CHAPTER 31 SQL Server Graph Databases

In This Chapter

- Graph Databases: A General Introduction
- Creating Node Tables and Edge Tables
- Querying Graph Data
- Modifying and Editing Data in Graph Databases
- Querying Graph Data Using Relational Queries

Microsoft introduced graph capabilities in SQL Server 2017. This new component is very important because most social networks are based on graphs and Transact-SQL statements do not support operations on graphs optimally.

The chapter discusses all extensions in SQL Server to support Graph Databases. The first section provides a general introduction to graph databases and then lists all properties of this component in SQL Server. The second section explains in detail the creation of node tables and edge tables. The third section discusses how graphs stored in node tables and edge tables can be queried and provides several examples to show this. The fourth section explains how to modify data in node tables and edge tables. The final section explains how graph data can be queried using relational queries.

Graph Databases: A General Introduction

Generally, a *graph* is a collection of nodes and edges, or, in less formal language, a set of entities and the relationships that connect them. Therefore, *nodes* of a graph are entities, such as Employee, Product, and Customer, and *edges* are relationships. How the nodes relate to the world is specified by edges. For instance, employees sell products, and products are bought by customers. Using graphs, you can model a wide variety of different scenarios from the real world, anything from the design of social networks to the construction of space shuttles.

Graph Databases: Models

A graph database is a database management system that supports the usual properties of DBMSs (see Chapter 1) using a graph data model as its underlying model. Graph databases are generally built for use with online transaction processing (OLTP) systems. Two main properties of graph database technology are how the data is stored and how the data is processed.

The underlying storage of a graph database can be native or nonnative. Native storage is optimized and designed for storing and managing graphs. A graph database is classified as nonnative when the storage comes from a "non-graph" database model, such as a relational, columnar, or object-oriented model. (The nonnative approach can lead to performance problems, because the storage engine of non-graph database models is not optimized for graphs.)

The processing engine of a graph database doesn't necessarily need to support native graph processing. The key element of graph technology in relation to data processing is whether or not index-free adjacency is supported. (*Index-free adjacency* means that every node contains a direct pointer to its adjacent node. That way, creation of indices is not necessary, at least for queries that reference a node and its adjacent ones.)

Property Graph Model

The most popular model in relation to graphs is called the *property graph model*. Besides this model, there are two others: Resource Description Framework (RDF) and hypergraphs. The discussion of these two models is outside the scope of this book.

A property graph has the following characteristic: it contains nodes and edges. Each node and each edge has a unique identifier. Also, each node has a collection of properties (key-value pairs). Finally, a node has a set of outgoing and incoming edges.

Edges are named and always have a start and an end node. Each edge has a label that denotes its connection type between two nodes. Edges, like nodes, can also contain properties.

Advantages of Graph Databases

The power of graph databases becomes apparent in relation to two issues:

- Performance
- Flexibility

It is well known that SQL does not deal very well with connected data. The reason is that the number of join operations (inner joins and self-referencing joins) increases significantly as the dataset gets bigger and the traversal of a graph goes deeper and deeper. (This is discussed further in the section "Querying Graph Data Using Relational Queries" at the end of this chapter.) In contrast to relational database systems, the performance of a graph database system remains constant, even when the dataset significantly grows. This is because queries are localized to a particular subgraph of the entire graph. As a result, the execution time for each query is proportional only to the size of the subgraph traversed to satisfy that query.

One of the main advantages of graphs is that they are *additive*, meaning that you can add new substructures (nodes, relationships, and subgraphs) to an existing graph without disturbing existing user applications in their functioning. This flexibility has positive implications for developers, because they can significantly increase their productivity. Also, because of the graph model's flexibility, you don't have to model your domain in depth, making it significantly easier to change business requirements later, if necessary.

SQL Server Graph Databases: An Introduction

As previously introduced, a graph is a collection of nodes and relationships (edges) between nodes. A graph in SQL Server Graph Databases is a collection of node tables and edge tables, where nodes are stored in node tables and relationships between nodes are stored in edge tables. Using these two forms of tables, SQL Server Graph Databases supports several general features of graph databases:

- The nodes and edges of a graph are first-class citizens and can have attributes or properties associated with them. (An object is considered a "first-class citizen" of a system if it can be stored in variables and data structures and can be passed as a parameter to a subroutine.)
- Pattern matching can be expressed easily. (As you will see shortly, pattern matching in SQL Server Graph Databases is expressed using the MATCH function.)
- Edges can be specified as directed or undirected. (Directed edges are specified using edge constraints.)
- Polymorphic queries can be expressed easily. (Polymorphic queries in relation to graphs are queries that return instances of a particular node as well as instances of all subgraphs of that node.) Note that node instances are logically the same as instances of entities, as explained in Chapter 1.

Creating Node Tables and Edge Tables

As you will see in the following examples, the creation of node tables and edge tables is straightforward. In other words, if you know the syntax of the CREATE TABLE statement for regular relational tables, it will be very easy for you to create node tables and edge tables. All examples that follow use the model of a graph shown in Figure 31-1.

The graph in Figure 31-1 contains three nodes: **Employee**, **Company**, and **City**. The **Employee** entity has four attributes (properties): **ID**, **Name**, **Age**, and **Sex**. The **Company** entity has four attributes: **ID**, **Name**, **Sector**, and **City**, while **City** has three attributes: **ID**, **Cityname**, and **Statename**.

The graph contains three edges: **WorksIn**, **LocatedIn**, and **LivesIn**. **WorksIn** has a property called **Starts** that specifies the year in which the particular employee started to work for the specified company. Similarly, the **LivesIn** edge has the **Since** property that specifies the date on which an employee moved to a particular city. (The third edge does not have any additional properties.) All three relationships are nonrecursive, meaning that they all connect two *different* nodes.

Creating Node Tables

A node table can be created in any user-defined database. The creation of such a table is very similar to creating a regular relational table, with one extension. Example 31.1 shows the creation of three node tables, corresponding to the entities shown in Figure 31-1.

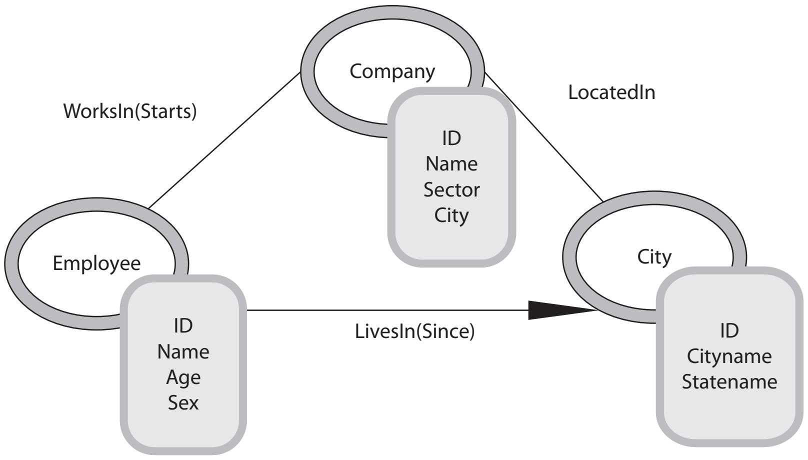

Figure 31-1 Graph used to demonstrate examples in this chapter

Example 31.1

```
CREATE DATABASE graph_db;
GO;
USE graph_db;
 CREATE TABLE  dbo.Company (
       ID   INT  NOT NULL PRIMARY KEY,
       name   VARCHAR (100) NULL,
       sector VARCHAR(25) NULL,
       city   VARCHAR (100) NULL) AS NODE;
 CREATE TABLE  dbo.Employee (
       ID   INT  NOT NULL PRIMARY KEY,
       name   VARCHAR (100) NULL,
       age    INT NULL,
       sex   char (10) NULL) AS NODE;
 CREATE TABLE  dbo.City (
       ID   INT  NOT NULL PRIMARY KEY,
       name   VARCHAR(100) NULL,
       stateName   VARCHAR(100) NULL) AS NODE;
```

The most important extension concerning node tables is the AS NODE clause, written at the end of the CREATE TABLE statement. This clause defines the corresponding table as a node table. When you specify this clause, the system adds two new columns of the BIT data type to the **sys.tables** catalog view: **is_node** and **is_edge**. For a node table, the value of **is_node** is set to 1, and the value of **is_edge** is set to 0. (A detailed description of metadata concerning graph databases is provided in the section "Editing Information Concerning Graph Databases" at the end of this chapter.)

Whenever you create a node table, along with the user-defined columns, an implicit **$node_id** column is created, which uniquely identifies each instance of the corresponding node table. The values in the **$node_id** column are automatically generated and are a combination of the **object_id** value of that node table and an internally generated value of the BIGINT data type. (When you display the values of the **$node_id** column, the corresponding computed values are displayed as JSON strings.) Also, **$node_id** is a pseudo column that maps to an internal name with a hex string appended to it. In other words, when you select **$node_id** from the table, the column name appears as **$node_id_\hex_string**.

After creation of node tables, you have to load data into them. As Example 31.2 shows, inserting rows into node tables works the same way as for any other regular table.

Example 31.2

```
USE graph_db;
INSERT INTO Employee (ID,Name,Sex)
      VALUES (1,'Matthew Smith','Male');
INSERT INTO Employee (ID,Name,Sex)
      VALUES (2,'Ann Jones','Female');
INSERT INTO Employee (ID,Name,Sex)
      VALUES (3,'John Barrimore','Male');
INSERT INTO Employee (ID,Name,Sex)
      VALUES (4,'James James','Male');
INSERT INTO Employee (ID,Name,Sex)
      VALUES (5,'Elsa Bertoni','Female');
INSERT INTO Employee (ID,Name,Sex)
      VALUES (6,'Elke Hansel','Female');

INSERT INTO Company VALUES (1,'Comp_A','Pharma','Kansas City');
INSERT INTO Company VALUES (2,'Comp_B','Manufacturing','Hoboken');
INSERT INTO Company VALUES (3,'Comp_C','Pharma','Indianopolis');
INSERT INTO Company VALUES (4,'Comp_D','IT','Lexington');
INSERT INTO Company VALUES (5,'Comp_E','IT','Madison');

INSERT INTO City VALUES (1,'Kansas City','Kansas');
INSERT INTO City VALUES (2,'Hoboken','New Jersey');
INSERT INTO City VALUES (3,'Indianopolis','Indiana');
INSERT INTO City VALUES (4,'Lexington','Kentucky');
INSERT INTO City VALUES (5,'Minneapolis','Wisconsin');
INSERT INTO City VALUES (6,'Madison','Wisconsin');
```

Creating Edge Tables

An edge table represents a relationship between two graph nodes. Therefore, each row of an edge table contains instances of the corresponding relationship. An edge table has, among others, three hidden columns: **$edge_id**, **$from_id**, and **$to_id**. The values of the **$edge_id** column specify the unique IDs and are stored as JSON documents. (For the description of

JSON, see Chapter 29.) The other two columns represent the references between the instances of the relationship—that is, the rows of the edge table to the instances of both connected node tables. The **$from_id** column stores the **$node_id** values of the nodes from which the edges originate, and the **$to_id** column stores the **$node_id** values of the nodes at which the edges terminate.

NOTE A *hidden column* is a column that exists in the table but cannot be selected. Besides the three hidden columns specified, there are several other hidden columns in relation to SQL Server Graph Databases. One of these columns is **graph_id**, which is used internally by the SQL Server system to manage graph data in the proper way (and is described later in the chapter, in the section "Graph Databases: System Functions").

Similar to node tables, for each edge table, the system adds two new columns to the **sys.tables** catalog view: **is_node** and **is_edge**. As expected, in contrast to node tables, the values of the **is_node** and **is_edge** columns are 0 and 1, respectively.

NOTE Generally, graphs can be undirected or directed. In an undirected graph, all the edges are bidirectional. In a directed graph, all the edges point in one direction. Single edges can be thought of in the same way: an undirected edge is bidirectional, while a directed edge points in a specified direction. Referring to Figure 31-1 earlier in the chapter, the edge **LivesIn** is directed, while the **WorksIn** and **LocatedIn** edges are undirected.

Example 31.3 shows the creation of the three edge tables: **WorksIn**, **LocatedIn**, and **LivesIn**.

Example 31.3

```
USE graph_db;
CREATE TABLE WorksIn (starts INT) AS EDGE;
CREATE TABLE LocatedIn AS EDGE;
CREATE TABLE LivesIn (Since DATE NULL
   CONSTRAINT Emp_to_City CONNECTION (Employee TO City)) AS EDGE;
```

The creation of the first two edge tables in Example 31.3 is straightforward. You just create a table in the regular way and append the AS EDGE clause. The creation of the **LivesIn** edge is different. The reason is that the first two edges (**WorksIn** and **LocatedIn**) are undirected edges, meaning that you can traverse them in both directions. For instance, in the case of the **LocatedIn** edge, you can traverse from the **Company** node to the **City** node and vice versa. In the case of the **LivesIn** edge, you can traverse only from the **Employee** node to the **City** node, because this edge is directed.

The creation of directed edges is possible in SQL Server 2019 by using *edge constraints*. With this feature, you can apply restrictions during the creation of an edge by using the CONNECTION clause. This clause is used in Example 31.3 to specify that you can traverse the **LivesIn** edge only from the **Employee** node to the **City** node. (This is explained further in the discussion after Example 31.6 in the next section.)

After creating graph objects, you can examine them using Object Explorer. You will see that there is a new subfolder called Graph in the Tables folder. All graph objects will be inside this subfolder. Note that names of auto-generated fields include a GUID, but you can reference these fields with their short names. (A short name is a pseudo column and you can use it in queries.)

Inserting Data into Edge Tables

In contrast to the creation of edge tables, which looks like the creation of regular relational tables (extended with the AS EDGE clause), loading data into edge tables is different than loading data into relational tables. Remember that an edge table represents a relationship between two nodes in a graph. For this reason, each INSERT statement, which loads an instance of such a relationship, must specify both the instance of the node where the relationship originates and the instance of the node where the relationship terminates. To do this, you use the already mentioned **$node_id** values from the **$from_id** and **$to_id** columns. (As you will see in Example 31.4, we use subqueries to solve this problem.)

Example 31.4 shows the insertion of rows into the **WorksIn** edge table.

Example 31.4

```
--To insert data into an edge table we need to provide the reference for
-- the $from_id and $to_id as a reference point to both nodes
USE graph_db;
INSERT INTO WorksIn VALUES ((SELECT $node_id FROM Employee WHERE id = 1),
      (SELECT $node_id FROM Company WHERE id = 1), 2015);
INSERT INTO WorksIn VALUES ((SELECT $node_id FROM Employee WHERE id = 2),
      (SELECT $node_id FROM Company WHERE id = 2), 2018);
INSERT INTO WorksIn VALUES ((SELECT $node_id FROM Employee WHERE id = 3),
      (SELECT $node_id FROM Company WHERE id = 3), 2015);
INSERT INTO WorksIn VALUES ((SELECT $node_id FROM Employee WHERE id = 4),
      (SELECT $node_id FROM Company WHERE id = 3), 2016);
INSERT INTO WorksIn VALUES ((SELECT $node_id FROM Employee WHERE id = 5),
      (SELECT $node_id FROM Company WHERE id = 3), 2017);
INSERT INTO WorksIn VALUES ((SELECT $node_id FROM Employee WHERE id = 6),
      (SELECT $node_id FROM Company WHERE id = 4), 2018);
```

The first INSERT statement in Example 31.4 defines a relationship in the **WorksIn** edge table between the instance of the **Employee** node (entity) with the ID value 1 and the instance of the **Company** node (entity) with the ID value 1. To add data to the **$from_id** column, you must specify the **$node_id** value associated with the **Employee** entity. One way to get this value is to include a subquery that targets the entity, using its primary key value. You can take the same approach for the **$to_id** column in relation to the **Company** entity.

NOTE Insertion of the data in this way demonstrates why it is useful to add primary keys to each node table (see Example 31.1), but not necessary for the edge tables. The primary keys on the node tables make it much easier to provide the **$node_id** value to the INSERT statements in Example 31.4.

Examples 31.5 and 31.6 show the insertion of rows in the other two edge tables.

Example 31.5

```
USE graph_db;
INSERT INTO LocatedIn VALUES ((SELECT $node_id FROM Company WHERE id = 1),
      (SELECT $node_id FROM City WHERE id=2))
INSERT INTO LocatedIn VALUES ((SELECT $node_id FROM Company WHERE id = 2),
      (SELECT $node_id FROM City WHERE id=1));
INSERT INTO LocatedIn VALUES ((SELECT $node_id FROM Company WHERE id = 3),
      (SELECT $node_id FROM City WHERE id=3));
INSERT INTO LocatedIn VALUES ((SELECT $node_id FROM Company WHERE id = 4),
      (SELECT $node_id FROM City WHERE id=2));
```

Example 31.6

```
USE graph_db;
INSERT INTO LivesIn VALUES ((SELECT $node_id FROM Employee WHERE id = 1),
      (SELECT $node_id FROM City WHERE id=6), '1.1.2018');
INSERT INTO LivesIn VALUES ((SELECT $node_id FROM Employee WHERE id = 2),
      (SELECT $node_id FROM City WHERE id=5), '2.1.2018');
INSERT INTO LivesIn VALUES ((select $node_id FROM Employee WHERE id = 3),
      (SELECT $node_id FROM City WHERE id=4), '3.1.2018');
INSERT INTO LivesIn VALUES ((SELECT $node_id FROM Employee WHERE id = 4),
      (SELECT $node_id FROM City WHERE id=2), '4.1.2018');
INSERT INTO LivesIn VALUES ((SELECT $node_id FROM Employee WHERE id = 5),
      (SELECT $node_id FROM City WHERE id=3), '5.1.2018')
INSERT INTO LivesIn VALUES ((SELECT $node_id FROM Employee WHERE id = 6),
      (SELECT $node_id FROM City WHERE id=1), '6.1.2018');
```

All the INSERT statements in Example 31.6 succeed because they all insert edge instances that connect *employees* to *cities*. However, if you try to insert an edge instance the other way around (from the **City** node to the **Employee** node), like the following example:

```
INSERT INTO LivesIn VALUES ((SELECT $node_id FROM City WHERE id = 1),
      (SELECT $node_id FROM Employee WHERE id=6), '6.1.2018');
```

the INSERT statement fails, because traversing from the **City** node to the **Employee** node is forbidden in the specification of the **LivesIn** table (see Example 31.3), and you get the following error message:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the EDGE constraint "Emp_to_City". The
conflict occurred in database "graph_db", table "dbo.LivesIn".
```

NOTE The CONNECTION constraint is similar to all other constraints that you can specify for a regular table. For this reason, you can drop such a constraint using the DROP CONSTRAINT clause of the ALTER TABLE statement (see Example 5.26 in Chapter 5).

Some Remarks to Directed and Undirected Relationships

As you already know, there are two different forms of relationships: undirected and directed. An undirected relationship specifies that the instances of two corresponding nodes can be connected in both directions, while in the case of directed edges the connection is possible only in one direction, which is specified with the corresponding edge constraint.

The implementation of edge constraint for SQL Server Graph Databases has been very important, because that way the main requirements in relation to directed relationships have been fulfilled.

This is not true for undirected relationships. Note that edges are actually binary relationships between *two particular* nodes. If you take a look at Example 31.3, you will see that the creation of the **WorksIn** edge table does not have any reference to the corresponding tables (**Employee** and **Company**), which are part of this relationship. Therefore, you can write any syntactically correct, but semantically wrong, INSERT statement and both of them will be accepted by the system. In other words, the first INSERT statement in Example 31.5

```
INSERT INTO WorksIn  VALUES ((SELECT $node_id FROM Employee WHERE id = 1),
    (SELECT $node_id FROM Company WHERE id = 1), 2015);
```

is semantically and syntactically correct. But, if you change "Company" to "City" such as shown here,

```
INSERT INTO WorksIn  VALUES ((SELECT $node_id FROM Employee WHERE id = 1),
    (SELECT $node_id FROM City WHERE id = 1), 2015);
```

the system will insert this statement even though it is semantically wrong. This makes a burden for the programmer, who has to worry about the correct names of the node tables during insertion of rows for a particular edge table.

For instance, the code to create an edge of a graph in the Cypher language, which is an alter ego of the query language of SQL Server Graph Databases, contains explicitly the names of the nodes:

```
CREATE (p:Person)-[:LIKES]->(t:Technology)
```

Therefore, in this case, the system takes care whether the logically correct names of nodes are specified and can use these names to check the semantic meaning of the INSERT statements used during insertion of rows. (Cypher is a declarative query language implemented to efficiently query graphs. It is a query language of another graph database system called neo4J.)

We can hope that the design of edge tables will be improved in one of the next versions of SQL Server to consider this issue.

Querying Graph Data

SQL Server Graph Databases supports the MATCH function to query graph data. The syntax and the semantics of this function are similar to those of the MATCH function from the Cypher query language of another graph database system, Neo4J. Cypher is a declarative query language that is implemented to efficiently query graphs. (Note that currently only part of the original language has been implemented in SQL Server Graph Databases.)

After the introduction of the MATCH function, we will discuss recursive relationships and how they can be implemented in SQL Server Graph Databases.

The MATCH Function

The MATCH function allows you to specify a search pattern based on relationships between two nodes. This function is a part of the WHERE clause of the SELECT statement that queries node and edge tables. Example 31.7 shows the use of this function.

Example 31.7

```
-- Get the names of the companies and the names of their employees
USE graph_db;
SELECT DISTINCT Cmp.Name CName, Emp.Name EName
           FROM Employee Emp, WorksIn, Company Cmp
           WHERE MATCH(Emp-(WorksIn)->Cmp);
```

The result is

CName	EName
Comp_A	Matthew Smith
Comp_B	Ann Jones
Comp_C	Elsa Bertoni
Comp_C	James James
Comp_C	John Barrimore
Comp_D	Elke Hansel

Each MATCH function must contain a search pattern. Such patterns represent one or more relationships. For each relationship, you must specify the originating node and the terminating node, as well as the edge table that connects the two nodes together. You must also specify the direction of the relationship, using dashes and arrows, with the edge table situated between the two node tables. In Example 31.7, the term

```
(Emp-(WorksIn)->Cmp)
```

specifies that the originating node is **Employee** (represented by its alias, **Emp**), the edge table is **WorksIn**, and the terminating node is **Company** (represented by its alias, **Cmp**). Generally, if you want to specify a single relationship called **Edge** with the originating node **Node1** and terminating node **Node2**, you would use the following syntax:

```
MATCH(Node1-(Edge)->Node2)
```

NOTE As shown in the syntax, the name of the edge table in the MATCH function is enclosed in parentheses, with a dash preceding its name and a dash and right arrow following its name. This specifies a relationship that moves from left to right. You can reverse this order and specify a relationship that moves from right to left by reversing the order of the node names (that is, MATCH(Node2-(Edge)->Node1). Generally, this relationship is semantically different from the left-to-right relationship, as demonstrated in Examples 31.15 and 31.16 later in the chapter.

Examples 31.8 and 31.9 show how to narrow the result of the query in Example 31.7 by using the AND operator.

Example 31.8

```
-- Get the name of the company where Matthew Smith works
USE graph_db;
SELECT Cmp.Name CName
           FROM Employee Emp, WorksIn, Company Cmp
            WHERE MATCH(Emp-(WorksIn)->Cmp)
                  AND Emp.Name= 'Matthew Smith';
```

The result is

CName
Comp_A

Example 31.9

```
-- Get the list of the employees who live in Madison
USE graph_db;
SELECT Emp.Name EName
        FROM Employee Emp, LivesIn, City
         WHERE MATCH(Emp-(LivesIn)->City)
               AND City.Name='Madison';
```

The result is

EName
Matthew Smith

SQL Server Graph Databases allows you to use other clauses of the SELECT statement together with the MATCH function. Example 31.10 uses the ORDER BY clause to sort the result rows according to the first column in the SELECT list.

Example 31.10

```
-- Get the list of companies located in the city of Hoboken
-- Sort the companies according to their names
USE graph_db;
SELECT Cmp.Name CName
           FROM City C, LocatedIn, Company Cmp
            WHERE MATCH(Cmp-(LocatedIn)->C)
                  AND C.Name='Hoboken'   ORDER BY 1;
```

The result is

CName
Comp_A
Comp_D

SQL Server Graph Databases allows you to traverse a graph as deep as you wish. In this case you have to specify multiple relationships using the MATCH function. One way to do it is to use the AND operator, as shown in Example 31.11.

NOTE The Boolean operators, OR and NOT, cannot be used with the MATCH function.

Example 31.11

```
USE graph_db;
SELECT Employee.name EName, Company.name CName
      FROM Employee, WorksIn, Company, LocatedIn, City
       WHERE MATCH(Employee-(WorksIn)->Company
             AND Company-(LocatedIn)->City )
             AND WorksIn.Starts='2017' AND Company.name='Comp_C';
```

The result is

EName	CName
Elsa Bertoni	Comp_C

Example 31.11 links the instances of the **Employee** node with the instances of the **Company** node using the **WorksIn** relationship. After that, it links the instances of the **Company** node with the instances of the **City** node using the **LocatedIn** relationship. By being able to link together multiple relationships, you can traverse a graph as deep as you wish.

Generally, you can link together multiple relationships without using the AND operator, as long as the particular search pattern specifies the same logic. In other words, the terminating node table of the previous pattern must be the originating table of the subsequent one. Example 31.12 solves the same problem as the previous one, linking together multiple relationships without using the AND operator.

Example 31.12

```
USE graph_db;
SELECT Employee.name, Company.name
    FROM Employee, WorksIn, Company, LocatedIn, City
     WHERE MATCH(Employee-(WorksIn)->Company-(LocatedIn)->City)
          AND WorksIn.Starts='2017' and Company.name='Comp_C';
```

Recursive Relationships

All relationships that have been used up to this point in this chapter are nonrecursive relationships, because each of them connects two *different* nodes. A recursive relationship is a special form of relationships in which a node is connected with itself.

In the following examples we will use a relationship called **Is_Liked**. This recursive relationship connects the **Employee** entity with itself to find employees who like other employees of the company. Example 31.13 creates the **Is_Liked** edge table and inserts several rows into this table.

Example 31.13

```
-- Create an edge table for the recursive relationship
USE graph_db;
 CREATE TABLE dbo.Is_Liked(start_date DATE) AS EDGE;
-- Insert several rows
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 1),
   (SELECT $node_id FROM Employee WHERE ID = 2),'1.1.2017');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 1),
   (SELECT $node_id FROM Employee WHERE ID = 3),'2.1.2018');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 1),
   (SELECT $node_id FROM Employee WHERE ID = 4),'3.1.2019');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 2),
   (SELECT $node_id FROM Employee WHERE ID = 3),'4.1.2016');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 2),
   (SELECT $node_id FROM Employee WHERE ID = 5),'5.1.2017');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 2),
   (SELECT $node_id FROM Employee WHERE ID = 6),'6.1.2017');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 3),
   (SELECT $node_id FROM Employee WHERE ID = 4),'7.1.2018');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 4),
   (SELECT $node_id FROM Employee WHERE ID = 5),'8.1.2016');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 4),
   (SELECT $node_id FROM Employee WHERE ID = 6),'9.1.2017');
INSERT INTO Is_Liked VALUES
  ((SELECT $node_id FROM Employee WHERE ID = 5),
   (SELECT $node_id FROM Employee WHERE ID = 6),'10.1.2019');
```

Example 31.14 displays all employees who like other employees.

Example 31.14

```
USE graph_db;
SELECT E1.name AS SourceName, E2.name AS TargetName
```

```
    FROM Employee E1, Is_Liked, Employee E2
     WHERE MATCH(E1-(Is_Liked)->E2);
```

The result is

SourceName	TargetName
Matthew Smith	Ann Jones
Matthew Smith	John Barrimore
Matthew Smith	James James
Ann Jones	John Barrimore
Ann Jones	Elsa Bertoni
Ann Jones	Elke Hansel
John Barrimore	James James
James James	Elsa Bertoni
James James	Elke Hansel
Elsa Bertoni	Elke Hansel

The syntax of the SELECT statement in Example 31.14 is similar to the syntax of the same statement in previous examples. The only difference is that you have to use at least one alias for the **Employee** node table because that name appears twice in the query.

Example 31.15

```
-- Display all employees that like Matthew Smith
USE graph_db;
SELECT E2.name AS FriendName
     FROM Employee E1, Is_Liked, Employee E2
     WHERE MATCH(E1-(Is_Liked)->E2)
         AND E1.name = 'Matthew Smith';
```

The result is

FriendName
Ann Jones
John Barrimore
James James

The following example retrieves all employees who are liked by Matthew Smith.

Example 31.16

```
-- Display all employees who are liked by Matthew Smith
USE graph_db
SELECT E2.name AS FriendName
```

```
    FROM Employee E1, Is_Liked, Employee E2
    WHERE MATCH(E2-(Is_Liked)->E1)
        AND E1.name = 'Matthew Smith';
```

The result of Example 31.16 does not contain any rows. If you take a closer look at Examples 31.15 and 31.16, you will see that relationship is between the same nodes, one moving from left to right and the other from right to left. Semantically, the results of these two examples tell us that three employees in the company like Matthew Smith (Example 31.15), but Matthew does not like any other employee (Example 31.16).

Example 31.17 shows how you can specify "second-level" likes.

Example 31.17

```
-- Display all employees who like employees that like Ann Jones
USE graph_db;
SELECT Person3.name AS FriendName
    FROM Employee Person1, Employee Person2,
        Is_Liked, Is_Liked Is_Liked2, Employee Person3
    WHERE MATCH(Person1-(Is_Liked)->Person2-(Is_Liked2)->Person3)
        AND Person1.name = 'Ann Jones';
```

The result is

FriendName
James James
Elke Hansel

Example 31.17 also shows, generally, how you can navigate through a graph as deep as you want. Each time you reference the next sublevel of the graph, you have to add the name of the edge table, together with the name of the self-referencing node table, first in the FROM clause of the query and, after that, in the MATCH function, appending both names or their aliases to the single parameter of this function.

Modifying and Editing Data in Graph Databases

The first section of this chapter discussed how to use the INSERT statement to load data into node and edge tables. This section explains the other two modification operations, DELETE and UPDATE. Generally, deletion of rows in node and edge tables can be performed without any restrictions, while modification of values of columns is limited and depends on the type of the particular column.

Deleting Graph Data

Before we discuss how SQL Server Graph Databases deletes instances of node and edge tables, Example 31.18 provides one more look at how data is inserted into an edge table.

Example 31.18

```
USE graph_db;
INSERT into LivesIn VALUES ((SELECT $node_id FROM Employee WHERE id = 6),
      (SELECT $node_id FROM City WHERE id = 5), '2.1.2018');
```

The INSERT statement in Example 31.18 adds a new instance of the relationship **LivesIn**. This instance connects instance 6 of the **Employee** node table with instance 5 of the **City** node table. With the query in Example 31.19, you can check whether this instance is properly inserted.

Example 31.19

```
USE graph_db;
SELECT E.Name, C.Name
    FROM Employee E, LivesIn, City C
    WHERE MATCH(E -(LivesIn)->C) AND E.id = 6 AND C.id = 5;
```

The result is

Name	Name
Elke Hansel	Minneapolis

The query in Example 31.19 is based on two **$node_id** values. One approach to delete this instance is to use two nested SELECT statements, as shown in Example 31.20 (which is the same way they are used in Example 31.18). Note that the deletion is based on the values of the **$from_id** and **$to_id** columns of the **LivesIn** table and the corresponding values of **$node_id**.

Example 31.20

```
USE graph_db;
DELETE  FROM LivesIn
      WHERE $from_id =(SELECT $node_id FROM Employee WHERE id = 6)
        AND $to_id = (SELECT $node_id FROM City WHERE id = 5);
```

Another way to delete this instance of the relationship **LivesIn** is to use the MATCH function, as shown in Example 31.21. (Note that the syntax of the DELETE statement does not correspond to the standardized syntax of that statement. It is a proprietary extension in SQL Server.)

Example 31.21

```
USE graph_db;
DELETE  LivesIn
      FROM Employee E, LivesIn, City C
      WHERE MATCH(E-(LivesIn)->C) AND E.id = 6 AND C.id = 5;
```

The MATCH function in Example 31.21 searches for a pattern based on the specified relationship. In this case, MATCH selects all rows of the **LivesIn** relationship. The other two conditions in the WHERE clause of the query restrict the deletion to the particular instance of that relationship.

Updating Graph Data

In contrast to the DELETE statement, the UPDATE statement has some limitations. Generally, you can apply the UPDATE statement only to modify the values of the user-defined columns, as demonstrated in Example 31.22.

Example 31.22

```
USE graph_db;
UPDATE WorksIn  SET Starts = 2020
   WHERE  $from_id = (SELECT $node_id FROM Employee WHERE id = 6)
     AND $to_id = (SELECT $node_id FROM Company WHERE id = 1);
```

Example 31.22 modifies the value of the **Starts** column of the **WorksIn** edge table by using the values of the **$node_id** columns of the originating and terminating tables.

Editing Information Concerning SQL Server Graph Databases

To edit information concerning SQL Server Graph Databases, you can use either catalog views or system functions, both of which are described in the following subsections.

Graph Databases: Catalog Views

Two catalog views, **sys.tables** and **sys.columns**, have been extended to contain metadata concerning graph databases. Also, two new catalog views, **sys.edge_constraints** and **sys.edge_constraint_clauses**, are used to edit metadata concerning existing edge constraints.

As you already know, when you create a node table or an edge table, the system adds two new columns of the BIT data type, **is_node** and **is_edge**, to the **sys.tables** catalog view. For a node table, the value of **is_node** is set to 1, and the value of **is_edge** is set to 0. Conversely, the values of the **is_node** and **is_edge** columns of an edge table are 0 and 1, respectively.

Example 31.23 retrieves the names of all tables of the **graph_db** database that are either node tables or edge tables. (The corresponding values of the **is_node** and **is_edge** columns are displayed, too.)

Example 31.23

```
USE graph_db;
SELECT name, is_node, is_edge
FROM sys.tables
  WHERE is_node = 1 OR is_edge = 1;
```

The result is

Employee	1	0
City	1	0
Company	1	0

WorksIn	0	1
LocatedIn	0	1
LivesIn	0	1
Is_Liked	0	1

The **sys.columns** catalog view has been extended with two new columns: **graph_type** and **graph_type_desc**. They indicate the types of columns that the Database Engine generated. The type is indicated by a predefined numerical value and its related description. Microsoft does not provide a great deal of specifics about the columns, but you can find some details in the Microsoft documentation.

Example 31.24 uses the **sys.edge_constraints** and **sys.edge_constraint_clauses** catalog views to display metadata concerning existing edge constraints in the **LivesIn** edge table.

Example 31.24

```
USE graph_db;
SELECT
    EC.name AS Edge_constraint
    , OBJECT_NAME(EC.parent_object_id) AS Edge_table
    , OBJECT_NAME(ECC.from_object_id) AS From_node_table
    , OBJECT_NAME(ECC.to_object_id) AS To_node_table
   FROM sys.edge_constraints EC
     INNER JOIN sys.edge_constraint_clauses ECC
       ON EC.object_id = ECC.object_id
   WHERE EC.parent_object_id = object_id('LivesIn');
```

The result is

Edge_constraint	Edge_table	From_node_table	To_node_table
Emp_to_City	LivesIn	Employee	City

In Example 31.24, the **sys.edge_constraints** and **sys.edge_constraint_clauses** catalog views are joined together using the **object_id** column to display the information concerning the specified edge constraint (see Example 31.3). The constraint name (**Emp_to_City**) and the name of the corresponding edge table (**LivesIn**) are found in the **sys.edge_constraints** catalog view. Similarly, the names of the corresponding node tables are found in the **sys.edge_constraint_clauses** catalog view.

Graph Databases: System Functions

As you already know from Chapter 9, system functions are used to access catalog views. SQL Server contains six system functions related to Graph Databases. These functions are described in Table 31-1.

The next two examples demonstrate how these system functions can be used. Example 31.25 uses the NODE_ID_FROM_PARTS function.

System Function	Description
OBJECT_ID_FROM_NODE_ID	Extracts the object ID from a **$node_id** value
GRAPH_ID_FROM_NODE_ID	Extracts the graph ID from a **$node_id** value
NODE_ID_FROM_PARTS	Constructs a JSON node ID from an object ID and graph ID
OBJECT_ID_FROM_EDGE_ID	Extracts the object ID from an **$edge_id** value
EDGE_ID_FROM_PARTS	Constructs the edge ID from an object ID and identity
GRAPH_ID_FROM_EDGE_ID	Extracts the graph ID from an **$edge_id** value

Table 31-1 System Functions Related to SQL Server Graph Databases

Example 31.25

```
USE graph_db;
SELECT NODE_ID_FROM_PARTS(OBJECT_ID('dbo.Company'), 0);
```

The result is

```
{"type":"node","schema":"dbo","table":"Company","id":0}
```

Example 31.25 constructs a JSON document for a given **node_id** value from an **object_id** value, with the help of the NODE_ID_FROM_PARTS system function. This function has two parameters: the first is **object_id** value of the corresponding node table, and the second specifies the value of the **graph_id** column. As you can see from the result of this example, it returns the JSON document with four name/value pairs that correspond to the displayed value of the first row (ID = 0) in the **Company** node table.

NOTE The practical use of the NODE_ID_FROM_PARTS system function, as shown in Example 31.25, is in the case that you want to load data from another source and intend to assign the existing ID to each row as the graph ID.

Example 31.26 shows how the NODE_ID_FROM_PARTS function can be used to insert a row in an edge table.

Example 31.26

```
DECLARE @table1 INT = OBJECT_ID('dbo.Company');
DECLARE @table2 INT = OBJECT_ID('dbo.City');
INSERT INTO LocatedIn ($from_id, $to_id)
  VALUES (NODE_ID_FROM_PARTS(@table1, 1),
                    NODE_ID_FROM_PARTS(@table2, 2));
```

The batch in Example 31.26 obtains the object IDs from the relationship's originating and terminating node tables (**Company** and **City**) and saves them into the **@table1** and **@table2** variables, respectively. These variables are then used as the parameters of the NODE_ID_FROM_PARTS function to insert a new row into the **LocatedIn** edge table.

Querying Graph Data Using Relational Queries

This section demonstrates how Transact-SQL can be used to write queries over graphs. Graph query languages in general, and the query language of SQL Server Graph Databases in particular, are better suited to query *connected* data than Transact-SQL. In other words, query latency in a graph database is proportional to how much of the graph you choose to explore in a query, and is not proportional to the amount of data stored, which is the most important issue of Transact-SQL. The next five examples help to explain this.

First, Example 31.27 creates a relational table called **Employee1**, which has the same structure as the **Employee** node table (see Example 31.1).

Example 31.27

```
USE graph_db;
  CREATE TABLE  dbo.Employee1 (
  ID   INT  NOT NULL PRIMARY KEY,
  name   VARCHAR (100) NULL,
  sex   char (10) NULL);

INSERT INTO Employee1 VALUES (1,'Matthew Smith','Male');
INSERT INTO Employee1 VALUES (2,'Ann Jones','Female');
INSERT INTO Employee1 VALUES (3,'John Barrimore','Male');
INSERT INTO Employee1 VALUES (4,'James James','Male');
INSERT INTO Employee1 VALUES (5,'Elsa Bertoni','Female');
INSERT INTO Employee1 VALUES (6,'Elke Hansel','Female');
```

The only syntactical difference between the **Employee1** and the **Employee** table is that the former does not have the AS NODE option. (The six INSERT statements in Example 31.27 are identical to the corresponding statements for the **Employee** table in Example 31.2.)

To understand performance issues of performing queries over graphs in a relational database system such as the Database Engine, we will look at several examples concerning the **Is_Liked** relationship, introduced in the earlier section "Recursive Relationships." To implement such a recursive relationship using relational tables, you have to create a new table, as shown in Example 31.28.

Example 31.28

```
USE graph_db;
CREATE TABLE Employee1_Friend
  (EmployeeID INT NOT NULL,
   FriendID INT);
INSERT INTO  Employee1_Friend VALUES (1,2)
INSERT INTO  Employee1_Friend VALUES (1,3)
INSERT INTO  Employee1_Friend VALUES (1,4)
INSERT INTO  Employee1_Friend VALUES (2,3)
INSERT INTO  Employee1_Friend VALUES (2,5)
INSERT INTO  Employee1_Friend VALUES (2,6)
INSERT INTO  Employee1_Friend VALUES (3,4)
INSERT INTO  Employee1_Friend VALUES (4,5)
INSERT INTO  Employee1_Friend VALUES (4,6)
INSERT INTO  Employee1_Friend VALUES (5,6)
```

The **Employee1_Friend** table has two columns. The first, **EmployeeID**, represents the ID of an employee, while the second, **FriendID**, is the ID of another employee who likes the employee listed in the **EmployeeID** column. Therefore, the first INSERT statement in Example 31.28 can be interpreted as the employee with ID = 2 (Ann Jones) likes the employee with ID = 1 (Matthew Smith).

The next three examples show how you can implement the queries from Examples 31.15, 31.16, and 31.17 using T-SQL. Example 31.29 implements the query from Example 31.15.

Example 31.29

```
-- Corresponds to Example 31.15
  USE graph_db;
SELECT E1.name
   FROM Employee1 E1 JOIN Employee1_Friend
      ON Employee1_Friend.FriendID = E1.ID
                  JOIN Employee1 E2
      ON Employee1_Friend.EmployeeID = E2.ID
   WHERE E2.name = 'Matthew Smith';
```

Concerning performance, the query in Example 31.29 is rather inexpensive because the number of qualified rows is restricted with the condition in the WHERE clause (E2.name = 'Matthew Smith'). If you create an index for the **name** column of the **Employee1** table, the query in Example 31.29 will be executed fairly quickly.

As you already know from Example 31.15, a recursive relationship such as **Is_Liked** must not be reflexive, meaning that an employee can be liked by another one, but the former must not share the same feelings. For this reason, Examples 31.15 and 31.16 display the different result sets.

The relational implementation of Example 31.16 is given in Example 31.30 and the performance of this query is similar to the performance of Example 31.29. (The result of Example 31.30 does not contain any rows.)

Example 31.30

```
-- Corresponds to Example 31.16
USE graph_db;
SELECT E1.name
   FROM Employee1 E1 JOIN Employee1_Friend
    ON Employee1_Friend.EmployeeID = E1.ID
                 JOIN Employee1 E2
    ON Employee1_Friend.FriendID  =  E2.ID
  WHERE E2.name = 'Matthew Smith';
```

Example 31.31, which answers the query "Find employees who like the employees who like Ann Jones," is syntactically and computationally complex.

Example 31.31

```
-- Corresponds to Example 31.17
USE graph_db;
SELECT E1.name AS EMP_Name, E2.name AS FriendOfFriend
   FROM Employee1_Friend Ef1 JOIN Employee1 E1
   ON Ef1.EmployeeID = E1.ID
   JOIN Employee1_Friend Ef2 ON Ef2.EmployeeID = Ef1.FriendID
   JOIN Employee1 E2 ON Ef2.FriendID = E2.ID
   WHERE E1.name = 'Ann Jones' AND Ef2.FriendID <> E1.ID;
```

In Example 31.31, the number of JOIN operations is high and the WHERE clause contains the inequality (Ef2.FriendID <> E1.ID). As you already know from the section "Query Analysis" in Chapter 19, an expression with the NOT (<>) operator cannot be used by the optimizer as a search argument, and the only access the optimizer uses in this case is the table scan. Therefore, the execution of this query will be slow.

Things get more complex if you dig deeper into the graph with the **Is_Liked** relationship. Though it is possible to get an answer to the query given in Example 31.31 in a fairly short period of time, queries that extend to more degrees of liking will have very poor performance. Therefore, the higher the degree of liking, the poorer the performance.

Summary

SQL, as a common database language, does not deal very well with connected data because the number of join operations (inner joins and self-referencing joins) increases significantly as the dataset gets bigger and the traversal of a graph goes deeper. For this reason, Microsoft has implemented SQL Server Graph Databases as an important tool to store connected data efficiently.

A graph in SQL Server Graph Databases is a collection of node tables, which store nodes of the graph, and edge tables, which store relationships (edges of the graph) between nodes. As "first-class citizens" in Graph Databases, nodes and edges of a graph can have attributes (properties) associated with them. Graph extensions are fully integrated in the Database Engine. This means that the same storage engine used to store relational data is also used to store graph data. Additionally, you can retrieve graph data and relational data via a single query. SQL Server Graph Databases also supports all the security and compliance features available in SQL Server.

The next chapter describes SQL Server Machine Learning Services in general and support for R in particular.

Exercises

Using the following exercises, extend the graph presented in this chapter with a new node table called **Restaurant** and a new edge table called **Visits**.

E.31.1 Create a node table, **Restaurant**, with the following columns:

- ID (INT PK)
- name (VARCHAR(20))
- city (VARCHAR(20))

E.31.2 Insert three rows in the **Restaurant** table:

- (1, 'A', 'Hoboken')
- (2, 'B', 'Lexington')
- (3, 'C', 'Madison')

E.31.3 Create the edge table called **Visits**. Add a column called **rating**, for use by employees to rate restaurants where they eat. Note that the edge between the nodes **Employee** and **Restaurant** is directed, from **Employee** to **Restaurant**.

E.31.4 Insert the following six rows into the **Visits** table:

```
USE graph_db;
INSERT INTO Visits VALUES ((SELECT $node_id FROM Employee WHERE id = 1),
      (SELECT $node_id FROM Restaurant WHERE id = 1), 5);
INSERT INTO Visits VALUES ((SELECT $node_id FROM Employee WHERE id = 2),
      (SELECT $node_id FROM Restaurant WHERE id = 2), 3);
INSERT INTO Visits VALUES ((SELECT $node_id FROM Employee WHERE id = 3),
      (SELECT $node_id FROM Restaurant WHERE id = 3), 2);
INSERT INTO Visits VALUES ((SELECT $node_id FROM Employee WHERE id = 4),
      (SELECT $node_id FROM Restaurant WHERE id = 3), 3);
INSERT INTO Visits VALUES ((SELECT $node_id FROM Employee WHERE id = 5),
      (SELECT $node_id FROM Restaurant WHERE id = 3), 1);
INSERT INTO Visits VALUES ((SELECT $node_id FROM Employee WHERE id = 6),
      (SELECT $node_id FROM Restaurant WHERE id = 2), 4);
```

E.31.5 Find the names of restaurants that Ann Jones visits.

E.31.6 Find the names of restaurants visited by employees who Ann Jones likes.