Introduction to GraphView

Welcome to GraphView, a middleware that helps you use Microsoft SQL Server and Azure SQL Database to manage and process graphs in a highly efficient manner. In this document, we provide a quick introduction to the library, including its design philosophy, basic concepts, functionality and programming APIs. We refer readers to our papers and technical reports for detailed algorithms and implementation details.

What is GraphView

GraphView is a library that connects to a SQL Server or an Azure SQL Database instance, stores graph data in tables and queries graphs through an SQL-extended language. It is not an independent database, but a middleware that accepts graph operations and translates them to T-SQL executed in SQL Server or Azure SQL Database. As such, GraphView can be viewed as a special connector to SQL Server/Azure SQL Database. Developers will experience no differences than the default SQL connector provided by the .NET framework (i.e., `SqlConnection`), only except that this new connector accepts graph-oriented statements.

Design philosophy

Graph data is becoming ubiquitous. Today's solutions for managing graphs have been mostly centered on the concept of NoSQL. A common argument against SQL databases for graph processing is that graph traversals are implemented as a sequence of table joins and join operations are expensive. This statement, however, is inaccurate. It is too vague to substantiate the claim that SQL databases are fundamentally incapable of efficiently handling graph-based data. First, not all join operations are expensive; depending on input sizes and available data structures such as indexes and views, a join operation can be efficient. More importantly, a graph traversal in a native graph database is not join-free. To understand this, consider a native graph database in which every node is physically represented as a record that contains the node's properties, as well as one or more adjacency lists of its neighbors. In this graph database, a traversal from a node to its neighbors involves three steps: (1) retrieve the node's physical record from the data store, (2) iterate through its adjacency list and (3) for each neighbor, retrieve its physical record using its ID (or reference). The last retrieval step is essentially a join; it is logically equivalent to a nested-loop join that uses a record's key—a logical address or reference—to locate a record. The nested-loop join in SQL databases is not always expensive, when there is an index on nodes' IDs. It can even be fairly efficient in the context of main-memory SQL databases when index accesses involve no random disk reads.

Thus, we postulate that the second step—iterating through neighbors—in a graph traversal is the key gap between native graph stores and SQL databases. In conventional SQL databases, data is usually normalized to avoid redundancy and update anomalies. Entities in a graph are highly connected and mainly present many-to-many relationships. By data normalization, a many-to-many relationship between two entities yields a junction table, with two columns referencing the two entities respectively. Such an organization means that an entity's properties are separated from graph topology. To traverse from an entity to its neighbors, the query engine needs an additional join to look up the junction table to obtain the topology information associated with the node, yielding poorer cache locality than native graph databases.

The idea of GraphView is to bridge this gap between native graph stores and SQL databases, making SQL Server (and Azure SQL Database) behave in the same way as a native graph store. GraphView is built on top of SQL Server, re-using SQL functions whenever possible. By using SQL functions appropriately, the physical data representation and runtime behavior of GraphView closely resemble those of native graph databases. In fact, thanks to years of research and development of SQL Server, GraphView inherits many sophisticated optimizations that have been neglected by native graph databases, providing unparalleled performance advantages.

Features

GraphView is a DLL library through which you manage graph data in SQL Server (version 2008 and onward) and Azure SQL Database (v12 and onward). It provides all features a standard graph database is expected to have. And in addition, since GraphView relies on SQL Server, it inherits numerous features and performance enhancements commonly available in the relational world that are often missing in native graph databases.

GraphView offers the following major features:

- Graph database. A graph database in GraphView is a conventional SQL database. The graph database consists of one or more types of nodes and edges, each of which may have one or more properties.

- Data manipulations. GraphView provides an SQL-extended language for graph manipulation, including inserting/deleting nodes and edges. The syntax is similar to `INSERT`/`DELETE` statements in SQL, but is extended to accommodate graph semantics.

- Queries. GraphView's query language allows users to match graph patterns against the graph in a graph database. The query language extends the SQL `SELECT` statement with a `MATCH` clause, in which the graph pattern is

specified. Coupled with loop/iteration statements from T-SQL, the language also allows users to perform iterative computations over the graph.

- Indexes. To accelerate query processing, GraphView also allows users to create indexes. All indexes supported by SQL Server and Azure SQL Database are available, including not only conventional B-tree indexes but also new indexing technologies such as columnstore indexes.

- Transactions. All operations in GraphView are transaction-safe. What is more, there is no limit on a transaction's scope. So a transaction can span nodes, edges or even graphs.

- SQL-related features. GraphView inherits many features from SQL Server and Azure SQL Database. Below is a short list of features that are crucial to administration tasks:

    - Access control. GraphView uses the authentication mechanism of SQL Server to control accesses to graph databases. A user can access a graph database if SQL Server says so.

    - Replication. GraphView stores graph data in a SQL Server database. A replication of the database will result in a replication of all graph data.

    - Backup. GraphView maintains SQL Server databases that are visible to SQL Server administrators. Administrators can apply backup operations to the database explicitly.

Graph Databases

GraphView uses a SQL database to host graph data. GraphView connects to an existing SQL database using a SQL connection string. It does not provide any API's to manage the database. Any database management, such as setting the database's properties or managing user accounts, should be done by T-SQL statements or through SQL Server Management Studio.

GraphView maintains all data and meta-data using SQL objects such as tables, indexes and user-defined functions. These objects are visible to SQL users with appropriate permissions. As a middleware, GraphView does not prevent you from modifying these objects bypassing GraphView. It is advised that you apply operations, such as replication and backup, to the entire database and do not change these SQL objects separately. Doing so may result in an inconsistent/corrupted state, from which GraphView is unable to recover.

Opening/closing a graph database

You open a graph database by instantiating a `GraphViewConnection` object. `GraphViewConnection` is similar to `SqlConnection` and is instantiated by a connection string of a SQL database.

```
using GraphView;
……
string connectionString = "Data Source= (local); Initial
Catalog=GraphTesting; Integrated Security=true;";

GraphViewConnection gdb =
  new GraphViewConnection(connectionString);

try {
  // Connects to a database. Creates objects needed by
  // GraphView if they do not exist.
  gdb.Open(true);
}
catch(DatabaseException e) {
  // Exception handling goes here
}
```

When the connection string points to an Azure SQL Database instance, you open a graph database in Azure:

```
using GraphView;
……
var sqlConnectionBuilder = new SqlConnectionStringBuilder();
sqlConnectionBuilder["Server"] =
  "tcp:graphview.database.windows.net,1433";
sqlConnectionBuilder["User ID"] = "xxx";
sqlConnectionBuilder["Password"] = "xxx";
```

```
sqlConnectionBuilder[“Database”] = “GraphTesting”;
GraphViewConnection gdb =
  new GraphViewConnection(connectionString);

try {
  // Connects to a database. Creates objects needed by
  // GraphView if they do not exist.
  gdb.Open(true);
}
catch(DatabaseException e) {
  // Exception handling goes here
}
```

The SQL database GraphView connects to must be CLR-enabled. For Azure SQL Database (v12 and onward), CLR is automatically enabled. For SQL Server, CLR must be enabled separately by the database administrator.

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

GraphView maintains a number of SQL objects in the SQL database to host data and meta-data of the graph. When a graph database is opened, GraphView automatically checks if these objects exist. A `DatabaseException` exception is thrown, if some objects are missing, unless the flag of `Open()` is set to be true, in which case missing objects will be created automatically.

You close a graph database connection by invoking `GraphViewConnection.Close()` or `GraphViewConnection.Dispose()`. The method internally closes `SqlConnection` associated with the instance. You should expect the same behavior/result when invoking `GraphViewConnection.Close()` in unusual circumstances. For instance, when the connection is closed, any uncommitted changes will be lost. And a closed connection cannot be re-opened. A new connection to the graph database needs to be instantiated.

```
using GraphView;
……
GraphViewConnection gdb =
  new GraphViewConnection(connectionString);
try {
  gdb.Open(true);
  // Data manipulations go here
  gdb.Close();
}
catch(DatabaseException e) {
  // Exception handling goes here
```

```
}
```

Creating node types (node tables)

A graph consists of nodes and edges, both associated with zero or more properties. In GraphView, you create node types to define the graph. A node type is analogous to a table and specifies the format of a collection of graph nodes: node properties and outgoing edges connecting to other types of nodes (including the type of itself). Each node instance, analogous to a tuple in a table, belongs to one type. In this manual, "node type" and "node table" are used interchangeably.

A node type is created by the `CREATE TABLE` statement. The syntax of the statement is the same as its SQL counterpart, except that each field has an annotation, specifying the role of the field, being a node property or an adjacency list.

When a field is annotated as an adjacency list—a collection of edges pointing to the node's neighbors, additional annotations further specify two things: which type of nodes these edges point to and the edges' properties, if there are any. As such, the `CREATE TABLE` statement defines both nodes and their associated edges.

```
CREATE TABLE EmployeeNode (
  [ColumnRole:"NodeId"]
  WorkId varchar(32),
  [ColumnRole:"Property"]
  Name varchar(32),
  [ColumnRole:"Edge", Reference:"EmployeeNode"]
  Colleagues VARBINARY,
  [ColumnRole:"Edge", Reference:"ClientNode",
    Attributes: {credit:"int", note:"string"}]
  Clients VARBINARY
)
```

The above statement defines the node type `EmployeeNode`. Compared to the conventional `CREATE TABLE` statement, each field/column is preceded by an annotation. The BNF syntax of the annotation is as follows:

```
<column_annotation> ::=
'[' ColumnRole : [ NODEID | PROPERTY | EDGE ],
   [ Reference: sink_table,
     [ Attributes: {property_name: "property_type" [ , … n ] } ]
   ]
']'
```

In the annotation, `ColumnRole` specifies a field as one of the three roles, `NodeID`, `Property` or `Edge`. `NodeID` is equivalent to the primary key specification in SQL. Once defined, you are able to locate a single node using a key. Similar to the primary key, it is optional for a node type; without any field being `NodeID`, you would need other searching criteria to locate graph nodes and there is no guarantee that the result set is singleton.

When the value of `ColumnRole` is `Edge`, the field is an adjacency list, a canonical data structure representing graph topology. An adjacency list is a list in which each element represents an edge that points to a neighbor node. The data type of such fields is always `varbinary`. In the annotation, `ColumnRole` is followed by `Reference`, specifying the type of nodes to which the edges point. In other words, the value of `Reference` is another node type created by a `CREATE TABLE` statement. In the above example, two fields `Colleagues` and `Client` are annotated as `Edge`; they represent two lists of edges pointing to node instances of `EmployeeNode` and `ClientNode` respectively.

Following `Reference` is `Attributes`, an optional annotation specifying edges' properties. An edge property is a pair of a name and a data type. In the example, the `Colleagues` field has no such an annotation. So the `Colleagues` edges have no properties. The `Clients` field, on the other hand, has the `Attributes` annotation; every edge in the list has two properties, "credit" and "note". The "credit" property is of type `int` and the "note" property is of type `string`. Currently, a node property can be of any data type SQL Server supports and an edge property is of one of the following types: `int`, `long`, `float`, `double`, and `string`.

To execute the `CREATE TABLE` statement, you invoke the `GraphViewConnection.CreateNodeTable()` method.

```
using GraphView;
……
try {
    gdb.Open(true);
    string createNodeSt = "CREATE TABLE EmployeeNode(……)";
    gdb.CreateNodeTable(createNodeSt);
    gdb.Close();
}
catch(DatabaseException e) {
    // Exception handling goes here
}
```

The invocation is transactional. Once returned, GraphView creates all necessary SQL objects such as tables and user-defined functions in the database. If an error occurred in the process, e.g., you do not have permission to create SQL objects, an exception will be thrown by SQL Server. And GraphView will not leave a footprint in the database.

Deleting node tables

To delete a node table, you issue the `DROP TABLE` statement:

```
using GraphView;
……
```

```
try {
    gdb.Open(true);
    gdb.CreateNodeTable("DROP TABLE EmployeeNode");
    gdb.Close();
}
catch(DatabaseException e) {
    // Exception handling goes here
}
```

The `DROP TABLE` statement will only be successfully executed, if no edges in the graph point to any node instance in this node table. This constraint ensures that the graph is always sound and no edge points to a "phantom" node that does not exist. If there are remaining incoming edges, you need to use the `DELETE EDGE` statement (which will be introduced later) to explicitly remove them before dropping the node table.

Query Language

Any database needs a query language for users to query and manipulate data. GraphView's query language inherits SQL syntax and extends it to accommodate graphs. The query language is highly expressive: the query languages of all existing native graph stores can be expressed with it. In this chapter, we introduce the syntax and semantics of the query language of GraphView.

`SELECT` statement

You use the `SELECT` statement to query data in SQL databases. GraphView extends the `SELECT` statement to allow you to query graphs. At the core of this extension is the introduction of the `MATCH` clause, a clause specifying one or more paths that form a graph pattern to be matched against the graph in the database.

The conventional `SELECT` statement in SQL consists of `SELECT`, `FROM` and `WHERE`, with peripheral clauses such as `GROUP-BY`, `ORDER-BY` and `HAVING`. The `FROM` clause specifies a list of table references, each of which defines a table variable and binds tuples in the table to the variable. The result set is the Cartesian product of the binding tuples of the variables, with each combination satisfying the condition in the `WHERE` clause.

The `SELECT` statement in GraphView inherits a similar semantics, with the `FROM` clause specifying a list of node table references, each defining a variable and binding graph nodes in a node table to that variable. The `WHERE` clause defines conditions the binding nodes are expected to satisfy.

The extension of the `SELECT` statement in GraphView is the `MATCH` clause between `FROM` and `WHERE`. The `MATCH` clause specifies one or more path expressions. Graph nodes bound to the variables satisfy the query, if they are connected through the specified paths.

An atomic component of a path expression is an edge that connects two node variables. An edge is in a triple of <source, edge, sink>. A path expression is a sequence of such triples, with the sink being the source of the following edge. The syntax of path expressions is shown as follows:

```
<match_path> ::=
( node_table_alias -
'[' edge_name [ AS edge_alias ] ']' -> ) [ , … n ]
node_name_alias
```

In the syntax, `node_table_alias` is a node table alias defined in the `FROM` clause. The `edge_name` is a column from the node table of

`node_table_alias`. Only if this column is specified as an adjacency-list column in the `CREATE TABLE` statement is the path expression valid.

Below is an example query matching a triangle over the employee-client graph:

```
SELECT En1.Name, En2.Name, Cn.Name
FROM EmployeeNode AS En1, EmployeeNode AS En2, ClientNode AS Cn
MATCH En1-[Colleagues AS Cg]->En2-[Clients AS C1]->Cn,
      En1-[Clients AS C2]->Cn
WHERE En1.Name = 'Jane Doe' AND C1.credit > 1000
```

The `FROM` clause defines three table aliases (or node variables), `En1`, `En2` and `Cn`. The `MATCH` clause, highlighted in bold font, includes two path expressions connecting the three variables: two employee nodes are connected through a `Colleagues` edge; the two employee nodes are both connected to a client node through two `Clients` edges.

All edges in the paths have aliases, i.e., `Cg`, `C1` and `C2`. Edge aliases can be referenced later to query the edge's properties, e.g., the predicate `C1.credit > 1000` in the `WHERE` clause. Following the convention of table aliases, when an edge has no explicit alias, its alias is the same as the edge name. Two edges cannot have the same alias; otherwise, an exception will be thrown during parsing.

In addition to the `MATCH` clause extension, the `SELECT` statement in GraphView inherits most operators and functions that are supported in the `SELECT` statement in SQL Server. Below is a list of operators/functions that are supported:

- Arithmetic operators.
- Boolean comparisons, such as >, <, and =.
- `IS NULL` or `IS NOT NULL`.
- `NOT`.
- `BETWEEN … AND …`
- `EXISTS` predicate
- `IN` predicate.
- `LIKE` predicate.
- Functions supported by SQL Server.

Nesting

The SELECT statement with the MATCH clause forms the building block of a SELECT query in GraphView. Similar to the SELECT statement in SQL, this building block can be nested inside another block, enabling nested queries, an important feature for expressive query languages.

A canonical example of nested queries is the use of EXISTS or IN predicates in the WHERE clause. The sub-query of an EXISTS/IN predicate establishes a new query context, and allows you to perform semi-joins with the outer SELECT query.

```
SELECT En1.Name, En2.Name
FROM EmployeeNode AS En1, EmployeeNode AS En2,
MATCH En1-[Colleagues AS Cg]->En2
WHERE EXISTS (
    SELECT
    FROM ClientNode AS Cn
    MATCH En1-[Clients AS C2]->Cn,En2-[Clients AS C1]->Cn
)
```

The above query finds two employee nodes who are connected as colleagues. The EXSITS predicate further specifies connections between the two employee nodes: they are both connected to a client node. Note the semantic difference from the prior example query: the prior query returns all client nodes connecting two employee nodes, whereas this query disregards specific client nodes, but only guarantees that there is at least one client shared by two employees.

Interoperability

A unique advantage of GraphView's language is its interoperability with SQL. In the FROM clause of GraphView's SELECT statement, you can also include regular tables. Once defined, regular tables can be used freely in the SELECT and WHERE clauses, including joining with node tables. They, however, cannot appear in the MATCH clause. An exception will be thrown if the MATCH clause contains regular tables.

```
SELECT En1.Name, En2.Name, Cn.Name
FROM EmployeeNode AS En1, ClientNode AS Cn, Region as R
MATCH En1-[Clients]->Cn
WHERE R.Name = 'NY' AND R.RegionID = Cn.RegionID
```

This is a query exemplifying interoperability between graphs and relational data. Region is a regular table, describing region information not present in the graph. In the query, it is defined as a regular table with two other node tables, and joined with ClientNode to filter graph patterns.

In an extreme case, when all tables in the FROM clause are regular tables, the SELECT statement reduces to an old SQL statement. GraphView does nothing for these queries, but directly passes them to SQL Server to execute.

Inserting/deleting nodes

In GraphView, you insert and delete nodes through `INSERT NODE` and `DELETE NODE` statements. They are almost identical to `INSERT`, `DELETE` statements in SQL, except that (1) the column-list specification is mandatory when inserting node properties, and (2) all `SELECT` sub-queries are now replaced by the extended `SELECT` statement with the `MATCH` clause.

```
INSERT NODE INTO ClientNode (name, regionID) VALUES ('Jane', 3)
```

This statement inserts a client node to the graph. The node's name is Jane and the region ID is 3.

This property values of the to-be-inserted node can also come from a `SELECT` statement:

```
INSERT NODE INTO ClientNode (name, region)
SELECT customer_name, regionID
FROM SaleRecord
WHERE Time = '01/01/2015'
```

In this example, node properties are constructed by a `SELECT` statement. And this statement is not a GraphView `SELECT`, but a SQL `SELECT`.

To delete graph nodes, you use the `WHERE` clause to define the criteria on which nodes to be deleted.

```
DELETE NODE FROM ClientNode
WHERE sales < 100
```

This query deletes client nodes whose sales are smaller than a certain amount.

The following query uses an `EXISTS` predicate to locate nodes by graph topology: the subquery identifies employee nodes that do not connect to any client nodes.

```
DELETE NODE FROM EmployeeNode
WHERE NOT EXISTS (
    SELECT *
    FROM ClientNode AS c
    MATCH EmployeeNode-[Clients]->c
)
```

For `DELETE NODE` statements, only nodes with no incoming edges are deleted. This is because deleting such nodes silently will leave "phantom" edges that do not point to any valid graph nodes, thereby resulting in a corrupted graph. Therefore, when a node has at least one edge pointing to it, it will be skipped during deletion. You need to delete all edges pointing a node using the `DELETE EDGE` statement, before you can delete the node safely.

Inserting/deleting edges

You manipulate edges through `INSERT EDGE` and `DELETE EDGE` statements. Unlike `INSERT/DELETE NODE` statements that are very similar to their counterparts in SQL, edge-related statements differ in several aspects: first, the operation's target is not a table any more, but a column, a column that is annotated as an adjacency list. Second, an edge specification includes edge properties as well as its source and sink nodes. The source and sink nodes must exist in the graph and are bound to node variables.

The syntax of the `INSERT EDGE` statement is shown as follows:

```
<insert_edge> ::=
    INSERT EDGE INTO node_table_name.edge_name
    SELECT source_alias, sink_alias [, edge_property [, … n ] ]
    FROM ( node_table [ AS alias ] ) [, … n ]
    [ MATCH match_path [, … n ] ]
    [ WHERE condition ]
```

Note the element list in the `SELECT` clause. The list starts with two table aliases (or node variables) defined the `FROM` clause. The two variables are bound to source and sink nodes, respectively. The Cartesian product of their binding nodes form a list of edges to be inserted to the adjacency list `edge_name`. Following the two variables are edges' properties to be inserted, if there are any.

```
INSERT EDGE INTO EmployeeNode.Clients
SELECT e, c, 99, 'referred by Alan'
FROM EmployeeNode AS e, ClientNode AS c
WHERE e.WorkId = 156 AND c.ClientId = 22
```

This is an example query inserting a `Clients` edge between an employee node and a client node. Assuming `WorkId` and `ClientId` are primary keys of the two node tables, the two predicates in the `WHERE` clause uniquely identify two nodes. The edge between them is to be inserted into the adjacency list `Clients`. The `SELECT` clause also specifies the properties of the to-be-inserted edge: `99` (as an integer) and `'referred by Alan'` (as a string).

```
INSERT EDGE INTO EmployeeNode.Colleagues
SELECT e1, e3
FROM EmployeeNode AS e1, EmployeeNode AS e2, EmployeeNode AS e3
MATCH e1-[Colleagues AS c1]->e2-[Colleagues AS c2]->e3
```

This is another example inserting multiple edges at a time. The source and sink nodes of to-be-inserted edges are both bound to employee nodes, i.e., `e1` and `e3`, whose relationships are further defined by the `MATCH` clause. Overall, the query inserts new `Colleagues` edges as shortcuts of 2-hop paths.

The `DELETE EDGE` statement starts by specifying the adjacency list from which edges are deleted. In the specification, source, edge and sink are all given aliases, by which you specify matching criteria of the edges to be deleted.

```
DELETE EDGE [e1]-[Colleagues AS c]->[e2]
From EmployeeNode AS e1, EmployeeNode AS e2
```

```
WHERE e1.WorkId = 73 AND e2.WorkId = 16
```

Query Execution

In GraphView, you execute a GraphView query or script through `GraphViewCommand`. `GraphViewCommand` provides similar interfaces as `SqlCommand`. In particular, it provides two methods, `ExecuteReader()` and `ExecuteNonReader()`, to execute commands and collect results. The returned results are retrieved using `SqlDataReader`, the same data reader used to retrieve SQL query results.

```
using System.Data.SqlClient;
using GraphView;
……
try {
    gdb.Open(true);
    string query = "SELECT … FROM … WHERE …";
    GraphViewCommand gcmd = new GraphViewCommand(gdb, query);
    DataReader dataReader = gcmd.ExecuteReader();
    while (dataReader.Read()) {
        // Retrieve results through DataReader
    }
    dataReader.Close();
    gdb.Close();
}
catch(Exception e) {
    // Exception handling goes here
}
```

To execute a data manipulation statement, you invoke the statement through `ExecuteNonReader()`.

```
using GraphView;
……
try {
    gdb.Open(true);
    string nodeSt = "INSERT NODE INTO …";
    GraphViewCommand cmd = new GraphViewCommand(gdb, nodeSt);
    cmd.ExecuteNonReader(selectQuery);
    gdb.Close();
}
catch(Exception e) {
    // Exception handling goes here
}
```

Stored procedures

A stored procedure in a SQL database is a group of SQL statements pre-compiled and stored in the server. Using stored procedures has many benefits such as reduced server-client network traffic and query compilation cost. It is particularly beneficial

to transactional workloads, as one query/script is only translated once and then invoked repeatedly thereafter.

GraphView too provides stored procedures. A stored procedure in GraphView is created by the `CREATE PROCEDURE` statement. It is the same as its SQL counterpart, except that `INSERT/DELETE` statements are replaced with `INSERT/DELETE NODE` and/or `INSERT/DELETE EDGE` statements, and `SELECT` statements are replaced by GraphView's extended `SELECT` statements.

The following example uses `GraphViewConnection.CreateProcedure()` to create a stored procedure:

```
using GraphView;
……
try {
    gdb.Open(true);
    string sp_script = "
      CREATE PROCEDURE graphSp1
        @input1 int,
        @input2 varchar
      AS
      SELECT *
      FROM EmployeeNode AS e, ClientNode AS c
      MATCH [e1]-[Clients AS edge]->[c]
      WHERE edge.credit = @input AND e.Name = @input2";
    gdb.CreateProcedure(sp_script);
    gdb.Close();
}
catch(Exception e) {
    // Exception handling goes here
}
```

The following is another example creating a graph manipulation procedure:

```
using GraphView;
……
try {
    gdb.Open(true);
    string sp_script = "
      CREATE PROCEDURE graphSp2
        @input1 varchar,
        @input2 varchar
      AS
      BEGIN
        BEGIN TRAN;
        INSERT NODE INTO ClientNode(Name) VALUES (@input1);
        INSERT NODE INTO ClientNode(Name) VALUES (@input2);
        COMMIT TRAN;
      END";
    gdb.CreateProcedure(sp_script);
    gdb.Close();
}
```

```
catch(Exception e) {
    // Exception handling goes here
}
```

After a procedure is created, you invoke it using `GraphViewCommand`, a similar practice as invoking procedures using `SqlCommand`.

```
using GraphView;
……
try {
    gdb.Open(true);
    using (var cmd = gdb.CreateCommand()) {
      cmd.CommandType = CommandType.StoredProcedure;
      cmd.CommandText = "graphSp2";
      cmd.Parameters.Add("@input1", SqlDbType.VarChar, 32);
      cmd.Parameters.Add("@input2", SqlDbType.VarChar, 32);
      cmd.Parameters["@input1"] = "Alice";
      cmd.Parameters["@input2"] = "Bob";
      cmd.ExecuteNonQuery();
    }
}
catch(Exception e) {
    // Exception handling goes here
}
```

Maintenance

A database administrator has many maintenance tasks. In addition to conventional tasks that apply to SQL databases, such as designing backup/restore stragedy, managing data and log file sizes, GraphView has specialized tasks that need to be taken care of.

Node table statistics

Node tables contain graph nodes and edges. To be able to execute graph queries efficiently, it is important to maintain graph statistics up to date, so that the query optimizer can choose the right execution plan. In the current release, graph statistics are not populated/updated automatically. It is advised that you periodically update the statistics by calling the provided API after major data changes.

```
using GraphView;
……
try {
    gdb.Open(true);
    gdb.UpdateTableStatistics("dbo", "EmployeeNode");
    gdb.Close();
}
catch(DatabaseException e) {
    // Exception handling goes here
}
```

The first input of `UpdateTableStatistics` is the schema name. The default value is "dbo". The second input is the name of the node table.

Adjacency list re-organization

GraphView represents an adjacency list as a table column. When the list is updated, i.e., edges are inserted and/or deleted, GraphView employs a lazy update strategy that defers the updates and logs the "delta" separately. This technique reduces writes to data and log files, and therefore improve update throughput. But it creates obsolete edges that need to be recycled later. As part of maintenance, you should invoke `GraphViewConnection.MergeDeta()` periodically when the system is undergoing a low volume of workload:

```
 using GraphView;
……
try {
    gdb.Open(true);
    gdb.MergeDelta("dbo", "EmployeeNode");
    gdb.Close();
}
```

```
catch(DatabaseException e) {
    // Exception handling goes here
}
```

The first input of the method is the schema name. The default value is “dbo”. The second input is the name of the node table. This method will recycle obsolete edges in all adjacency lists of the input node table.

When updates only apply to certain adjacency lists and you want to re-organize affected lists, you use `GraphViewConnection.MergeDeta(string, string, string[])`, where the last input is an array of adjacency-list columns you want to re-organize.