Getting Started with SQL Server Graph

The concept of storing data in a graph structure is not new. Generally speaking, the data in graph databases is not stored in a relational structure, so storing graph data in SQL Server therefore required careful planning and execution. The SQL Server relational engine supports graph data as of SQL Server 2017 and this support represents the first steps of a full graph functionality implementation in the product.

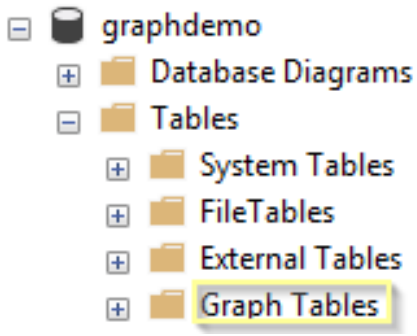
A graph is a way of modelling data with nodes (aka vertices) and edges. The nodes generally represent entities or “things” (i.e. people, products, places, customers) and the edges represent relationships between entities (i.e. lives in, works for, purchased at). Properties are supported on both nodes and edges and are a way of capturing additional information about the entities and relationships being modelled.

There are a variety of graph database implementations available on the market and Microsoft has two of them; Cosmos DB Gremlin API and SQL Server Graph. SQL Server Graph is available in the 2017 on-premise product version and in Azure SQL Database. In Cosmos DB, the underlying storage behind the graph is a document store (technically; atom-record-sequence), while in SQL Server the data is stored in rows of special relational tables. Cosmos DB refers to nodes as vertices while in SQL Server they are nodes. Both products use edges to describe the relationships between nodes.

SQL Server Implementation

The “special” nature of graph relational table storage consists of the addition of several system defined and managed columns in the graph tables. Beyond this, the tables are regular relational tables and therefore can be indexed with clustered, non-clustered and even columnstore indexes. It is possible to have multiple node types and multiple edge types in single tables, but generally you would separate them and create additional columns in each table for the node and edge properties specific to each type.

Graph tables have their own folder in the SSMS Tables tree;

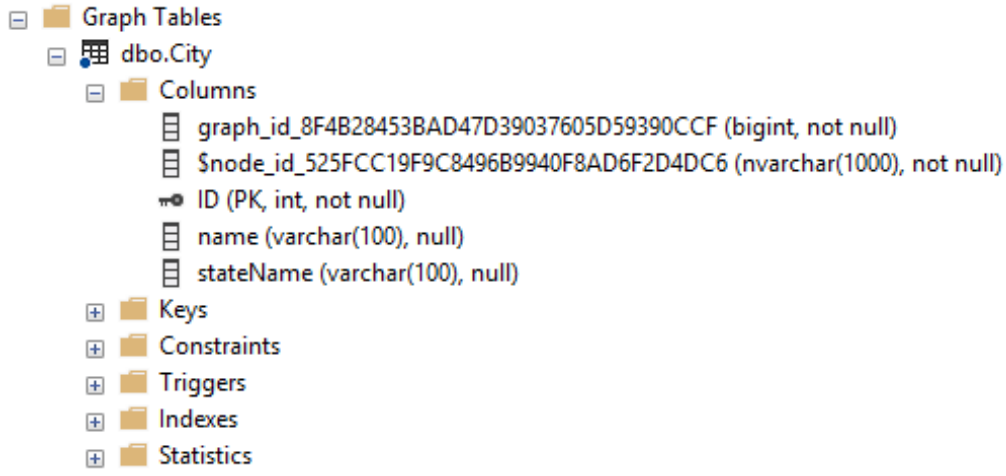


To create a node table you just need to add “AS NODE” to a create table statement. To create an edge table, similarly, just requires “AS EDGE” on a create table statement. Below is a script that creates a small sample graph structure with City, Person and Restaurant nodes and friendof, likes, livesIn and locatedIn edges.

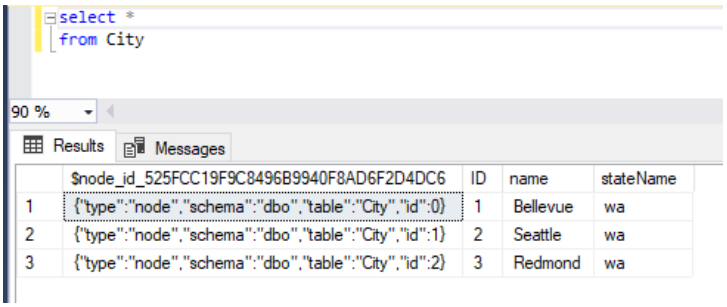
```
CREATE TABLE dbo.City
(
    ID int NOT NULL PRIMARY KEY,
    name varchar(100) NULL,
    stateName varchar(100) NULL,
) AS NODE
GO
CREATE TABLE dbo.Person
(
    ID int NOT NULL PRIMARY KEY,
    name varchar(100) NULL,
) AS NODE
GO
CREATE TABLE dbo.Restaurant
(
    ID int NOT NULL PRIMARY KEY,
    name varchar(100) NULL,
    city varchar(100) NULL
) AS NODE
GO
CREATE TABLE dbo.friendOf AS EDGE
GO
CREATE TABLE dbo.likes
(
    rating int NULL
) AS EDGE
GO
CREATE TABLE dbo.livesIn AS EDGE
GO
CREATE TABLE dbo.locatedIn AS EDGE
GO
```

As notes, we could have combined the two edge tables into a single table, since they don’t have edge specific properties, but separation makes the model more understandable.

Node tables have a system generated $node_id column. The $node_id column is a pseudo column that you can use in queries, but the actual table column name has a string of hex digits after it.



The node tables also have a graph_id column, similarly followed by a string of hex digits, but this column is not selectable and does not show up in a select * from the table.



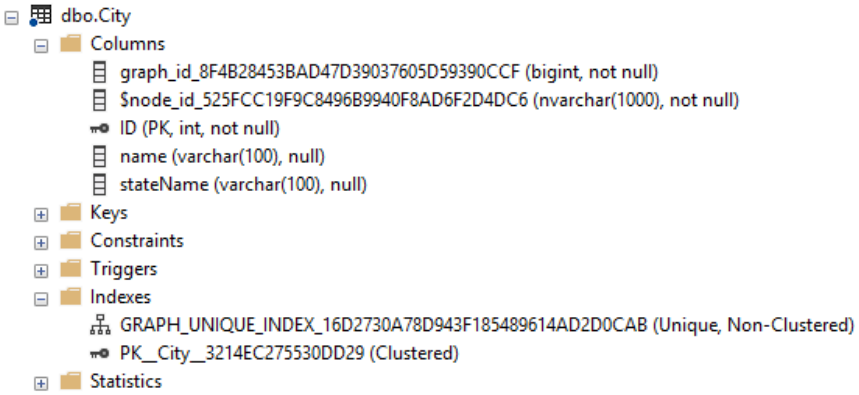
```
select *
from City
```

90 %

Results | Messages

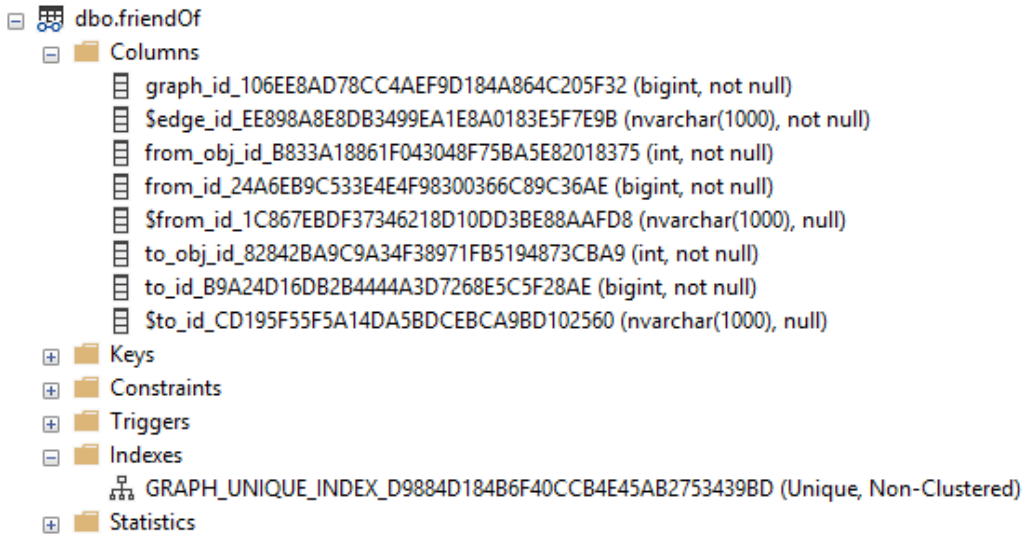
	$node_id_525FCC19F9C8496B9940F8AD6F2D4DC6	ID	name	stateName
1	{"type":"node","schema":"dbo","table":"City","id":0}	1	Bellevue	wa
2	{"type":"node","schema":"dbo","table":"City","id":1}	2	Seattle	wa
3	{"type":"node","schema":"dbo","table":"City","id":2}	3	Redmond	wa

When the node table is created, a unique index on the $node_id column is created. During loading, you may want to disable the unique index temporarily for performance reasons (remember to re-enable it).

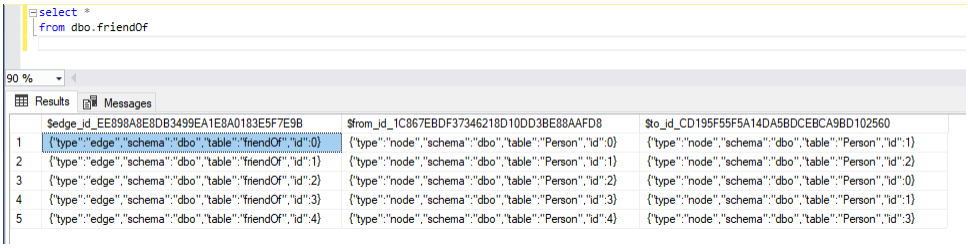


Edge tables have an $edge_id column, $from_id and $to_id columns (again, these are all pseudo column references, the actual columns have hex strings after them). There are also some other columns in the

underlying table (i.e. graph_id, from_obj_id, from_id etc.), but these are not accessible and are system maintained. The key columns for most uses are the $from_id and $to_id columns.



In edge tables, some of the columns are also hidden from a select *, but you can see them in SSMS.



```
select *
from dbo.friendOf
```

90 %

Results | Messages

	$edge_id_EE898A8E8DB3499EA1E8A0183E5F7E9B	$from_id_1C867EBDF37346218D10DD3BE88AAFD8	$to_id_CD195F55F5A14DA5BDCEBCA9BD102560
1	{"type":"edge","schema":"dbo","table":"friendOf","id":0}	{"type":"node","schema":"dbo","table":"Person","id":0}	{"type":"node","schema":"dbo","table":"Person","id":1}
2	{"type":"edge","schema":"dbo","table":"friendOf","id":1}	{"type":"node","schema":"dbo","table":"Person","id":1}	{"type":"node","schema":"dbo","table":"Person","id":2}
3	{"type":"edge","schema":"dbo","table":"friendOf","id":2}	{"type":"node","schema":"dbo","table":"Person","id":2}	{"type":"node","schema":"dbo","table":"Person","id":0}
4	{"type":"edge","schema":"dbo","table":"friendOf","id":3}	{"type":"node","schema":"dbo","table":"Person","id":3}	{"type":"node","schema":"dbo","table":"Person","id":1}
5	{"type":"edge","schema":"dbo","table":"friendOf","id":4}	{"type":"node","schema":"dbo","table":"Person","id":4}	{"type":"node","schema":"dbo","table":"Person","id":3}

Performance Tuning

To improve query performance, you will want indexes on the $node_id for all node tables and for edge tables an index on the $from_id and the $to_id columns (i.e. the actual columns). Based on some non-exhaustive performance testing, creating single indexes on the $from_id and $to_id gave approximately the same, if not slightly better performance than composite indexes on ($from_id, $to_id) and ($to_id, $from_id).

In one of our engagements, the customer was mostly interested in taking their dataset and exploring it with Graph by doing analytics queries on it. In this case, the most significant performance boost was seen using a clustered column store index. Below is a table of several analytics type queries that we did for this customer engagement. As part of the engagement, we experimented with the performance of SQL Graph on a few different Service Level Objectives (SLOs) of Azure SQL DB and on a high performance laptop. The bottom part of the table shows the significant performance boost of adding a clustered columnstore index.

SLO	Query1	Query2	Query3	Query4	Query5	Query6	Query7
S4	446	2292	753	34	80	3	5+ hours
S9	59	77	88	4	9	0	3:58:23
P1	460	2703	1090	89	154	6	N/R
P4	168	734	882	16	38	1	4+ hours
P11	36	88	80	5	10	0	4+ hours
Local SSD	152	560	724	42	20	3	593 - 1080

DTU	Cost $/mo
200	300
1600	2400
125	465
500	1860
1750	7000

With CCI	Query1	Query2	Query3	Query4	Query5	Query6	Query7
S4	209	568	239	5	11	0	240
P4	16	153	37	2	5	0	38
Local SSD	13	83	36	2	2	0	37

DTU	Cost $/mo
200	300
500	1860

Note that for OLTP type workloads, CCI indexes may not help performance (or may even negatively impact performance). If you decide to use columnstore indexes, you will need to keep an eye on index quality and may also require non-clustered indexes for performance. Fortunately, as a table, you have the flexibility and power to tune indexes as required.

Inserting Data

The graph tables are regular relational tables, so inserting data into them can be done with a regular insert SQL statement. When inserting into a node table, you only set the columns representing the properties of the node, you do not set the $node_id column. When inserting into an edge table, you set the $from_id and $to_id to $node_id references from node tables. You can easily use Bulk Copy API to populate tables, but you need to refer to the actual column name and we needed to explicitly map the columns to get the API to function correctly.

Sample Node table inserts

```
INSERT INTO Person VALUES (1,'John');
INSERT INTO Person VALUES (2,'Mary');
INSERT INTO Person VALUES (3,'Alice');
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');
```

Sample Edge table inserts

```
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);
```

(Since the customer schema and queries are confidential, for illustration purposes, the sample schema and sample queries above are borrowed from a complete sample script that you can find here).

Querying Data

The extensions to TSQL consist of a new "WHERE" predicate "MATCH". You always have the alternative to explicitly join the node and edge tables using normal join syntax, but the ASCII-art MATCH predicate can be used to easily traverse a graph in a join free manner;

```
-- 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);
```

Note that you can combine match clauses with ANDs and you can also use regular where clause filters in combination with match filters. There is a strong argument there is nothing you can do with this syntax that you can't do with regular TSQL, but remember that this is V1.

(Again these queries are borrowed for illustration purposed from here).

For more information – check out the links below;

Documentation

SQL Graph overview - https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017

Architecture - https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-architecture?view=sql-server-2017

Match in TSQL - https://docs.microsoft.com/en-us/sql/t-sql/queries/match-sql-graph?view=sql-server-2017

Potentially Useful Intros

https://stephanefrechette.com/sql-graph-sql-server-2017/#.W3w2AUxFz5s

https://www.sqlshack.com/introduction-sql-server-2017-graph-database/

Thanks to Shreya Verma, Senior Program Manager on the SQL Graph team for doing a review on this paper.

Please feel free to reach out with any questions or comments.

Mitch van Huuksloot, P.Eng., MBA
Solution Architect
Data Migration/Modernization Jumpstart Engineering Team
Data & AI CTO Office | Enterprise Services | Microsoft Corporation

Feedback and suggestions

If you have feedback or suggestions for improving this data migration asset, please contact the Data Migration Jumpstart Team (askdmjfordmtools@microsoft.com). Thanks for your support!

Note: For additional information about migrating various source databases to Azure, see the Azure Database Migration Guide.