL.MCI.DEN.SEA.
ATL	SFO	2675	.ATL.MCI.DEN.SFO.
ATL	LAX	2350	.ATL.IAH.LAX.

To satisfy multiple requests for the shortest paths between two cities, you might want to materialize the result set in a table and index it as shown in Listing 9-42:

Listing 9-42: Load shortest road paths into a table

```
WITH Roads2
AS
(
  SELECT city1 AS from_city, city2 AS to_city, distance FROM dbo.Roads
  UNION ALL
  SELECT city2, city1, distance FROM dbo.Roads
),
RoadPaths AS
(
  SELECT from_city, to_city, distance,
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  SELECT F.from_city, T.to_city, F.distance + T.distance,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))

  FROM RoadPaths AS F
    JOIN Roads2 AS T
    ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
            THEN 1 ELSE 0 END = 0
    AND F.to_city = T.from_city
```

```
),
RoadsMinDist
AS
(
  SELECT from_city, to_city, MIN(distance) AS mindist
  FROM RoadPaths
  GROUP BY from_city, to_city
)
SELECT RP.*
INTO dbo.RoadPaths
FROM RoadsMinDist AS RMD
  JOIN RoadPaths AS RP
    ON RMD.from_city = RP.from_city
    AND RMD.to_city = RP.to_city
    AND RMD.mindist = RP.distance;

CREATE UNIQUE CLUSTERED INDEX idx_uc_from_city_to_city
  ON dbo.RoadPaths(from_city, to_city);
```

Once the result set is materialized and indexed, a request for the shortest path between two cities can be satisfied instantly. This is practical and advisable when information changes infrequently. As is often the case, there is a tradeoff between "up to date" and "fast." The following query requests the shortest path between Los Angeles and New York, producing the output shown in Table 9-55:

```
SELECT * FROM dbo.RoadPaths
WHERE from_city = 'LAX' AND to_city = 'JFK';
```

Table 9-55: Shortest Path between LA and NY

from_city	to_city	distance	path
LAX	JFK	2945	.LAX.DEN.MCI.ORD.JFK.

A more efficient solution to the shortest paths problem uses loops instead of recursive CTEs. It is more efficient for similar reasons to the ones described earlier; that is, in each iteration of the loop you have access to all previously spooled data and not just to the immediate previous level. You create a function called *fn_RoadsTC* that returns a table variable called @RoadsTC. The table variable has the attributes *from_city*, *to_city*, *distance* and *route*, which are self-explanatory. The function's code first inserts into @RoadsTC a row for each (*city1, city2*) and (*city2, city1*) pair from the table Roads. The code then enters a loop that iterates as long as the previous iteration inserted rows to @RoadsTC. In each iteration of the loop, the code inserts new routes that extend the existing routes in @RoadsTC. New routes are added only if the source and destination do not appear already in @RoadsTC with the same or shorter distance. Run the code in Listing 9-43 to create the *fn_RoadsTC* function.

Listing 9-43: Creation script for the *fn_RoadsTC* UDF

```
IF OBJECT_ID('dbo.fn_RoadsTC') IS NOT NULL
  DROP FUNCTION dbo.fn_RoadsTC;
GO
CREATE FUNCTION dbo.fn_RoadsTC() RETURNS @RoadsTC TABLE
(
  uniquifier INT          NOT NULL IDENTITY,
  from_city  VARCHAR(3)   NOT NULL,
  to_city    VARCHAR(3)   NOT NULL,
  distance   INT          NOT NULL,
  route      VARCHAR(MAX) NOT NULL,
  PRIMARY KEY (from_city, to_city, uniquifier)
)
AS
BEGIN
  DECLARE @added AS INT;

  INSERT INTO @RoadsTC
    SELECT city1 AS from_city, city2 AS to_city, distance,
      '.' + city1 + '.' + city2 + '.'
    FROM dbo.Roads;
```

```
  SET @added = @@rowcount;

  INSERT INTO @RoadsTC
    SELECT city2, city1, distance, '.' + city2 + '.' + city1 + '.'
    FROM dbo.Roads;

  SET @added = @added + @@rowcount;

  WHILE @added > 0 BEGIN
    INSERT INTO @RoadsTC
    SELECT DISTINCT TC.from_city, R.city2,
      TC.distance + R.distance, TC.route + city2 + '.'
    FROM @RoadsTC AS TC
      JOIN dbo.Roads AS R
        ON R.city1 = TC.to_city
    WHERE NOT EXISTS
      (SELECT * FROM @RoadsTC AS TC2
      WHERE TC2.from_city = TC.from_city
        AND TC2.to_city = R.city2
        AND TC2.distance <= TC.distance + R.distance)
    AND TC.from_city <> R.city2;

  SET @added = @@rowcount;

  INSERT INTO @RoadsTC
    SELECT DISTINCT TC.from_city, R.city1,
      TC.distance + R.distance, TC.route + city1 + '.'
    FROM @RoadsTC AS TC
      JOIN dbo.Roads AS R
        ON R.city2 = TC.to_city
    WHERE NOT EXISTS
      (SELECT * FROM @RoadsTC AS TC2
      WHERE TC2.from_city = TC.from_city

        AND TC2.to_city = R.city1
        AND TC2.distance <= TC.distance + R.distance)
      AND TC.from_city <> R.city1;

    SET @added = @added + @@rowcount;
  END
  RETURN;
END
GO
```

The function might return more than one row for the same source and target cities. To return shortest paths and distances, use the following query:

```
SELECT from_city, to_city, distance, route
FROM (SELECT from_city, to_city, distance, route,
       RANK() OVER (PARTITION BY from_city, to_city
                    ORDER BY distance) AS rk
       FROM dbo.fn_RoadsTC()) AS RTC
WHERE rk = 1;
```

The derived table query assigns a rank value (*rk*) to each row, based on *from_city, to_city* partitioning, and *distance* ordering. This means that shortest paths will be assigned with the rank value 1. The outer query filters only shortest paths (*rk = 1*).

Once you're done querying the RoadPaths table, don't forget to drop it:

```
DROP TABLE dbo.RoadPaths;
```

Conclusion

This chapter covered the treatment of graphs, trees, and hierarchies. I presented iterative/recursive solutions for graphs, and also solutions where you materialize information describing a tree. The main advantage of the iterative/recursive solutions is that you don't need to materialize and maintain any additional attributes; rather, the graph manipulation is based on the stored edge attributes. The materialized path solution materializes an enumerated path, and possibly also the level for each node in the tree. The maintenance of the additional information is not very expensive, and you benefit from simple and fast set-based queries. The nested sets solution materializes left and right values representing set containment relationships, and possibly the level in the tree. This is the most elegant solution of the ones I presented, and also it allows simple and fast queries. However, maintaining the materialized information is very expensive, so typically this solution is practical for either static trees or small dynamic trees.

In the last section, I presented solutions to transitive closure and shortest path problems.

Because this chapter concludes the book, I feel I should also add some closing words.

If you ask me what's the most important thing I hope you carry from this book, I'd say giving special attention to fundamentals. Do not underestimate or take them lightly. Spend time on identifying, focusing on, and perfecting fundamental key techniques. When faced with a tough problem, solutions will flow naturally.

"Matters of great concern should be treated lightly."

"Matters of small concern should be treated seriously."

— *Hagakure, The Book of the Samurai* by Yamamoto Tsunetomo

The meaning of these sayings is not what appears on the surface. The book goes on to explain:

> "Among one's affairs there should not be more than two or three matters of what one could call great concern. If these are deliberated upon during ordinary times, they can be understood. Thinking about things previously and then handling them lightly when the time comes is what this is all about. To face an event and solve it lightly is difficult if you are not resolved beforehand, and there will always be uncertainty in hitting your mark. However, if the foundation is laid previously, you can think of the saying, 'Matters of great concern should be treated lightly,' as your own basis for action."