# Graph Database

Graph Databases is a NoSQL database based on Graph Theory and it consists of objects called nodes, properties, and edges(relationships) to represent store, and search the relationships of data. Graph databases treat the relationships between nodes equally important as data itself. (1)

“A set of nodes, edges and properties to represent and store data. The relationships between data points often matter more than the individual points themselves” (Dataversity) (1)

“A database designed to treat the relationships between data as a first-class citizen in the Data Model”(Neo4J) (1)

“A graph data model consists of vertices that represent the entities in a domain, and edges that represent the relationships between these entities”(InfoWorld) (1)

**Nodes**

Each node represents an entity such as people, places, etc. You can think of nodes as records in a relational database. Every node in the graph database has at least one incoming or outgoing edge or both

**Edges**

Edges are also called as Relationships. These are the lines that connect the nodes and represent the relationship between the connecting nodes. One of the biggest differences between relational DB and graph databases is that relational databases don’t store relations between the records but graph databases store this information.

**Properties**

Properties store information related to nodes and edges.

In the below diagram, there are three nodes. Each node has its own properties. Nodes containing Id: 1 and id:2 are having three properties: id, Name and Age. Similarly, there is another node with three properties: Id, Type and Name.

The lines connecting these nodes are called edges or relationships. These relationships can be easily understood. Alice knows Bob since 2001/10/03 and Bob know Alice since 2001/10/04. Both Alice and Bob are part of the group called Chess and this relationship is shown using is\_member and Members relationships. (1)

Diagram

Description automatically generated

## When you should/shouldn’t use Graph Database?

## Cases where graph databases are NOT a good choice

* If the data is highly disconnected, and relationships between the data don’t matter (for example, customer transaction data), you don't need a graph database.
* If the requirement is to just store the data and you are using only simple queries, a graph database is not needed.
* If your data structure is fixed and consistent then there is no need of going for graph databases as graph databases are best suited for storing all types of data and changing business needs.
* Graph databases are not suited if you are querying for bulk data scans as they are not optimized for such operations.
* If the requirement is to store and retrieve entity properties that contain extremely large values (such as BLOBs, CLOBs, etc), then a graph database is not an ideal solution.

## Cases where graph databases are a good choice

* If you are dealing with highly connected data such as Facebook friend connections etc. then you should go for the database as graph databases are purposely built to handle the highly connected data.

## Graph Database Use cases

Graph Databases are used in almost all industries and are being used by thousands of companies around the world. Here are some of the use cases where Graph Databases find their applications -

* Fraud or Anomaly Detection
* Real-Time Recommendations
* Graph-Based Search
* Social Networks
* Machine Learning
* Identity and Access Management, etc.

## Graph Databases

As per the [db-engines.com](https://db-engines.com/en/ranking/graph+dbms), Neo4j is the market leader followed by Microsoft Azure Cosmos DB and ArangoDB. (1)

Chart, line chart

Description automatically generated

A picture containing graphical user interface

Description automatically generated

# **Pros and Cons of Graph Databases**

## Pros

* Graph databases query in real-time for highly connected data. As we have just gone through they are designed mainly for highly connected data.
* The performance of graph databases is astounding with small or big data and it makes them an ideal solution for real-time big data analytics queries where data size grows rapidly.
* With graph databases, you can manage constantly changing business needs and object types which are not easy in relational databases, etc.

## Cons

* Graph databases don’t have uniform query language. For example, Neo4j uses Cipher, and Cosmos DB uses SQL as a JSON query language, and so on. So, if you want to switch between the graph databases you will have to learn the query language of that graph database.
* Graph databases don’t do well for aggregating data so they should not be used in business intelligence.
* Graph databases don’t scale-out well, etc.

**GraphQL and Graph Databases**

Until recently, my understanding was that apart from 5 latin characters GraphQL has got little or nothing to do with Graph Databases . But things are changing. Graph Database vendors have started bringing these two technologies together. (3)

GraphQL is a contemporary API interface standard used as an alternative to REST. GraphQL was originally developed by Facebook. It provides a structured query language for querying data, along with a runtime implementation to invoke those queries upon your data. In pretty much all implementations it uses JSON as its serialization format. (3)

REST is a lightweight low-level API standard for performing uniform and a predefined set of stateless operations on data aligned with the HTTP verbs. With REST you have to design and  build your API interface contract for that set of operations - and while of course there are common patterns in REST APIs, this is very much arbitrary. REST makes no assumptions about the high level API contract or the underlying data model. The developer defines both. With GraphQL you only define the data model (the GraphQL schema), exposing to the GraphQL runtime and your consumers what the shape of the data is.

With a GraphQL query you can request only the minimum data required to fulfil your current need. Of course you can do this with a REST API too - in fact it is a common pattern with JSON REST APIs to filter response data by JSON fields using dot notation. But of course you need to build this yourself.

GraphQL does a lot more for you to boot, as well as out of the box query patterns for mutating data, it also supports subscriptions (pub/sub style) - so your GraphQL client can subscribe to server side events, and have data pushed to it.

## Caching and performance

Amazon’s AWS AppSync is a fully managed GraphQL service built upon the Apollo technologies - and can source data out of the box from other services such as DynamoDB, ElasticSearch, REST APIs etc. With fully managed caching and real-time data subscriptions, it really does make your developer life easy.

On the surface it still appears GraphQL has little to do with Graph Databases. Yet increasingly we are hearing about how they should go together like bread and butter, sausage and mash. (3)

## Not all graph databases are equal

I won't go into depth about what a Graph Database is, but am going to explore some of the options available, and the relationship between Graph Databases and GraphQL.

Graph databases come in several flavor combinations. First up you have RDF graphs vs Labelled-Property graphs. The first big choice. Your primary use case is important here. Essentially if you primarily need to walk, query and analyze paths through your graph, then Labelled-Property graphs are probably the way to go. If you are more interested in the nature of the relationships between things, and creating rich semantic representations of things, then RDF graphs hit the spot. Both types have their own well-developed query language. Labelled-Property graphs are typically queried with Cypher or Gremlin (e.g. Neo4J), RDF graphs are queried with SPARQL.

While most mainstream RDF based graph databases are fully compliant to the RDF and SPARQL W3C standards, not all RDF graph databases are equal. Each has its own implementation nuances and come with their own added extras (decisions, decisions!):

* An RDF graph database may or may not support reasoning (to some degree of the RDFS and OWL standards).
* Reasoning may be at query time (backward chaining) or at write-time (forward-chaining).
* The way reasoning is implemented can dramatically impact performance.
* Add-ons and extensions for GeoSPARQL, Path analysis, Full text search, connections to other datastores etc.

**Glue-Neptune:** Glue-Neptune is a Python library for AWS Glue that helps writing data to amazon Neptune from Glue jobs.

* Get Neptune connection information from the Glue Data Catalog
* Create label and node edge ID columns in Dynamic Frames, named in accordance with the Neptune CSV bulk load format for property graphs
* Write from Dynamic Frames directly to Neptune

Source:

* <https://github.com/aws-samples/amazon-neptune-samples/tree/master/gremlin/glue-neptune>
* <https://github.com/awslabs/amazon-neptune-tools/tree/master/glue-neptune>

Pros:

* High Performance
* Easy to Use
* Managed Service in AWS
* Support for SPARQL
* Advanced security capabilities including network security through VPC and encryption at rest using AWS Key Management Services

Cons:

* Vertical Scalability is disproportionately more expensive than horizontal scalability
* Does not support Analytic query algorithms

**Neo4j and Apache Spark:**

Apache spark includes a Neo4j connector. Apache Spark is a clustered, in-memory data processing solution that scales processing of large datasets easily across many machines. It also comes with GraphX and GraphFrames two frameworks for running graph compute operations on your data.

Spark can also serve as external Graph compute solution, where you

* Export data of selected subgraphs from Neo4J to Spark,
* Compute the analytics aspects, and
* Write the results back to Neo4j
* To be used in your Neo4j operations and Cypher queries

Pros:

* It is very efficient on large datasets to support relations between the nodes
* Inserting or updating any node or relation is also very easy through the UI or a script
* Provides very good graphical representation to analyze or present a dataset
* Very good interactive UI for analysis of any dataset

Cons:

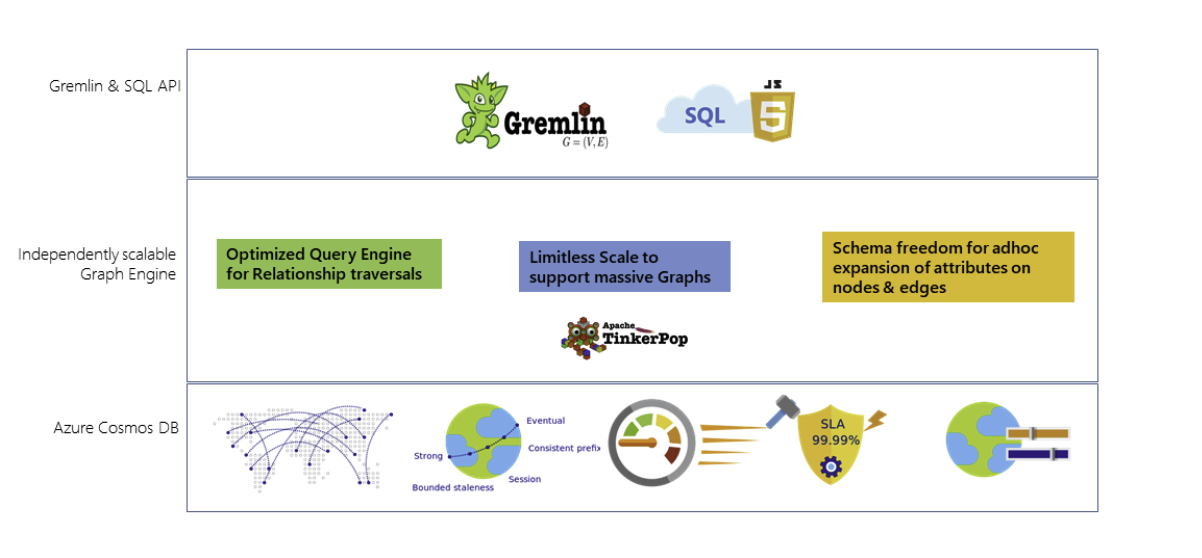
* Requires well configured server for a larger dataset
* Also graphical representation for less complex dataset is good but for complex dataset in which more than 10 relation possible graphs are not good
* Also the interactive UI for a complex dataset is little bit complex

Source:

* <https://www.trustradius.com/products/neo4j/reviews>

**Microsoft Azure Cosmos DB:**

It is the globally distributed, multi-model database service which supports document, key-value, graph via Gremlin API which is built based on Apache TinkerPop, a graph computing framework on a fully managed database service designed for any scale.



Pros:

* Elastically scalable thoughput and storage
* Multi-region replication
* Automatic indexing

Cons:

* Not ACID compliant
* Pricing

Source:

* <https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction>
* <https://stackshare.io/stackups/azure-cosmos-db-vs-neo4j>
* <https://www.trustradius.com/products/azure-cosmos-db/reviews?qs=pros-and-cons>

**Grakn:**

Grakn.AI is a database for AI. It is a deductive database in the form of knowledge graph that uses machine reasoning to simplify data processing challenges for AI applications. GRAKN.AI is composed of two parts: Grakn (the storage) and Graql (the language).

* **Grakn** is a database in the form of a knowledge graph that uses an intuitive ontology to model extremely complex datasets. It stores data in a way that allows machines to understand the meaning of information in the complete context of their relationships. Consequently, Grakn allows computers to process complex information more intelligently with less human intervention.
* **Graql**is a declarative, knowledge-oriented graph query language that uses machine reasoning for retrieving explicitly stored and implicitly derived knowledge from Grakn.

Source:

* <https://dzone.com/articles/get-started-with-graknai>
* <https://dev.to/manfromsiberia/weekend-experiments-with-grakn-ai-2a8i>

Pros:

* Impressive and infinitely flexible schema, so you can create complex knowledge models
* Built-in distributed analytics algorithms, so you can analyze multidimensional data interactively
* Graql’s strong abstraction means writing less code- and it also automatically optimizes query execution

Cons:

* Lack of timezone support- so you have to account for offset all on your own
* Hard-to-grok error messages- unless you’re a Java developer

Diagram

Description automatically generated

**JanusGraph:**

It is a scalable graph database optimized for storing and querying graphs containing hundreds of billions of vertices and edges distributed.

Source:

* <https://janusgraph.org/>
* <https://stackoverflow.com/questions/55059291/what-are-the-pros-and-cons-when-using-remote-janusgraph-connection-over-embedded>

Pros:

* Scalable
* Open Source
* Graph Data canbe stored in Cassandra, Hbase
* Native Integration with Apache TinkerPop graph stack: Gremlin graph query language, Gremlin Server

Cons:

* Tough to do a manual transaction management
* You have to use groovy script as string and send it to remove for transactional execution of your code

**StarDog:**

Stardog is a reusable, scalable knowledge graph platform that enables enterprises to unify all their data, including data sources and databases of every type, to get the answers needed to drive business decisions.

Source:

* <https://www.stardog.com/>
* <https://www.capterra.com/p/169579/Stardog/>
* <https://datalanguage.com/blog/graphql-and-graph-databases>

Pros:

* It helps in Data Capture, Data Migration, Data Security, Data Integration, Metadata Maanagement
* Fully compliant with RDF/SPARQL standards
* Extremely fast online write times
* Adequate text search capability using internal indexing
* Geospatial querying and other extensions
* Rich visual query and management workbench UI

Cons:

* License Cost

**GraphDB:**

External RDF and graph database with efficient reasoning, cluster and external index synchronization support

Source:

* <https://www.ontotext.com/products/graphdb/>
* <https://db-engines.com/en/system/Amazon+Neptune%3BGraphDB%3BNeo4j>

**How do GraphQL and Graph Databases fit together?**

As we learned earlier, GraphQL is a REST API alternative with a structured query language combined with a runtime for query processing and serving data to GraphQL clients (and it is almost always implemented with JSON in mind).

However, GraphQL does only provide another query mechanism for your graph database alongside SPARQL or Gremlin.  Moreover it is much less expressive, and most definitely not designed for exploiting the rich semantics of RDF and OWL.

In fact if you go all in on a michelin-starred Graph Database, and only use GraphQL to query it, I would argue you really probably don’t need a Graph Database - a NOSQL JSON store would probably suffice. Of course, if you have forward-chaining inference in your graph database (as with GraphDB), then your GraphQL queries will benefit from the RDFS/OWL semantics you have modelled your data with, which does add weight to the value-proposition.

**Ontologies:**

Ontologies are an important part of the picture. In the RDF world an ontology semantically describes the model and nature of your domain, and ultimately the data structures stored in your knowledge graphs, and if inference-aware, they describe the meaning of the data, and how we reason upon it.

**So do we need GraphQL to take advantage of these ontological graph models ?.**

No we don’t, we can do it with normal REST APIs (querying our graph with SPARQL) and using JSON-LD payloads - but of course we have to build more API code to do this.

With GraphQL though we can (we must) formally describe our data structures as GraphQL schemas, and so we have a great opportunity here to align our GraphQL schemas with those same ontology models that describe the data in our knowledge graph -  this could be either a subset of our ontology landscape, or possibly the superset, if we are using other GraphQL (served) data sources aside from the data in the graph database. This is starting to look like very nice decoupled data architecture with well structured cohesive data models, with possibly the most efficient mechanism for consuming data in our web/mobile applications (a GraphQL client).

Adding GraphQL client interfaces to a Graph Database does make life easier- if you are rapidly prototyping straight from the graph or federating data into an external GraphQL server there is a nice pattern for making a common set of well described data models.While GraphQL lowers the barrier to entry, GraphQL gives you only as much as it is able to with respect to querying and graph traversal; to really unlock the power of the graph database, you will need to use SPARQL.

# **General Note on Graph Database in AWS**

Graph Data Platform is a tough nut to crack in general. If you're into the AWS stack your pipelines consist of more than some basic SQL triggers; and possibly some combination of AWS Glue and Lambdas, ECS/Fargate, Kinesis, Firehose, S3 and more for the pipeline.

This comes down to build vs. buy, but on AWS we can think about three distinct ways to approach this problem:

1. You can build what you need from the ground up. Depending on your team's technical capabilities and time that you want to invest, this may be a suitable option. There are open source projects that enable Knowledge graph, notably Neo4J that leverages one of the EMR HDFS tools this specific open source option may not be as valuable (although you can use the APIs to express your data sets and to create knowledge graph between them).
2. Being fully on AWS opens up AWS Glue - but this is actually heavily predicated on using AWS Neptune since the Knowledge graph capabilities for AWS Glue are using Python libraries for interacting with AWS Neptune. The benefit of Glue is that if you use it and if you do a lot of your processing using Hive then you can get really really rich metadata definitions from Glue along with relatively simple plug and play interactions with Hive for the knowledge graph.
3. You can purchase an off the shelf product and integrate that into your ecosystem, most knowledge graph tools have pretty good automated metadata capture so you can import the shape and semantics of your data rather easily. What may be more difficult, especially if you are building complicated serverless pipelines using some combination of Python / JS / Go / etc. and SQL to interact with the database, is how you represent the data movement between your schemas in these tools. It will be difficult to find any tool that can automatically parse your code to represent your pipelines and systematically generate the semantics - so expect to have to spend lots of time documenting how your data moves.

# **Conclusion**

Using graph databases as a technical solution are also ideal for other emerging technologies, like AI, IoT, 5G applications, and so on. But they can also transform legacy systems.

A graph database may have many different infrastructure implementations but they all support graph modeling which interconnect different components with their associations. As mentioned, such a sea of change in data modeling will be an extremely simple and straightforward solution for many daily system scenarios. They will also offer much faster throughput and lesser DevOps requirements.

# Bibliography

1. **Dataversity.** Dataversity. [Online] October 7, 2017. https://www.dataversity.net/what-is-a-graph-database/.

2. **Ambi, Chetan.** The Beginner's Introduction to Graph Databases. [Online] November 17, 2020. https://medium.com/towards-artificial-intelligence/the-beginners-introduction-to-graph-databases-6de8e9a9bfce#:~:text=Cases%20where%20graph%20databases%20are,t%20need%20a%20graph%20database..

3. **Wilton, Paul.** GraphQL and Graph Databases. [Online] February 05, 2020. https://datalanguage.com/blog/graphql-and-graph-databases.