Table of Contents

Graph processing with SQL Server 2017

4/29/2017 • 2 min to read • Edit Online

THIS TOPIC APPLIES TO: SQL Server (starting with 2017) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server offers graph database capabilities to model many-to-many relationships. The graph relationships are integrated into Transact-SQL and receive the benefits of using SQL Server as the foundational database management system.

What is a graph database?

A graph database is a collection of nodes (or vertices) and edges (or relationships). A node represents an entity (for example, a person or an organization) and an edge represents a relationship between the two nodes that it connects (for example, likes or friends). Both nodes and edges may have properties associated with them. Here are some features that make a graph database unique:

- Edges or relationships are first class entities in a Graph Database and can have attributes or properties associated with them.
- A single edge can flexibly connect multiple nodes in a Graph Database.
- You can express pattern matching and multi-hop navigation queries easily.
- You can express transitive closure and polymorphic queries easily.

When to use a graph database

There is nothing a graph database can achieve, which cannot be achieved using a relational database. However, a graph database can make it easier to express certain kind of queries. Also, with specific optimizations, certain queries may perform better. Your decision to choose one over the other can be based on following factors:

- Your application has hierarchical data. The HierarchyID datatype can be used to implement hierarchies, but it has some limitations. For example, it does not allow you to store multiple parents for a node.
- Your application has complex many-to-many relationships; as application evolves, new relationships are added.
- You need to analyze interconnected data and relationships.

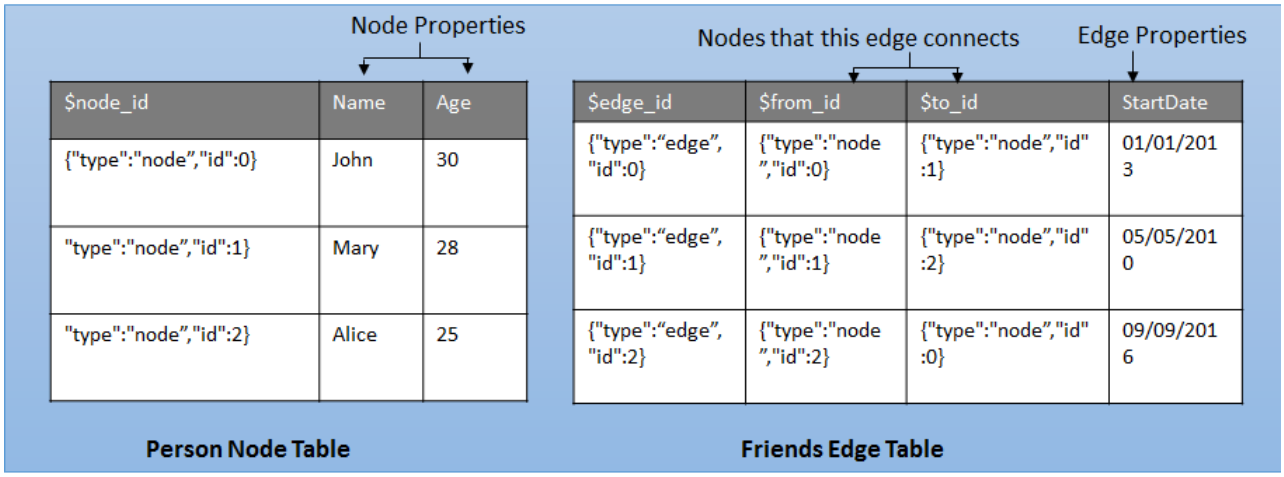
Graph features introduced in SQL Server 2017

We are starting to add graph extension to SQL Server, to make storing and querying graph data easier. Following features are introduced in the first release.

Create graph objects

Transact-SQL extensions will allow users to create node or edge tables. Both nodes and edges can have properties associated to them. Since, nodes and edges are stored as tables, all the operations that are supported on relational tables are supported on node or edge table. Here is an example:

```
CREATE TABLE Person (ID INTEGER PRIMARY KEY, name VARCHAR(100)) AS NODE;
CREATE TABLE friends (StartDate date) AS EDGE;
```



Nodes and Edges are stored as tables

Query language extensions

New `MATCH` clause is introduced to support pattern matching and multi-hop navigation through the graph. The `MATCH` function uses ASCII-art style syntax for pattern matching. For example:

```
-- Find friends of John
SELECT Person2.Name
FROM Person Person1, Friends, Person Person2
WHERE MATCH(Person1-(Friends)->Person2)
AND Person1.Name = 'John';
```

Fully integrated in SQL Server

Graph extensions are fully integrated in SQL Server engine. We use the same storage engine, metadata, query processor, etc. to store and query graph data. This enables users to query across their graph and relational data in a single query. Users can also benefit from combining graph capabilities with other SQL Server technologies like columnstore, HA, R services, etc. SQL graph database also supports all the security and compliance features available with SQL Server.

Tooling and ecosystem

Users benefit from existing tools and ecosystem that SQL Server offers. Tools like backup and restore, import and export, BCP just work out of the box. Other tools or services like SSIS, SSRS or PowerBI will work with graph tables, just the way they work with relational tables.

Next steps

Read the SQL Graph Database - Architecture

SQL Graph Architecture

5/31/2017 • 8 min to read • Edit Online

THIS TOPIC APPLIES TO: SQL Server (starting with 2017) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Learn how SQL Graph is architected. Knowing the basics will make it easier to understand other SQL Graph articles.

SQL Graph Database

Users can create one graph per database. A graph is a collection of node and edge tables. Node or edge tables can be created under any schema in the database, but they all belong to one logical graph. A node table is collection of similar type of nodes. For example, a Person node table holds all the Person nodes belonging to a graph. Similarly, an edge table is a collection of similar type of edges. For example, a Friends edge table holds all the edges that connect a Person to another Person. Since nodes and edges are stored in tables, most of the operations supported on regular tables are supported on node or edge tables.

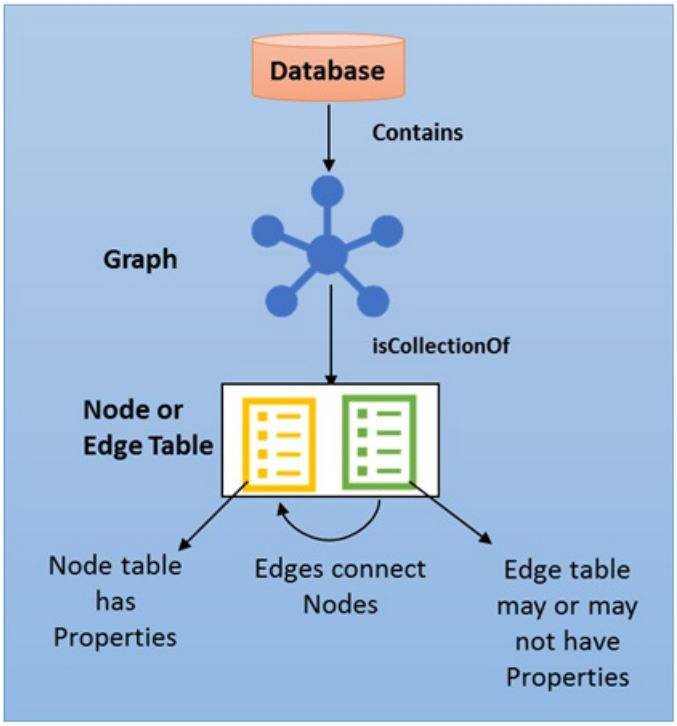


Figure 1: SQL Graph database architecture

Node Table

A node table represents an entity in a graph schema. Every time a node table is created, along with the user defined columns, an implicit `$node_id` column is created, which uniquely identifies a given node in the database. The values in `$node_id` are automatically generated and are a combination of `object_id` of that node table and an internally generated bigint value. However, when the `$node_id` column is selected, a computed value in the form of a JSON string is displayed. Also, `$node_id` is a pseudo column, that maps to an internal name with hex string in it. When you select `$node_id` from the table, the column name will appear as `$node_id_\<hex_string>`. Using pseudo-column names in quereis is the recommended way of querying the internal `$node_id` column and using internal name with hex string should be avoided.

It is recommended that users create a unique constraint or index on the `$node_id` column at the time of creation of

node table, but if one is not created, a default unique, non-clustered index is automatically created.

Edge Table

An edge table represents a relationship in a graph. Edges are always directed and connect two nodes. An edge table enables users to model many-to-many relationships in the graph. An edge table may or may not have any user defined attributes in it. Every time an edge table is created, along with the user defined attributes, three implicit columns are created in the edge table:

COLUMN NAME	DESCRIPTION
`$edge_id`	Uniquely identifies a given edge in the database. It is a generated column and the value is a combination of object_id of the edge table and a internally generated bigint value. However, when the `$edge_id` column is selected, a computed value in the form of a JSON string is displayed. `$edge_id` is a pseudo-column, that maps to an internal name with hex string in it. When you select `$edge_id` from the table, the column name will appear as `$edge_id_\<hex_string>`. Using pseudo-column names in quereis is the recommended way of querying the internal `$edge_id` column and using internal name with hex string should be avoided.
`$from_id`	Stores the `$node_id` of the node, from where the edge originates.
`$to_id`	Stores the `$node_id` of the node, at which the edge terminates.

The nodes that a given edge can connect is governed by the data inserted in the `$from_id` and `$to_id` columns. In the first release, it is not possible to define constraints on the edge table, to restrict it from connecting any two type of nodes. That is, an edge can connect any two nodes in the graph, regardless of their types.

Similar to the `$node_id` column, it is recommended that users create a unique index or constraint on the `$edge_id` column at the time of creation of the edge table, but if one is not created, a default unique, non-clustered index is automatically created on this column. It is also recommended, for OLTP scenarios, that users create an index on (`$from_id` , `$to_id`) columns, for faster lookups in the direction of the edge.

Figure 2 shows how node and edge tables are stored in the database.

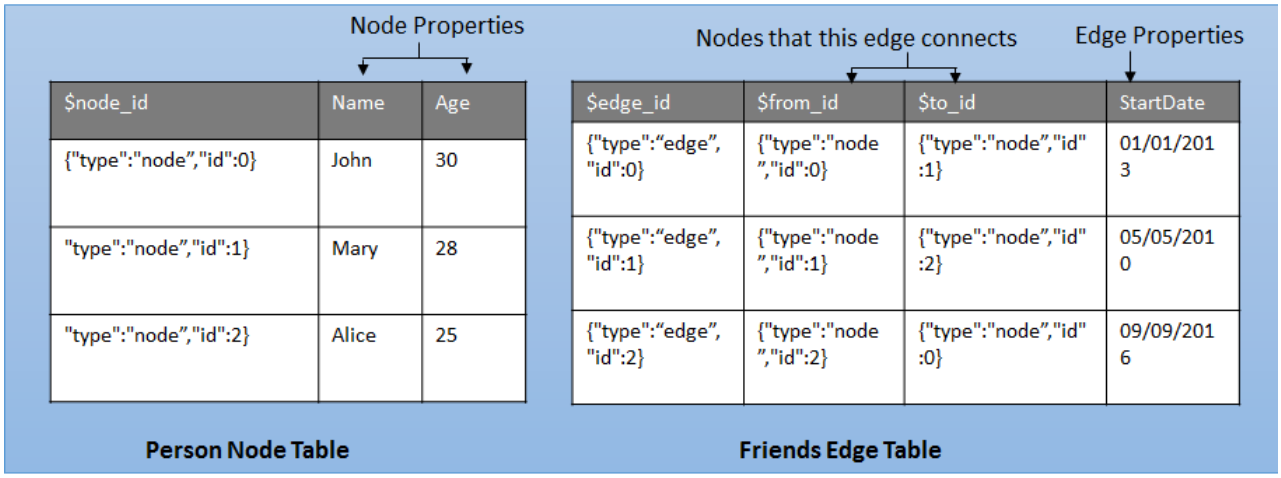


Figure 2: Node and edge table representation

Metadata

Use these metadata views to see attributes of a node or edge table.

SYS.TABLES

The following new, bit type, columns will be added to SYS.TABLES. If `is_node` is set to 1, that indicates that the table is a node table and if `is_edge` is set to 1, that indicates that the table is an edge table.

COLUMN NAME	DATA TYPE	DESCRIPTION
is_node	bit	1 = this is a node table
is_edge	bit	1 = this is an edge table

SYS.COLUMNS

The `sys.columns` view contains additional columns `graph_type` and `graph_type_desc`, that indicate the type of the column in node and edge tables.

COLUMN NAME	DATA TYPE	DESCRIPTION
graph_type	int	Internal column with a set of values. The values are between 1-8 for graph columns and NULL for others.
graph_type_desc	nvarchar(60)	internal column with a set of values

The following table lists the valid values for `graph_type` column

COLUMN VALUE	DESCRIPTION
1	GRAPH_ID
2	GRAPH_ID_COMPUTED
3	GRAPH_FROM_ID
4	GRAPH_FROM_OBJ_ID
5	GRAPH_FROM_ID_COMPUTED
6	GRAPH_TO_ID
7	GRAPH_TO_OBJ_ID
8	GRAPH_TO_ID_COMPUTED

`sys.columns` also stores information about implicit columns created in node or edge tables. Following information can be retrieved from sys.columns, however, users cannot select these columns from a node or edge table.

Implicit columns in a node table

COLUMN NAME	DATA TYPE	IS_HIDDEN	COMMENT
graph_id_<hex_string>	BIGINT	1	internal graph_id column
$node_id_<hex_string>	NVARCHAR	0	External node id column

Implicit columns in an edge table

COLUMN NAME	DATA TYPE	IS_HIDDEN	COMMENT
graph_id_<hex_string>	BIGINT	1	internal graph_id column
$edge_id_<hex_string>	NVARCHAR	0	external edge id column
from_obj_id_<hex_string>	INT	1	internal from node object id
from_id_<hex_string>	BIGINT	1	Internal from node graph_id
$from_id_<hex_string>	NVARCHAR	0	external from node id
to_obj_id_<hex_string>	INT	1	internal to node object id
to_id_<hex_string>	BIGINT	1	Internal to node graph_id
$to_id_<hex_string>	NVARCHAR	0	external to node id

System Functions

The following built-in functions are added. These will help users extract information from the generated columns. Note that, these methods will not validate the input from the user. If the user specifies an invalid `sys.node_id` the method will extract the appropriate part and return it. For example, OBJECT_ID_FROM_NODE_ID will take a `$node_id` as input and will return the object_id of the table, this node belongs to.

BUILT-IN	DESCRIPTION
OBJECT_ID_FROM_NODE_ID	Extract the object_id from a node_id
GRAPH_ID_FROM_NODE_ID	Extract the graph_id from a node_id
NODE_ID_FROM_PARTS	Construct a node_id from an object_id and a graph_id
OBJECT_ID_FROM_EDGE_ID	Extract object_id from edge_id
GRAPH_ID_FROM_EDGE_ID	Extract identity from edge_id
EDGE_ID_FROM_PARTS	Construct edge_id from object_id and identity

Transact-SQL reference

Learn the Transact-SQL extensions introduced in SQL Server, that enable creating and querying graph objects. The query language extensions help query and traverse the graph using ASCII art syntax.

Data Definition Language (DDL) statements

TASK	RELATED TOPIC	NOTES
CREATE TABLE	CREATE TABLE (Transact-SQL)	`CREATE TABLE` is now extended to support creating a table AS NODE or AS EDGE. Note that an edge table may or may not have any user defined attributes.
ALTER TABLE	ALTER TABLE (Transact-SQL)	Node and edge tables can be altered the same way a relational table is, using the `ALTER TABLE`. Users can add or modify user defined columns, indexes or constraints. However, altering internal graph columns, like `$node_id` or `$edge_id`, will result in an error.
CREATE INDEX	CREATE INDEX (Transact-SQL)	Users can create indexes on pseudo-columns and user defined columns in node and edge tables. All index types are supported, including clustered and non-clustered columnstore indexes.
DROP TABLE	DROP TABLE (Transact-SQL)	Node and edge tables can be dropped the same way a relational table is, using the `DROP TABLE`. However, in this release, there are no constraints to ensure that no edges point to a deleted node and cascaded deletion of edges, upon deletion of a node or node table is not supported. It is recommended that if a node table is dropped, users drop any edges connected to the nodes in that node table manually to maintain the integrity of the graph.

Data Manipulation Language (DML) statements

TASK	RELATED TOPIC	NOTES
INSERT	INSERT (Transact-SQL)	Inserting into a node table is no different than inserting into a relational table. The values for `$node_id` column is automatically generated. Trying to insert a value in `$node_id` or `$edge_id` column will result in an error. Users must provide values for `$from_id` and `$to_id` columns while inserting into an edge table. `$from_id` and `$to_id` are the `$node_ids` of the nodes a given edge connects.

TASK	RELATED TOPIC	NOTES
DELETE	DELETE (Transact-SQL)	Data from node or edge tables can be deleted in same way as it is deleted from relational tables. However, in this release, there are no constraints to ensure that no edges point to a deleted node and cascaded deletion of edges, upon deletion of a node is not supported. It is recommended that whenever a node is deleted, all the connecting edges to that node are also deleted, to maintain the integrity of the graph.
UPDATE	UPDATE (Transact-SQL)	Values in user defined columns can be updated using the UPDATE statement. Updating the internal graph columns, `$node_id`, `$edge_id`, `$from_id` and `$to_id` is not allowed.
MERGE	MERGE (Transact-SQL)	`MERGE` statement is not supported on a node or edge table.

Query Statements

TASK	RELATED TOPIC	NOTES
SELECT	SELECT (Transact-SQL)	Nodes and edges are stored as tables internally, hence most of the operations supported on a table in SQL Server or Azure SQL Database are supported on the node and edge tables
MATCH	MATCH (Transact-SQL)	MATCH built-in is introduced to support pattern matching and traversal through the graph.

Limitations and known issues

There are certain limitations on node and edge tables in this release:

- Local or global temporary tables cannot be node or edge tables.
- Table types and table variables cannot be declared as a node or edge table.
- Node and edge tables cannot be created as system-versioned temporal tables.
- Node and edge tables cannot be memory optimized tables.
- Users cannot update the $from_id and $to_id columns of an edge using UPDATE statement. To update the nodes that an edge connects, users will have to insert the new edge pointing to new nodes and delete the previous one.
- Cross database queries on graph objects are not supported.

Next Steps

To get started with the new syntax, see SQL Graph Database - Sample

Create a graph database and run some pattern matching queries using T-SQL

5/31/2017 • 4 min to read • Edit Online

THIS TOPIC APPLIES TO: SQL Server (starting with 2017) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This sample provides a Transact-SQL script to create a graph database with nodes and edges and then use the new MATCH clause to match some patterns and traverse through the graph.

Sample Schema

This sample creates a graph schema, as showed in Figure 1, for a hypothetical social network that has People, Restaurant and City nodes. These nodes are connected to each other using Friends, Likes, LivesIn and LocatedIn edges.

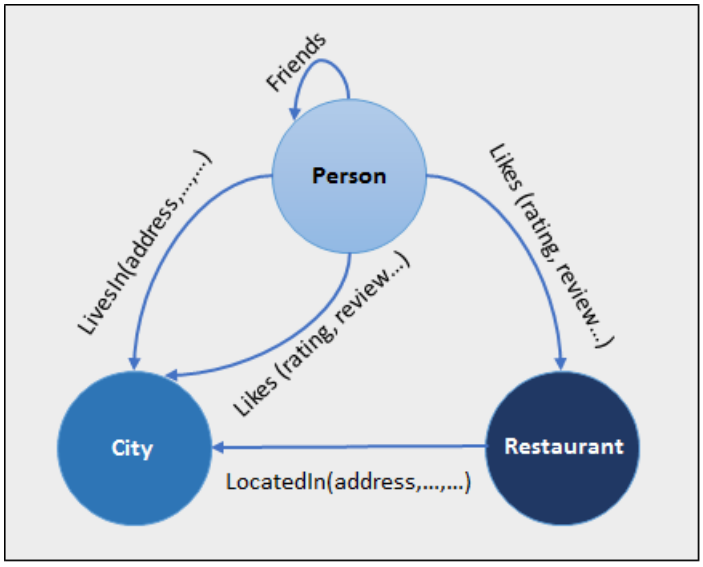


Figure 1: Sample schema with restaurant, city, person nodes and LivesIn, LocatedIn, Likes edges.

Sample Script

```
-- Create a graph demo database
CREATE DATABASE graphdemo;
go

USE  graphdemo;
go

-- Create NODE tables
CREATE TABLE Person (
  ID INTEGER PRIMARY KEY,
  name VARCHAR(100)
) AS NODE;

CREATE TABLE Restaurant (
  ID INTEGER NOT NULL,
  name VARCHAR(100),
  city VARCHAR(100)
) AS NODE;
```

```
CREATE TABLE City (
  ID INTEGER PRIMARY KEY,
  name VARCHAR(100),
  stateName VARCHAR(100)
) AS NODE;

-- Create EDGE tables.
CREATE TABLE likes (rating INTEGER) AS EDGE;
CREATE TABLE friendOf AS EDGE;
CREATE TABLE livesIn AS EDGE;
CREATE TABLE locatedIn AS EDGE;

-- Insert data into node tables. Inserting into a node table is same as inserting into a regular table
INSERT INTO Person VALUES (1,'John');
INSERT INTO Person VALUES (2,'Mary');
INSERT INTO Person VALUES (3,'Alice');
INSERT INTO Person VALUES (4,'Jacob');
INSERT INTO Person VALUES (5,'Julie');

INSERT INTO Restaurant VALUES (1,'Taco Dell','Bellevue');
INSERT INTO Restaurant VALUES (2,'Ginger and Spice','Seattle');
INSERT INTO Restaurant VALUES (3,'Noodle Land', 'Redmond');

INSERT INTO City VALUES (1,'Bellevue','wa');
INSERT INTO City VALUES (2,'Seattle','wa');
INSERT INTO City VALUES (3,'Redmond','wa');

-- Insert into edge table. While inserting into an edge table,
-- you need to provide the $node_id from $from_id and $to_id columns.
INSERT INTO likes VALUES ((SELECT $node_id FROM Person WHERE id = 1),
       (SELECT $node_id FROM Restaurant WHERE id = 1),9);
INSERT INTO likes VALUES ((SELECT $node_id FROM Person WHERE id = 2),
      (SELECT $node_id FROM Restaurant WHERE id = 2),9);
INSERT INTO likes VALUES ((SELECT $node_id FROM Person WHERE id = 3),
      (SELECT $node_id FROM Restaurant WHERE id = 3),9);
INSERT INTO likes VALUES ((SELECT $node_id FROM Person WHERE id = 4),
      (SELECT $node_id FROM Restaurant WHERE id = 3),9);
INSERT INTO likes VALUES ((SELECT $node_id FROM Person WHERE id = 5),
      (SELECT $node_id FROM Restaurant WHERE id = 3),9);

INSERT INTO livesIn VALUES ((SELECT $node_id FROM Person WHERE id = 1),
      (SELECT $node_id FROM City WHERE id = 1));
INSERT INTO livesIn VALUES ((SELECT $node_id FROM Person WHERE id = 2),
      (SELECT $node_id FROM City WHERE id = 2));
INSERT INTO livesIn VALUES ((SELECT $node_id FROM Person WHERE id = 3),
      (SELECT $node_id FROM City WHERE id = 3));
INSERT INTO livesIn VALUES ((SELECT $node_id FROM Person WHERE id = 4),
      (SELECT $node_id FROM City WHERE id = 3));
INSERT INTO livesIn VALUES ((SELECT $node_id FROM Person WHERE id = 5),
      (SELECT $node_id FROM City WHERE id = 1));

INSERT INTO locatedIn VALUES ((SELECT $node_id FROM Restaurant WHERE id = 1),
      (SELECT $node_id FROM City WHERE id =1));
INSERT INTO locatedIn VALUES ((SELECT $node_id FROM Restaurant WHERE id = 2),
      (SELECT $node_id FROM City WHERE id =2));
INSERT INTO locatedIn VALUES ((SELECT $node_id FROM Restaurant WHERE id = 3),
      (SELECT $node_id FROM City WHERE id =3));

-- Insert data into the friendof edge.
INSERT INTO friendof VALUES ((SELECT $NODE_ID FROM person WHERE ID = 1), (SELECT $NODE_ID FROM person WHERE ID
= 2));
INSERT INTO friendof VALUES ((SELECT $NODE_ID FROM person WHERE ID = 2), (SELECT $NODE_ID FROM person WHERE ID
= 3));
INSERT INTO friendof VALUES ((SELECT $NODE_ID FROM person WHERE ID = 3), (SELECT $NODE_ID FROM person WHERE ID
= 1));
INSERT INTO friendof VALUES ((SELECT $NODE_ID FROM person WHERE ID = 4), (SELECT $NODE_ID FROM person WHERE ID
= 2));
INSERT INTO friendof VALUES ((SELECT $NODE_ID FROM person WHERE ID = 5), (SELECT $NODE_ID FROM person WHERE ID
= 4));
```

```
-- Find Restaurants that John likes
SELECT Restaurant.name
FROM Person, likes, Restaurant
WHERE MATCH (Person-(likes)->Restaurant)
AND Person.name = 'John';

-- Find Restaurants that John's friends like
SELECT Restaurant.name
FROM Person person1, Person person2, likes, friendOf, Restaurant
WHERE MATCH(person1-(friendOf)->person2-(likes)->Restaurant)
AND person1.name='John';

-- Find people who like a restaurant in the same city they live in
SELECT Person.name
FROM Person, likes, Restaurant, livesIn, City, locatedIn
WHERE MATCH (Person-(likes)->Restaurant-(locatedIn)->City AND Person-(livesIn)->City);
```

Clean Up

Clean up the schema and database created for the sample.

```
USE graphdemo;
go

DROP TABLE IF EXISTS likes;
DROP TABLE IF EXISTS Person;
DROP TABLE IF EXISTS Restaurant;
DROP TABLE IF EXISTS City;
DROP TABLE IF EXISTS friendOf;
DROP TABLE IF EXISTS livesIn;
DROP TABLE IF EXISTS locatedIn;

USE master;
go
DROP DATABASE graphdemo;
go
```

Script explanation

This script uses the new T-SQL syntax to create node and edge tables. Shows how to insert data into node and edge tables using `INSERT` statement and also shows how to use `MATCH` clause for pattern matching and navigation.

COMMAND	NOTES
CREATE TABLE (Transact-SQL)	Create graph node or edge table
INSERT (Transact-SQL)	Insert into a node or edge table
MATCH (Transact-SQL)	Use MATCH to match a pattern or traverse through the graph