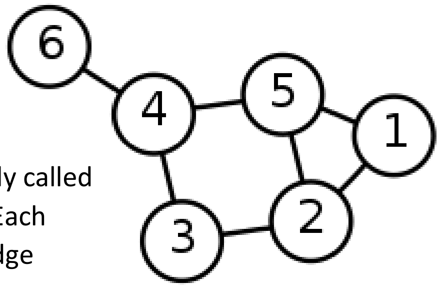
8 Types of NoSQL Databases – Graph Databases

8.1 Introduction

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.

8.1.1 What is 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.

8.1.2 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**.

8.1.3 Worked Example - or How do different database solutions differ?

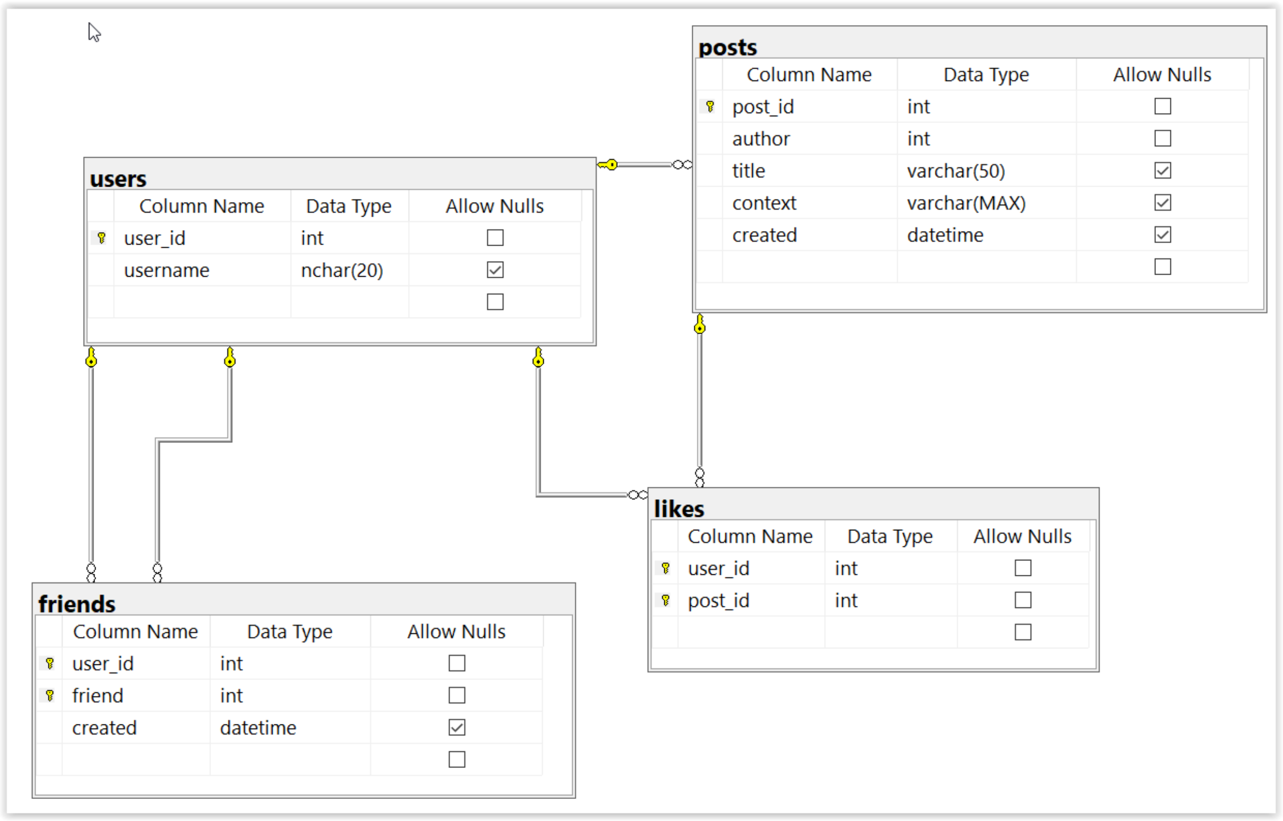
The best way to understand the benefits of such a solution is often to see it in action. As such, we will cover a worked example of a simple Social Network, implemented in a Relational Database (e.g. MySQL, SQL Server, PostgreSQL), a Document Database (e.g. MongoDB) and a Graph Database.

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:



posts

	Column Name	Data Type	Allow Nulls
🔑	post_id	int	☐
	author	int	☐
	title	varchar(50)	☑
	context	varchar(MAX)	☑
	created	datetime	☑
			☐

users

	Column Name	Data Type	Allow Nulls
🔑	user_id	int	☐
	username	nchar(20)	☑
			☐

likes

	Column Name	Data Type	Allow Nulls
🔑	user_id	int	☐
🔑	post_id	int	☐
			☐

friends

	Column Name	Data Type	Allow Nulls
🔑	user_id	int	☐
🔑	friend	int	☐
	created	datetime	☑
			☐

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.

All of the Friends of any User who has Liked one of my Posts, in alphabetical order of username:

```
select l.user_id,u.username
from posts p join likes l on p.post_id = l.post_id -- likes of my posts
join friends f on l.user_id = f.user_id  -- friends of users who liked my posts
join users u on f.user_id = u.user_id  -- usernames of those friends
where p.author = 10  -- my posts
order by username desc;
```

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 4 tables 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 (using e.g. **unwind**)

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

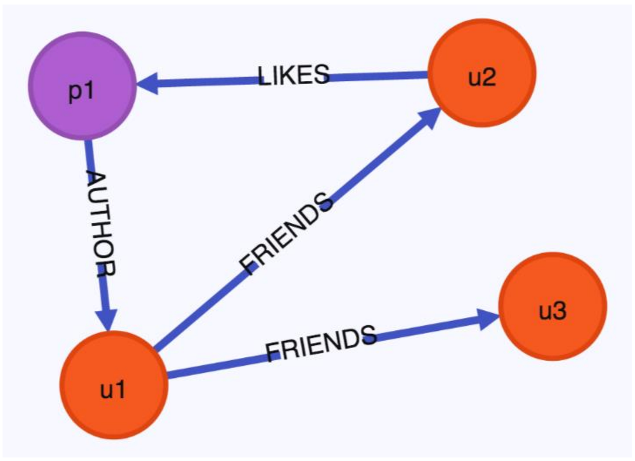
Query 3: Find all of the usernames that will actually solve our query

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.

8.1.4 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.

8.2 Neo4j

8.2.1 Introduction

Neo4j is an open-source, NoSQL, native graph database that provides an **ACID-compliant** transactional backend for your applications.

8.2.2 Neo4j Sandbox

Go to https://neo4j.com

Choose "Get started free” and log in (or first sign up).

Create a blank sandbox and open it with the browser.

8.2.3 Getting started

8.2.4 Some introductory queries

1. Execute the following command to delete all nodes and relationships

```
MATCH (n) DETACH DELETE n
```

2. Download the file friends.txt on Chamilo. Execute the CREATE statements.

3. Retrieve all nodes from the database

```
MATCH (n) RETURN n
```

4. Examine the schema of the database

```
CALL db.schema.visualization()
```

5. Retrieve all User nodes

```
MATCH (u:User)

RETURN u
```

6. Retrieve all User nodes with City = Los Angeles

```
MATCH (u:User)

WHERE u.city = 'Los Angeles'

RETURN u
```

7. Give all Users that do not have a city property, returning their names

```
MATCH (u:User)
WHERE (u.city is  null)
RETURN u.name,u.city
```

8. Give the number of friends for User Bradley

```
MATCH (u:User)-[:FRIEND]->(ou:User)

WHERE u.name = 'Bradley'

RETURN count(*) As NumberOfFriends
```

Check what happens if you omit the arrow. Explain

9. Give all (immediate) friends of User Lisa

```
MATCH (l:User)-[:FRIEND]->(ol:User)

WHERE l.name = 'Lisa'

RETURN ol
```

10. Give all friends of friends of User Lisa (= friends exactly 2 hops away)

```
MATCH (l:User)-[:FRIEND*2]->(ol:User)

WHERE l.name = 'Lisa'

RETURN ol
```

Explain the result.

This illustrates the strength of graph databases. Very interesting use case for e.g. Facebook advertisers.

11. Give all friends of friends of User Lisa (= friends exactly 2 hops away) that are not immediate friends of User Lisa

```
MATCH (l:User)-[:FRIEND*2]->(ol:User)

WHERE NOT ((l)-[:FRIEND]->(ol)) AND l.name = 'Lisa'

RETURN ol
```

12. Give all friends of friends of User Lisa (= friends exactly 2 hops away) that are not immediate friends of User Lisa and aren't User Lisa either

```
MATCH (l:User)-[:FRIEND*2]->(ol:User)

WHERE NOT ((l)-[:FRIEND]->(ol)) AND NOT ol = l AND l.name = 'Lisa'

RETURN ol
```

13. Find all the users reachable from Annie

```
MATCH (a:User)-[:FRIEND*..]->(f)

WHERE a.name = 'Annie'

RETURN f
```

14. Find the mutual friends between Annie and Lisa

```
MATCH (a:User)-[:FRIEND]->(mf)<-[:FRIEND]-(l:User)

where a.name = "Annie" and l.name = "Lisa"

RETURN mf.name
```

15. Find out which user shares the most common friends with Annie

```
MATCH (a:User)-[:FRIEND]->(f)<-[:FRIEND]-(ou:User)

WHERE a.name = 'Annie'

RETURN ou, COUNT(f)

ORDER BY COUNT(f) DESC

LIMIT 1;
```

16. Find a new friend for Annie based on maximum mutual friends

```
MATCH (a:User)-[:FRIEND]->(f)<-[:FRIEND]-(ou:User)

WHERE a.name = 'Annie' and NOT (a)-[:FRIEND]->(ou)

RETURN ou, COUNT(f)

ORDER BY COUNT(f) DESC

LIMIT 1
```

17. Give the mutual interests for User Bradley and User Lisa

MATCH (b:User)-[:INTEREST]->(i:Interest)<-[:INTEREST]-(l:User)

WHERE b.name = 'Bradley' AND l.name = 'Lisa'

RETURN i

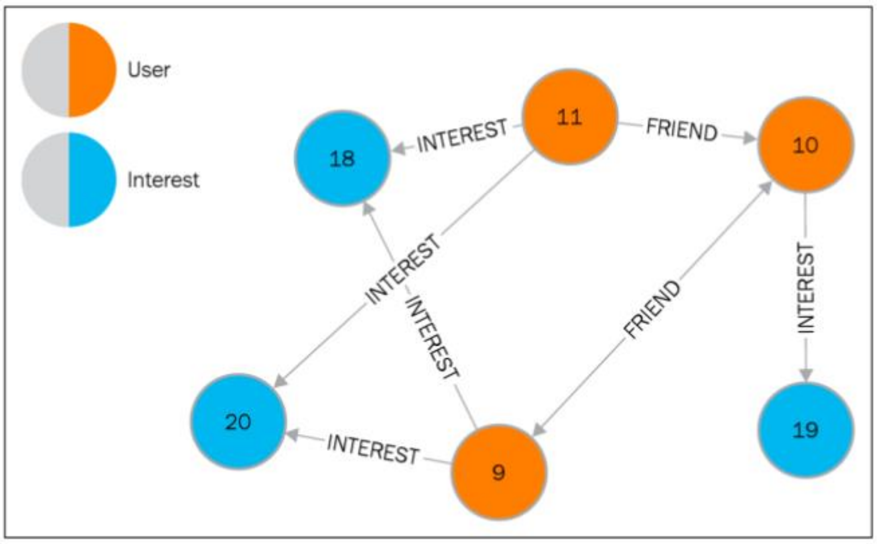
18. Find 3 new friends for Bradley based on maximum number of common interests

MATCH (b:User)-[:INTEREST]->(stuff)<-[:INTEREST]-(new_friend)

WHERE b.name = 'Bradley' AND NOT (b)-[:FRIEND]-(new_friend)

RETURN new_friend.name, count(stuff)

ORDER BY count(stuff) DESC LIMIT 3



19. Find the most recently status_posted by one of the friends of Annie

MATCH (a:User)-[:FRIEND]->(f)-[:STATUS]->(s:Status)

WHERE a.name = 'Annie'

RETURN f, s.date

ORDER BY f.name, s.date DESC

20. Find the status updates by users sharing interests with Bradley

MATCH (b:User)-[:INTEREST]->(i:Interest)<-[:INTEREST]-(ou:User)-[:STATUS]->(s:Status)

WHERE b.name = 'Bradley'

RETURN DISTINCT ou, s.text

8.3 Exercises

Exercise 1

Execute the following command to delete all nodes and relationships

MATCH (n)

DETACH DELETE n

Download the file flights on Chamilo. Execute the CREATE statements.

Retrieve all nodes from the database

MATCH (n) RETURN n

Examine the schema of the database

CALL db.schema()

Create the following Airports:

- name: 'JF Kennedy Airport' + city: 'New York'
- name: 'Austin Bergstrom International' + city: 'Austin'

Create the following Flight:

- flight_number: BG45 + month: August

Create the following 2 relationships

- origin of flight with flight_number BG45 was JFK and destination of this flight was Austin Bergstrom International

Answer the following questions

- Retrieve all flights that originate from any airport in Atlanta city that are destined to Dallas/Fort Worth
- Retrieve all flights that originate from either Atlanta or Dallas/Fort Worth
- Retrieve all flights that originate from any city, but not from Atlanta
- Retrieve all flights that start from either Atlanta or Dallas/Fort Worth and have been delayed due to late aircraft arrival
- Retrieve the delay times of the most delayed flights, along with the flight number and reason for delay
- Retrieve the unique names of the airports on which flights take off.
- Retrieve the 3 most delayed flights

Exercise 2

Execute the following command to delete all nodes and relationships

MATCH (n)

DETACH DELETE n

Download the file simpsons on Chamilo. Execute the CREATE statements.

Retrieve all nodes from the database

MATCH (n) RETURN n

Examine the schema of the database

CALL db.schema()

Answer the following questions

1. List all men over 40 along with their age. For the current year, use date().year. Sort descending by age.

Sample solution

name	age
"Abe Simpson"	95
"Clancy Bouvier"	92
"Herb Simpson"	67
"Homer Simpson"	65

2. Give the average age of all mothers. For the current year, use date().year.

Sample solution: 78.2222

3. List all men who are fathers along with their age. For the current year, use date().year. Sort ascending by age.

Sample solution

name	age
"Homer Simpson"	65
"Clancy Bouvier"	92
"Abe Simpson"	95

4. List all women who are daughters (in the dataset) along with their age. For the current year, use date().year. Sort descending by age.

Sample solution

name	age
"Patty Bouvier"	65
"Marge Bouvier"	64
"Selma Bouvier"	62
"Lisa Simpson"	33
"Maggie Simpson"	27
"Ling Bouvier"	26

5. Give everyone who is simultaneously father and son (in the dataset)

Sample solution

name
"Homer Simpson"

6. Give all grandfather-grandchildren relationships

Sample solution

grandfather	grandchild
Clancy Bouvier	Bart Simpson
Clancy Bouvier	Lisa Simpson

Clancy Bouvier	Maggie Simpson
Clancy Bouvier	Ling Bouvier
Abe Simpson	Bart Simpson
Abe Simpson	Lisa Simpson
Abe Simpson	Maggie Simpson

7. Give all couples who have been together for over 50 years in the following way. For the current year, use date().year.

Sample solution

Husband	**Wife**
"Abe Simpson"	"Mona Simpson"
"Clancy Bouvier"	"Jacqueline Bouvier"

Exercise 3

Execute the following command to delete all nodes and relationships

```
MATCH (n)

DETACH DELETE n
```

Download the file healthcare on Chamilo. Execute the CREATE statements.

Retrieve all nodes from the database

```
MATCH (n) RETURN n
```

Examine the schema of the database

```
CALL db.schema()
```

Answer the following questions

- Give for each patient the number of cancer symptoms
- Give for patient John the number of ingredients of pizza John is allergic from
- Give for patient John the number of drugs of his treatments that interacts with the drug Abc