You may have heard about Graph databases but are they right for you? In this Write Stuff article, Graham Cox looks at the concepts and application of Graph databases.

If you are reading this article then no doubt you have already heard of the concept of a Graph Database, and are looking to learn more about what they are and what they can do for you.

Graph Databases are currently gaining a lot of interest, as they can give very powerful data modeling tools that provide a closer fit to how your data works in the real world. This can allow a large level of flexibility to represent your data in a way that makes the most sense to everyone involved, whilst still making the most of the complex interactions between it.

This article aims to explore exactly what they are and where they can be a good fit in your application landscape.

What is a Graph?

Before we can understand what a Graph Database is, we first need to understand what is meant by a Graph.

In this context, a Graph Database represents a mathematical Graph. Specifically a Graph Database will typically be a Directed Graph.

In Mathematical terms, a Graph is simply a collection of elements - typically called Nodes (also called Vertices or Points) - that are joined together by Edges. Each Node represents some piece of information in the Graph, whereas each Edge represents some connection between two Nodes.

A Directed Graph is a special type of Graph where edges always have a direction associated with them. Conversely, an Undirected Graph would be one where the edges are simply links with no direction associated with them.

Once you start dealing with Graphs, you very quickly get involved in Graph Theory. This is a branch of Mathematics that deals with the complexities that Graphs can contain, and with how best to get information out of them.

Graphs are already prevalent in the real world, and in software development. For example, any time you try to use a Tube Map or trace a Family Tree, you are dealing with a Graph.

Even using the Internet on a daily basis is using a Graph. Each computer on the Internet - servers, routers, switches - is a Node, and each connection between them is an Edge. Some elements of Graph Theory are then very important in the infrastructure used here, in order to correctly connect distant computers together in the best way.

What is a Graph Database?

At it's most basic, a Graph Database is simply a Database Engine that models both Nodes and Edges in the relational Graph as first-class entities. This allows for you to represent complex interactions between your data in a much more natural form, and often allows for a closer fit to the real-world data that you are working with.

Graph Databases are often schema-less - allowing for the flexibility of a Document or Key/Value Store database - but supporting Relationships in a similar way to that of a traditional Relational Database. This doesn't mean that there is no data model associated with the database though. Simply that there is more flexibility in how you define it, which can often lead to the faster iteration of your projects.

This is all possible in other database solutions, but not always as elegantly as in a Graph Database and often involving link tables or nested documents to achieve the same level of expressiveness.

Graph Databases also often allow us to apply Graph Theory to our data in an efficient manner, allowing us to discover connections from our data that are otherwise difficult to see. For example, minimal routes between nodes, or disjoint sets within our data.

Worked Example - or How do different database solutions differ?

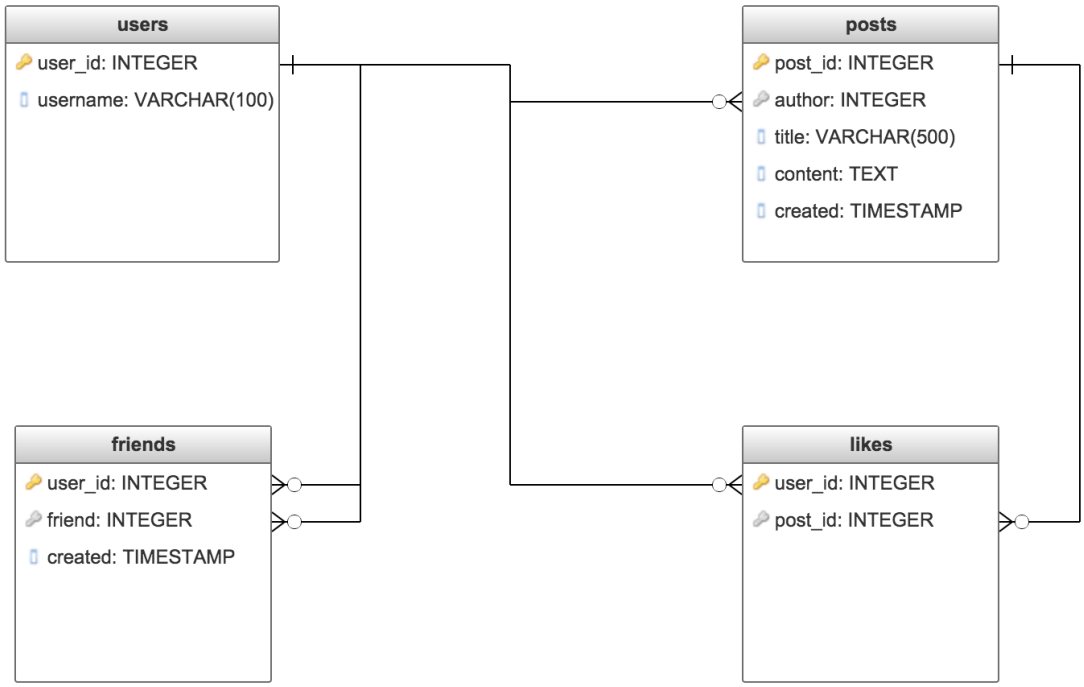
The best way to understand the benefits of such a solution is often to see it in action. As such, we will cover a worked example of a simple Social Network, implemented in a Relational Database (e.g. MySQL), a Document Database (e.g. MongoDB) and a Graph Database.

All three of these solutions will represent the same data but will do it in their own ways. This allows us to quickly see the commonalities and the differences between the three solutions.

Our simple Social Network will have only two types of entity - Users and Posts. Users have Friends, are able to write Posts, and are able to Like Posts. We are then going to explore how to retrieve a relatively complex answer from this - All of the Friends of any User who has Liked one of my Posts, in alphabetical order of username.

Relational

In a typical Relational Database, this will likely be modeled using four different tables - `users`, `posts`, `friends` and `likes`. These might look something like this:



We have ended up with 4 different tables, with 5 foreign key relationships between them.
Two of these tables are actual data, and the other two are nothing more than links between entities in our system.

Answering our query in this data model is complicated but can be achieved with a single query.

```
SELECT friends_of_likers.*
FROM posts
JOIN likes ON (posts.post_id = likes.post_id)
JOIN users likers ON (likers.user_id = likes.user_id)
JOIN friends ON (likers.user_id = friends.user_id)
JOIN users friends_of_likers ON (friends_of_likers.user_id = friends.friend)
WHERE posts.author = :me
ORDER BY friends_of_likers.username ASC
```

It's hardly pretty, and it's not especially easy to read this query to work out what it does.
It ends up joining together 5 resultsets just to get the results from one of them.
It will work though, and it will return all of the information we desire in only a single query - however efficient that may be.

Document Store

In a Document Store Database, there are a number of different ways that this can be modeled depending on exactly what you want to achieve. Often, relationships between entities of different types are difficult to achieve, either being modeled as a nested document or as a manually

enforced foreign key. We will go for a mixture of the two, giving us a `users` and a `posts` collection to work with.

Users

```
{
  "user_id": "u1",
  "username": "grahamcox",
  "friends": {
    "u2": "2017-04-25T06:41:11Z",
    "u3": "2017-04-25T06:41:11Z"
  }
}
```

Posts

```
{
  "post_id": "p1",
  "author": "u1",
  "title": "My first post",
  "content": "This is my first post",
  "created": "2017-04-25T06:41:11Z",
  "likes": [
    "u2"
  ]
}
```

Straight away we've reduced the number of entities we are modeling down to two - which is correct from our original data modeling. We've also made it so that we get some of the related data about an entity all in one go - a Post and all of the Likes, for example. However, the cross-links from Post to User and from User to User are harder to manage in this setup. Also, remember that most Document Databases don't support relational integrity so these cross-links need to be maintained by the software, and support needs to be built in for when they are broken.

However, in order to answer our query in this data model is going to need multiple queries. Because Document Stores don't generally support cross-links, we will need to do the various joins in code instead. In this case, we will need to:

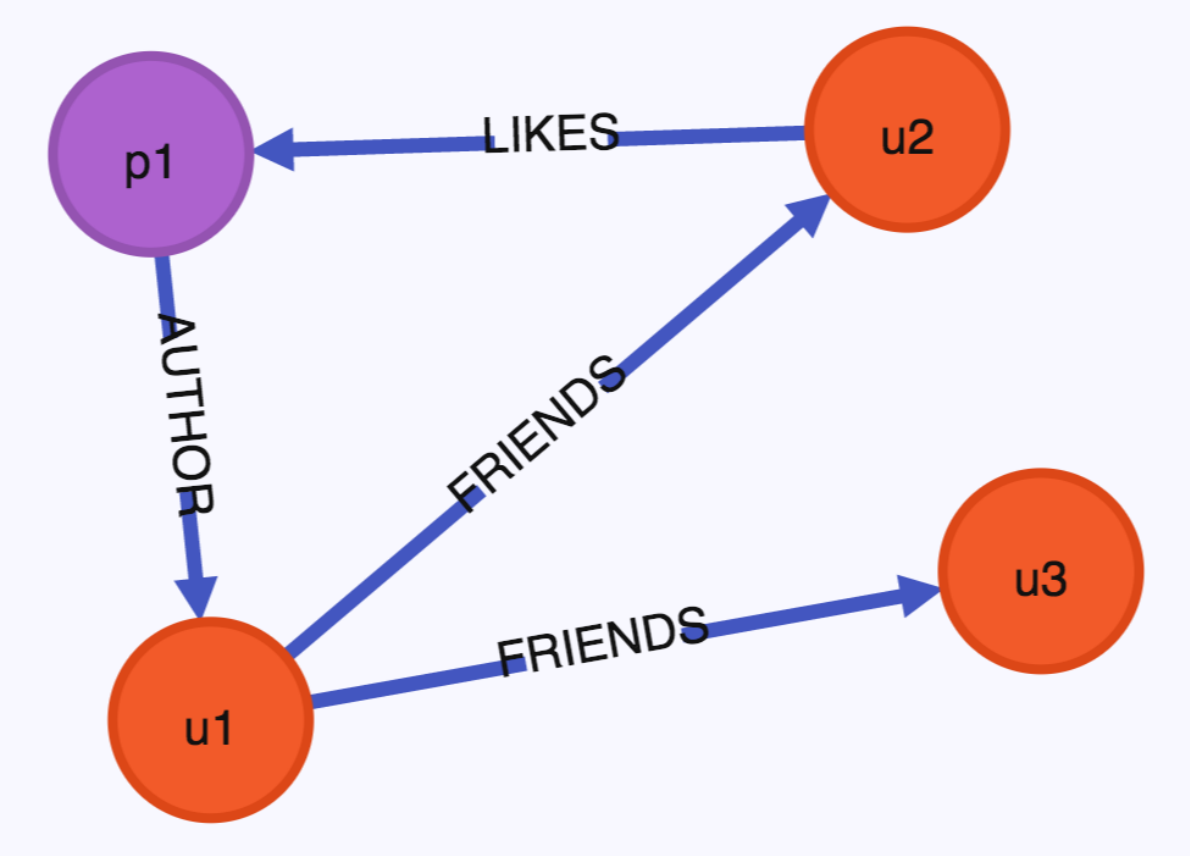
- Query 1: Find all of my posts, which will include the IDs of all the users who liked those posts.
- Manual processing: De-duplicate the list of User IDs
- Query 2: Find all of the users who liked any of my posts, which will include the IDs of all of the friends of those users
- Manual processing: De-duplicate this list of User IDs
- Query 3: Find all of the users that will actually solve our query

Each of these queries is relatively painless to execute - they are just returning documents on a simple key.
However, the fact that we need to do three different queries, and some manual processing in between each one is just painful. We can possibly reduce this by having some assumptions about

our data model - for example, if friends links are always both ways then we can merge the second and third queries together - but this is then adding limits into our data model to make these queries better. And these limits are not always correct to add in.

Graph Database

In a Graph Database, we can choose to model the Entities as our Nodes, and the Relationships as our Edges. This gets us closer to the Document Store model - where we only have two types of Entity - but with the power of the Relational Model - where we don't have to handle links between Entities manually, and where we can easily traverse these links inside the database itself. This might look something like this:



Here we have two different types of Node, and three different types of Edge.
Whilst not visible in the diagram, the Nodes and Edges can each contain data, similar to the Relational model.
For example, the "FRIENDS" Edge would contain the date when the Relationship was created, allowing us to list all Friends in time order.

This very quickly shows us that we have all of the power that we are used to from the Relational model, but with the flexibility we are used to from the Document Store model.

Now to answer our example query using this. This can be solved as follows (Using the Cypher query language)

```
MATCH (:User {id:{author}}) <-[:AUTHOR]- (:Post) <-[:LIKES]- (:User) <-
[:FRIENDS]- (u:User) RETURN (u)
```

This is actually not too dissimilar to the Relational Database query, except that the query is much more readable and the links are much more obvious.
We can also clearly see that there is a distinction between Nodes and Relationships here and that we are following Relationships to get from one Node to another. You can even traverse this query by simply tracing your finger across the named lines on the above diagram.

The real thing to notice though is that nowhere are we telling the database engine how to link the Nodes together. We simply tell it to follow a Relationship of a certain type and it handles everything for us automatically. No more necessity to match IDs in different tables and hope that they correspond correctly.

Should I use a Graph Database?

Obviously, a Graph Database will not always be the best fit for your needs. Every situation is different and you need to evaluate the requirements every time. The most important thing you need to do is evaluate your data model. It's very likely that it is highly relational. Most real world data models are. In this case, a Graph Database is already likely to be a good fit for your needs.

Next, determine the type of relationships that your data has. If it contains a number of Many-To-Many relationships then a Graph Database will probably work better for your needs than a traditional Relational Database. Even if it contains a number of One-To-One or One-To-Many relationships though, a Graph Database may make this easier to represent.

Thirdly, determine the schema of your data.
Graph Databases are generally much more flexible in the way that they allow you to store data, allowing for much more fluidity of the data present in each location. If your data needs are such that the schema is not absolutely rigid then a Graph Database may be a better fit, even if a Relational Database fits your needs otherwise.

Finally, determine what you want to do with your data.
If you want to do complex data analysis, or potentially expensive queries spanning multiple types of data, then a Graph Database may make this easier to achieve and will possibly make the queries run more efficiently.

Options for what to use

Once you've decided you want to use a Graph Database, the next hurdle is to decide which one to go for. There are quite a few options available, and we are going to briefly cover some of these here to help determine which is the best fit for your project.

We are going to summarize the features of Neo4J, OrientDB, ArangoDB and JanusGraph to help decide which is the best fit for your project. Note that these are not the only options to chose from, so please investigate the options fully before deciding.

Graph Databases Compared

Feature	Neo4J	OrientDB	ArangoDB	JanusGraph
First Release	2007	2010	2011	2017
Current Release	3.1.3 (March 2017)	2.2.19 (April 2017)	3.1.17 (April 2017)	0.1.0 (April 2017)
Written in	Java	Java	C++/JavaScript	Java
License	GPLv3 for Open Source Projects. Commercial for closed source projects.	Apache 2.0	Apache 2.0	Apache 2.0
Query Language	Cypher	OrientDB SQL	AQL	Gremlin
Live Backups	Enterprise only	Yes with OS Support	Yes	Depends on backend database
Replication	Enterprise only	Yes	Yes, but some features only with Enterprise	Depends on backend database
Sharding	Enterprise only	Yes, but with some limitations	Yes	Depends on backend database
Schema support	Yes	Schema support if desired	No	Schema support if desired

Neo4J is likely the name that most people know when thinking about Graph databases. It's the oldest option around, and the best-known name. However, it is not as feature rich as other options, and possibly not as performant (based solely on other benchmarks online, so not necessarily reliable.)

JanusGraph, on the other hand, is a very new name in the Graph Database scene. It has been in development since 2012 but had its first release in 2017. It is being worked on under The Linux Foundation and is completely free for anyone to use or contribute towards.

Summary

When starting a new project, there is often a tendency to use technologies that are well known, or else that are new and well discussed. Consideration should be taken to see if these are really the best fit for what you need though, or if something else might work better for you.

A Relational Database is often considered the safe option, and there is a myriad of NoSQL database solutions that get a lot of discussions online these days that may be tempting to use as well.

But if you want the best of both - the flexibility and speed of iteration that is common in NoSQL databases, combined with the relational modeling power from a Relational Database - then you should consider looking at a Graph Database instead, and see what it can do for you.

[[https://compose.com/articles/introduction-to-graph-databases]]