# Relational Databases vs Graph Databases

By: [Siddharth Mehta](https://www.mssqltips.com/sqlserverauthor/45/siddharth-mehta/)   |   Last Updated: 2019-07-25   |   [Comments](https://www.mssqltips.com/sqlservertip/6105/relational-databases-vs-graph-databases/#comments)   |   Related Tips: [More](https://www.mssqltips.com/sql-server-dba-resources/) > [SQL Server 2017](https://www.mssqltips.com/sql-server-tip-category/228/sql-server-2017/)

<https://www.mssqltips.com/sqlservertip/6105/relational-databases-vs-graph-databases/>

##### Problem

Relational databases are found almost in every conceivable business scenario, and SQL is arguably the de-facto standard of accessing data from database systems. With the advent of NoSQL database systems, as well as with some very successful adopters of graph like Google, Facebook, LinkedIn and others, graph has become quite popular and the database community is not that aware and open towards non-relational database management systems. With such a wide adoption of relational databases and a large community of relational database professionals, eventually one would encounter questions like what’s the difference between a relational database and a graph database, when should I consider using a graph database, etc.

In this tip, we will address questions that will help relational database developers understand the various considerations for using a graph database.

##### Solution

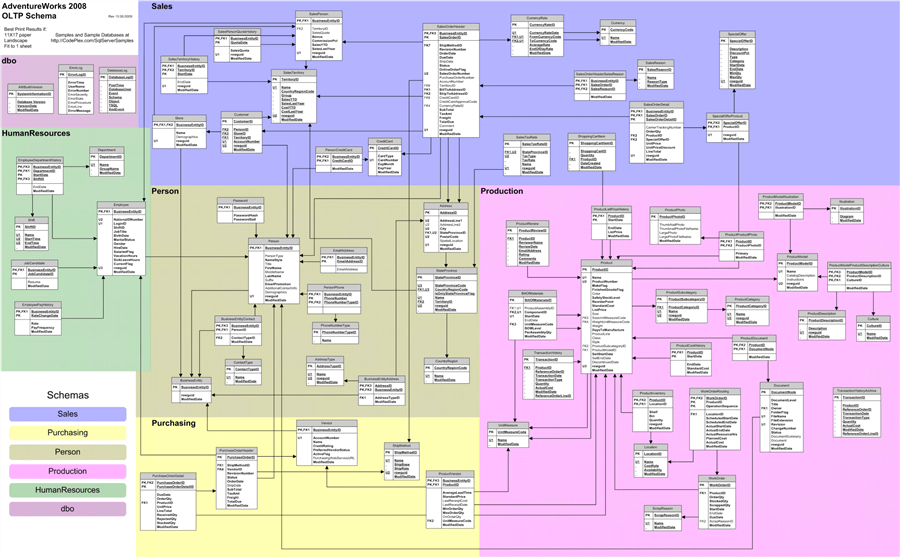
As the complexity in data and value in relationships increases, the ability of relational databases to address the data requirements decreases and use of graph databases increases, which leads to the adoption of graph databases for the right use-cases.

Data exists in various forms from simplest structures and relationships to the very complex forms. At its simplest form, data can be expressed in the form of key-value pairs. Key value databases store data in terms of unique identifiers which are also known as keys, and they have corresponding values. Examples of key values are connection string, session tokens, products in an e-commerce site, etc. Considering a relatively complex form of data with increased relationships, the next logical move from key-value stores goes towards NoSQL data stores.

NoSQL data stores are of various types like document oriented, key-value, columnar, object store, XML store, etc. The data complexity handled by these data stores expands to more complex structures like JSON documents, blob objects, unstructured data, etc. But these data elements are generally not expected to have very strong and rigid relationships. The data elements are self-sufficient and grouped under a common logical space like an index or a database without necessarily having to have a point to point relationship with other data elements.

Once the data complexity increases to complex schemas, stringent constraints on the data as well as transactions, the relationship of one entity with another, and the need to control as well as manage those relationship in a highly controlled manner, relational databases come into picture. Relational databases have been generally seen as the norm of database management unless the use-case requires out-of-the-ordinary characteristics from a database management system for structured data.

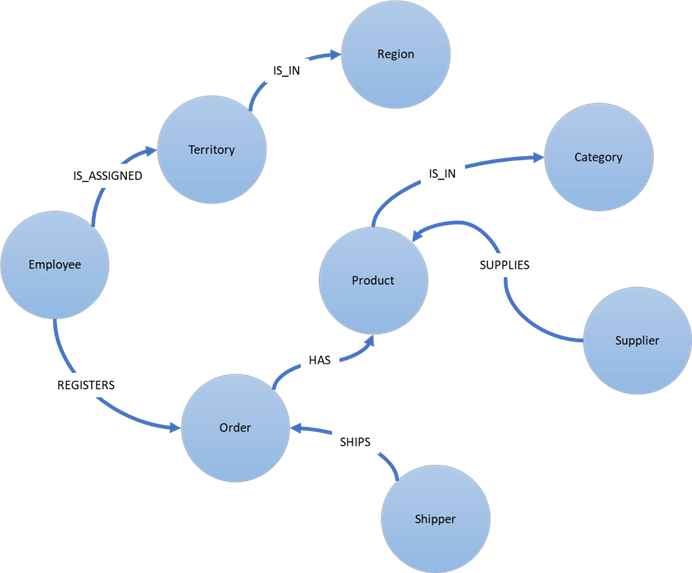
A relational database typically stores data in normalized schemas which is formed of a set of normalized as well as de-normalized tables organized typically under databases and schemas. These tables have fixed attributes also known as fields, which have features like data-types, constraints, etc. If the consumers have a rapidly changing need compared to the structure created in the relational databases, it requires a schema or structural change in the database to suit the needs of consumption. Else it would require a high level of overhead to modulate the data from the fixed structure of the relational databases to the needs of consumers.



The diagram shown above is a logical database diagram of the Northwind sample database that shows how tables are interconnected with primary and foreign keys. These relationships between normalized tables are evaluated at query time by joining attributes from one or more tables with another which is typically known as table JOINs. These are performance intensive operations, and the larger the scale of the data the harder it becomes to perform these joins to extract the desired data using the right relationships.

Entities can have one-to-one, one-to-many as well as many-to-many relationships. These relationships can be direct between two tables, or indirect as well. For example, a business can have departments, which can have employees. To find employees that belong to a business, either data would have to be joined through departments, or the data would have to be denormalized in a single table, which may cause loss of relationships. The more complex the data grows, the more one would normalize entities in a relational database and the representation of relationships becomes more convoluted.

At this stage, when the complexity or variability of data is extreme and the value or utilization of relationships between entities is of prime importance, graph databases becomes a natural choice. Some of the typical examples of use-cases for graph data models are fraud detection, supply-chain, network related data, etc. One of the most easily understood example of a graph is a social network graph, where people are entities and the associations between them are relationships. A subset of the relationships in the Northwind database can be represented as shown in the below diagram.



Let us understand the key characteristics of a graph.

* In a graph data model, the conceptual model becomes the actual physical model of the graph. We create entities first and then associate them with relationships, and the data is stored in the same manner unlike relational databases where data is always joined with one or more attributes.
* A graph database does not have any fixed schema, but graph can have directions in the edges, sub-graphs, weight of the edges and other such features that define relationships.
* Relationships are physically stored in the database along with actual data, which makes data retrieval much faster compared to relational databases which evaluate relationships at query time.
* Graph database reduce the amount of data required to derive insights typically in a highly connected data environment, as it does not have fixed data structure limitations like relational databases.
* A graph data model is composed of nodes and edges, where nodes are the entities and edges are relationships between those entities.
* Graph models are basically of two types – Labeled Property Graph and Resource Description Framework (RDF). A Property Graph generally has nodes and edges with unique ids, and internal structures attached to them in the form of key-value pairs. RDFs on the other hand are formed of triples also known as subject-predicate-object, which represent two nodes associated by an edge, without any internal structure.
* Gremlin is typically used to query a property graph, and SPARQL for querying an RDF graph. Cypher is another query language for graph querying.

Microsoft Azure Cosmos DB as well as Microsoft SQL Server both support hosting graph database models. Now that we understand why and when we would start using a graph database, I would highly encourage you to analyze how these databases support graph database models and the mechanism to exploit the maximum potential of these database systems for the right use-cases.

##### Next Steps

* Consider reading tips like [this](https://www.mssqltips.com/sqlservertip/4883/sql-server-2017-graph-database-example/) to start broadening your understanding about graph in SQL Server and staring your journey in the world of Graph Database Systems.

Last Updated: 2019-07-25

# Graph processing with SQL Server

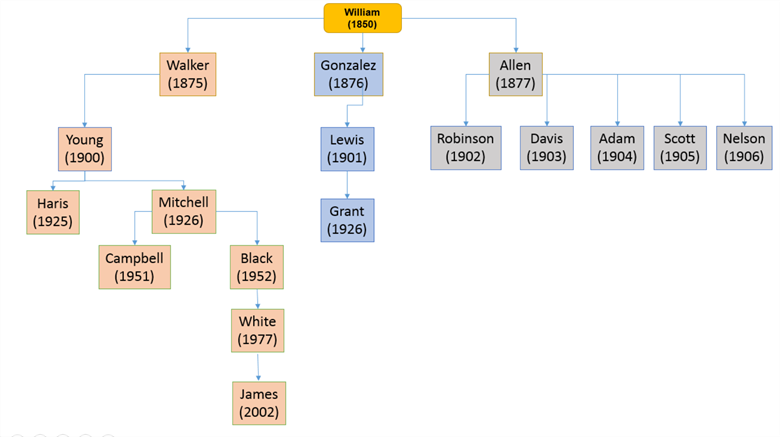
By: [Jayendra Viswanathan](https://www.mssqltips.com/sqlserverauthor/296/jayendra-viswanathan/)   |   Last Updated: 2018-06-04   |   [Comments (3)](https://www.mssqltips.com/sqlservertip/5429/graph-processing-with-sql-server/#comments)   |   Related Tips: [More](https://www.mssqltips.com/sql-server-dba-resources/) > [SQL Server 2017](https://www.mssqltips.com/sql-server-tip-category/228/sql-server-2017/)

##### Problem

I came across Graph processing in SQL Server which to me is an advanced version of using [Common Table Expressions](https://www.mssqltips.com/sql-server-tip-category/26/scripts/) (CTE). SQL Server offers graph database capabilities to model many-to-many relationships. The graph relationships are integrated into T-SQL and receive the benefits of using SQL Server as the foundations database management system. In simple words, a graph database is the combination of NODES and EDGES. NODES represent an entity, EDGES represent a relationship between two nodes. Both nodes and edges may have properties associated with them. In this tip we are going to see a hierarchical structure of a family tree. The family tree will be based on the chart below. We will use SQL Server graph functionality to store and retrieve the data.

##### Solution

Here is a family tree that we will use for this tip.



William was born in 1850 and is the father of the family. The above chart is self-explanatory with names of William's sons and the year of birth.  Let’s process the above chart with Graph processing with SQL Server.

First, we are going to create the base data for William's family. Then create a NODE table which will create nodes for each record. Create an EDGE table which will be used to match the NODEs of William's family. The EDGE table will contain the actual output. We will use MATCH SQL command to virtually create the descendent tables as NODES and EDGES to fetch records per our needs.

We can use the below SQL commands to create the base data for William's family. I created a fresh database too.

DROP DATABASE IF EXISTS FamilyDB;

GO

CREATE DATABASE FamilyDB;

GO

USE FamilyDB;

DROP TABLE IF EXISTS MyFamily;

CREATE TABLE MyFamily (

FmlyNum NUMERIC(8) not null,

Name VARCHAR(40) NOT NULL,

FmlyLink NUMERIC(8),

YOB INT,

INUM INT)

The above statements will create the base table “MyFamily”.

Let's insert the records. The top most 1st level is William who was born in 1850 and hence William is kept in the top most node and the FmlyLink column is kept as NULL. Other records have a FmlyLink code for their father. Each record is linked between the FmlyNum and FmlyLink column.

INSERT INTO MyFamily values

(10000, 'William', NULL, 1850, 1),

(140000,'ALLEN', 10000, 1877, 1),

(60000, 'ROBINSON',140000, 1902, 1),

(70000, 'DAVIS', 140000, 1903, 1),

(80000, 'ADAM', 140000, 1904, 1),

(90000, 'SCOTT', 140000, 1905, 1),

(100000,'NELSON', 140000, 1906, 1),

(20000, 'GONZALEZ',10000, 1876, 1),

(30000, 'LEWIS', 20000, 1901, 1),

(31000, 'GRANT', 30000, 1926, 1),

(40000, 'WALKER', 10000, 1875, 2),

(120000,'YOUNG', 40000, 1900, 2),

(50000, 'HARRIS', 120000, 1925, 2),

(130000,'MITCHELL',120000, 1926, 2),

(110000,'CAMPBELL',130000, 1951, 2),

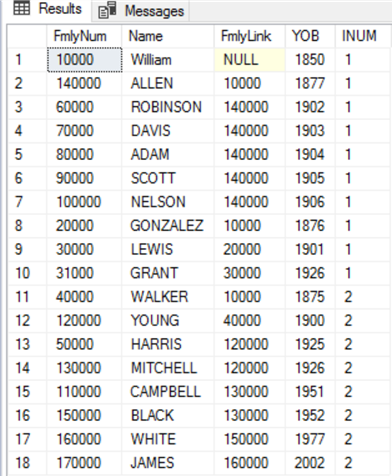
(150000,'BLACK', 130000, 1952, 2),

(160000,'WHITE', 150000, 1977, 2),

(170000,'JAMES', 160000, 2002, 2);

By running the above INSERT statements, the base data of William's family is ready.

SELECT \* FROM MyFamily



## Node Table

DROP TABLE IF EXISTS MyFmlyNode;

CREATE TABLE MyFmlyNode(

FNO Int Identity(1,1),

FmlyNum NUMERIC(8) NOT NULL,

Name VARCHAR(40),

FmlyLink NUMERIC(8),

INUM INT

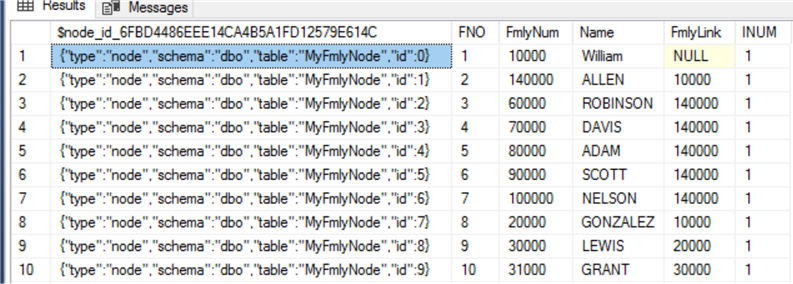
) AS NODE;

INSERT INTO MyFmlyNode(FmlyNum, NAME, FmlyLink, INUM)

SELECT FmlyNum, NAME, FmlyLink, INUM

FROM MyFamily

SELECT \* FROM MyFmlyNode



We have created a NODE table and inserted the records with $NODE\_ID as the key column.

Let's see how {"type":"node","schema":"dbo","table":"MyFmlyNode","id":0} can be explained:

* The first column is the $node\_id, this column is automatically created with the user defined columns that is created with the “AS NODE” keyword in the CREATE statement. In our case we have FNO, FmlyNum, Name and FmlyLink are user created columns. The NODE table is an entity in a graph schema. The values in the $node\_id column are automatically generated and are a combination of object\_id of that node table and an internally generated bigint value. We can also create a unique constraint or index on the $node\_id column.

## Edge Table

An EDGE table maintains the relationship between two NODES. In the below statement we will extract the matching $node\_id’s from the NODE table and insert the value in the EDGE table, so that unique graph ids are created to maintain a relationship.

DROP TABLE IF EXISTS MyFmlyEdge;

CREATE TABLE MyFmlyEdge(

FmlyNum numeric(8)

) AS EDGE

INSERT INTO MyFmlyEdge

SELECT e.$node\_id, m.$node\_id ,e.fno

FROM dbo.MyFmlyNode e

INNER JOIN dbo.MyFmlyNode m ON e.FmlyNum = m.FmlyLink;

SELECT \* FROM MyFmlyEdge

Let’s discuss the SELECT query part of the above INSERT statement. If you notice the INSERT statement into the Edge table, the two NODEs from the MyFmlyNode table are extracted based on the key column e.FmlyNum = m.FmlyLink. Now the query returns the matching $node\_id data from Node tables. The extracted $node\_id’s are inserted into the EDGE table. By INSERTING the matching NODE’s into the EDGE table, the link/connection is established between the Node and Edge tables.

Below is the output of the SELECT statement for entire William's family.



I would like to also touch base on that the above join query can also be performed by using INSERT statements into the EDGE table by using a WHERE clause from the NODE table. Here are a few sample INSERT statements for understanding how this can be done.

INSERT INTO MyFmlyEdge

VALUES ((SELECT $node\_id FROM MyFmlyNode WHERE FNO = 1),

(SELECT $node\_id FROM MyFmlyNode WHERE FNO = 2),

111);

INSERT INTO MyFmlyEdge

VALUES ((SELECT $node\_id FROM MyFmlyNode WHERE FNO = 2),

(SELECT $node\_id FROM MyFmlyNode WHERE FNO = 3),

111);

INSERT INTO MyFmlyEdge

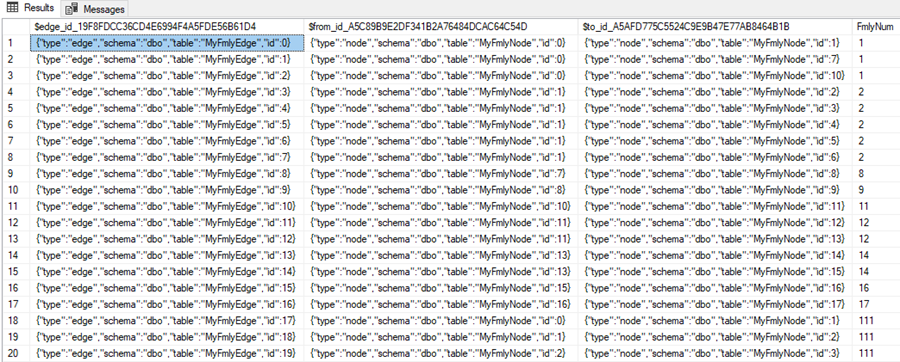
VALUES ((SELECT $node\_id FROM MyFmlyNode WHERE FNO = 3),

(SELECT $node\_id FROM MyFmlyNode WHERE FNO = 4),

111);

After running the above INSERT statements, the below query will show us the output from the EDGE table.

SELECT \* FROM MyFmlyEdge



Now the EDGE table MyFmlyEdge is loaded with matched records for William's Family.

I am going to cleanup these records we just inserted, so our results are correct for the family tree that we started with.

DELETE FROM MyFmlyEdge

WHERE FmlyNum = 111

## Query data based on the chart

Based on William’s family chart above, let's query the table and see how we can get data.

## MATCH (SQL Graph)

A match query can be used to query any SQL Graph, we can query the NODE and EDGE using the MATCH statement. Nodes can be traversed an arbitrary number of times in the same query. In our example, we will use the MATCH statement to traverse between NODES and EDGES. MATCH can be used to traverse TOP DOWN or BOTTOM UP as per the requirements.

## Second NODE query

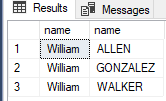
Return data for William and his sons.

SELECT MyFmlyNode1.name, MyFmlyNode2.name

FROM MyFmlyNode MyFmlyNode1, MyFmlyEdge, MyFmlyNode MyFmlyNode2

WHERE MATCH(MyFmlyNode1-(MyFmlyEdge)->MyFmlyNode2)

AND MyFmlyNode1.name = 'William';



If you notice William is the first node and the second nodes are ALLEN, GONZALEZ and WALKER who are William's sons. MATCH is used to traverse the NODES from top to bottom. To connect the two NODES we use this **->** that shows we are going left to right.

Note that the relationship could go either way as shown below.

SELECT MyFmlyNode1.name, MyFmlyNode2.name

FROM MyFmlyNode MyFmlyNode1, MyFmlyEdge, MyFmlyNode MyFmlyNode2

WHERE MATCH(MyFmlyNode1-(MyFmlyEdge)->MyFmlyNode2)

AND MyFmlyNode1.name = 'William';

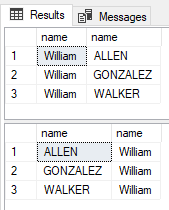
SELECT MyFmlyNode1.name, MyFmlyNode2.name

FROM MyFmlyNode MyFmlyNode1, MyFmlyEdge, MyFmlyNode MyFmlyNode2

WHERE MATCH(MyFmlyNode1<-(MyFmlyEdge)-MyFmlyNode2)

AND MyFmlyNode2.name = 'William';

The first query shows William first and then his sons and the second query shows the sons first and then William.



## Third NODE Query

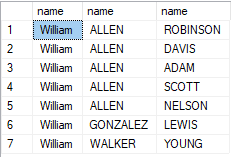
This will return data for William, his sons and any son that has sons.

SELECT MyFmlyNode1.name,MyFmlyNode2.name,MyFmlyNode3.name

FROM MyFmlyNode MyFmlyNode1, MyFmlyEdge, MyFmlyNode MyFmlyNode2, MyFmlyEdge MyFmlyEdge2, MyFmlyNode MyFmlyNode3

WHERE MATCH(MyFmlyNode1-(MyFmlyEdge)->MyFmlyNode2-(MyFmlyEdge2)->MyFmlyNode3)

AND MyFmlyNode1.name = 'William';



* If you notice William is the first node
* The second nodes are ALLEN, GONZALEZ and WALKER who are William's sons.
* The third nodes are
  + ROBINSON, DAVIS, ADAM, SCOTT, NELSON who are son's of ALLEN.
  + LEWIS is the son of GONZALEZ.
  + YOUNG is the son of WALKER

## Return Data for All Lineages

To return data for all connections from the oldest to the most recent generation we can use a query like below.

with Fmly

AS

(

SELECT r1.NAME AS TopNode,r2.NAME AS ChildNode,CAST(CONCAT(r1.NAME,'-<',r2.NAME) AS varchar(250)) AS Output,r1.$node\_id AS parentid, r2.$node\_id as bottomnode,1 as Tree

FROM MyFmlyNode r1

JOIN MyFmlyEdge e ON e.$from\_id = r1.$node\_id

JOIN MyFmlyNode r2 ON r2.$node\_id = e.$to\_id AND r1.NAME IN( 'WILLIAM')

UNION ALL

SELECT c.ChildNode,r.NAME,CAST(CONCAT(c.Output,'-<',r.NAME) AS varchar(250)),c.bottomnode,r.$node\_id,Tree + 1

FROM Fmly c

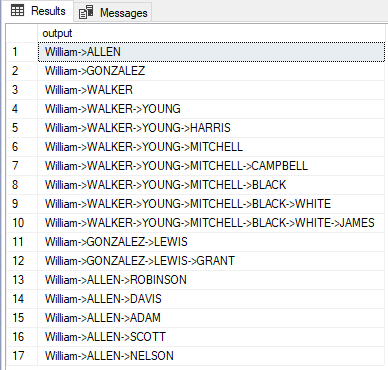
JOIN MyFmlyEdge e ON e.$from\_id = c.bottomnode

JOIN MyFmlyNode r ON r.$node\_id = e.$to\_id

)

SELECT output FROM Fmly

This shows all of the connections from left to right.



## Conclusion

We have seen how the MATCH clause can be used to fetch data from EDGE tables. I hope the tip will be useful for the SQL Server developer community.

##### Next Steps

* Check out this tip on the [SQL Server MERGE command](https://www.mssqltips.com/sqlservertip/5373/sql-server-merge-statement-usage-and-examples/)