Graph Databases

1 Introduction

A great deal of today's software applications are built around databases and perform various tasks on large data sets to serve certain business interests. Relational databases have been around for decades and have been optimized to serve this purpose best. However, with the growing of the internet and the vastly increasing amount of data (big data), new types of databases have emerged under the designation NoSQL. Common representatives of this kin are key-value stores and document-oriented stores, but also the less established graph databases. In contrast to its relatives, the latter focuses strongly on relations and the topology between entities instead of the entities themselves. Graph databases therefore provide customized query languages with rich expressiveness regarding relations and also feature algorithms from graph theory. They are therefore suitable for problems where efficient graph traversal and related graph algorithms are essential.

The subject of this document is to examine the usability and performance of graph databases. As an example for a graph-related problem, two games based on the Wikipedia will be demonstrated:

1. Wikipedia Philosophy game
   Select a random article. Follow the first non-italic, non-parenthesis-enclosed link of the main text and repeat this step until a cycle is detected or the article Philosophy is reached. Several persons analyzed this issue und came to the conclusion that roughly 95% of all articles link to Philosophy this way[1].
2. 5 clicks to Jesus
   Select a random article. Using all links of the main text, try to reach the article Jesus in 5 clicks (links) or less.

2 Data sets

Three dumps of the Wikipedia articles have been used throughout the tests:

Language	Date of dump	# Articles	# Links
English	2014-12-08	15113788	164379808
Russian	2015-01-10	3039356	36903313
Plattdütsch	2015-01-08	31814	607331

The English Wikipedia was used initially with the goal to solve the original philosophy game. However, during preprocessing we came to the conclusion that the data set was far too large to build a fair database for benchmarks and tests. The reason for this is that, although all articles have ids, Wikipedia stores its links using the textual name of the linked page, not its id. Still, well-designed relational databases use integral ids to refer to foreign entities. Considering a fair comparison, we tried to resolve all textual links (several million) to their page ids. However, with <10 resolved ids per second due to the large dataset, we aborted this process as it would have probably taken months to complete. It would be possible to resolve the ids without a database using e.g. a custom coded program and a hash map with enough size to store 10 GiBs of strings as keys with associated values. Nevertheless, this would not solve the equivalent problem when inserting into a graph database

[1] http://matpalm.com/blog/2011/08/13/wikipedia-philosophy/

where relationships can only be created after still matching the destination page by either string or integer. The ultimate choice was to drop the large data set and try a smaller one. First experiments with the Russian Wikipedia showed that resolving all pages would still require approximately 11 days. Furthermore, due to the language's nature, Unicode support became an essential requirement, doubling the storage size of the database. While inserting all links, we discovered that our SQL database (LocalDb) has a limit of 10 GiB per table. Thus also the Russian Wikipedia was too large for our experiment. Finally, we have chosen a quite small Wikipedia which can still somehow be understood, the Plattdüütsch Wikipedia.

3 Handling large files

The database creation with the selected test data was unfortunately preceded by a long period of preprocessing (several weeks). It turned out, that the size of the original English Wikipedia dump (~ 49 GiB) was quite hard to handle. Here is a list of difficulties we had to face during processing the dumps and how we got around them:

3.1 The file size of ~ 49 GiB

Files with such sizes cannot be opened in ordinary text editors like Notepad++. We tried a number of proprietary (shareware) text editors claiming to be able to deal with this dimensions. Although a lot of tools could view the file, none was capable of performing operations like search&replace or regular expressions at reasonable speeds (e.g. one software performed search&replace at 27 lines/s on a 164.379.808 lines file). We therefore used a tool to split the first e.g. 10 MiB form the file to analyze the data layout and try some transformations. If they worked out, we coded them using C# and processed the file line-by-line.

3.2 Lines longer than String can hold

Initially, all destination links of a page where parsed from the wiki markup of each page. We later found out that Wikipedia also provides SQL dumps of their link table which is generated from the wiki markup. However, this SQL dump contained all values as tuples of a single line INSERT statement. With several million tuples on a single line this data could only be read chunk-by-chunk which is impractical when we want to run a regular expression over multiple chunks. We therefore dropped the SQL dump and stayed with the links parsed from the wiki markup.

3.3 String.Split() fails to allocate memory

The implementation of String.Split() inside the .NET framework returns an array of all substrings split from the original one. From reading through the source code we also found out that quite some temporal data is also allocated during splitting. This caused String.Split() to fail when splitting larger lines. We therefore implemented a lazy version of splitting strings.

3.4 Regex and performance

Link extraction from wiki markup is done using regular expressions which run on the wiki markup of each page. During a performance analysis we found out that most of the time our preprocessing tools were not bound by disk IO but by the execution speed of the regular expression. Therefore, processing of pages and link extraction has been parallelized and compiled regular expressions where used to process dumps at reasonable speeds (e.g. 40 min compared to several hours when run on the English Wikipedia)

3.5 SQL INSERT throughput

Initially, our preprocessor generated SQL scripts with millions of SQL INSERT statements. Despite connection pooling and even holding a single connection for all queries, the database could only

process around 1000 queries per second which was far too slow for a dataset with several million records. Therefore BULK INSERTs where used with appropriately generated CSV files yielding a much higher throughput

3.6 Unicode

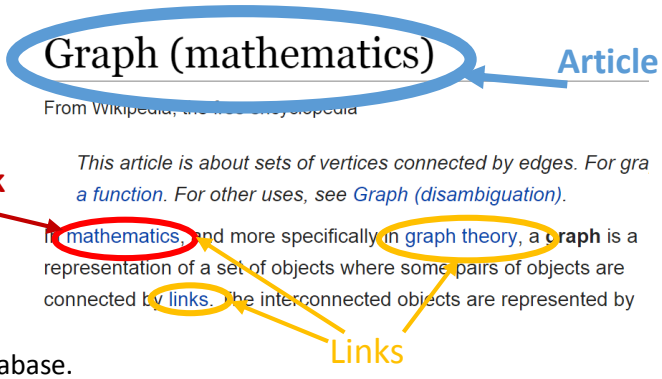
By switching to a non-English Wikipedia, support for a wide range of characters became a requirement. Switching to Unicode seemed a good option but introduced a bunch of problems. First, Unicode allows a single textual character to be represented by multiple physical characters (e.g. combining characters and surrogates). Splitting a string or creating a substring was suddenly a non-trivial task. Furthermore, SQL databases require special column types to support Unicode (e.g. nvarchar). Finally, also the various encodings were a little annoying as MS SQL Server (and LocalDb) does not support UTF-8 whereas our graph database of choice (Neo4j) requires input files in UTF-8.

4 Method and Datamodel

To compare a graph database and a relational database we used the same dataset for both systems. The same tasks (conceptionally) were performed on both database systems and the time was measured. From the same data a graph representation containing nodes and relationships as well as a relational schema based on tables and rows was created.

Both contain (Wikipedia-)pages and links between pages. Also the first link of the main text of each page is stored. Every page stores

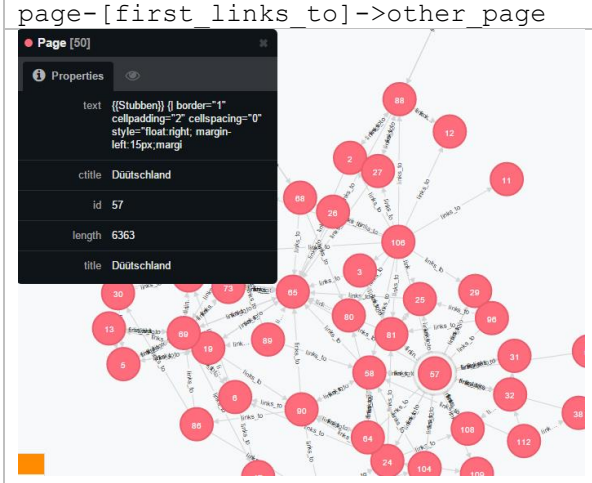
- the page title,
- the page text (100 characters) and
- the length of the text.



The links between the pages and the first link are differently modelled, depending on the database. Whereas the ones in the graph database are represented as relationship between nodes, the relational database uses ids and a link-table.

Graph Structure	Relational Schema

`(Page {id, title, text, length})`
`page-[links_to]->other_page`
`page-[first_links_to]->other_page`
	`Page(id, title, text, length)`
	`Link(src, dst)`
	`FirstLink(src, dst)`



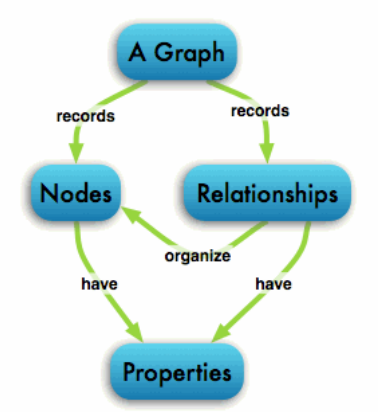
	id	title	ctitle	length	text
1	1	Astronomie	Astronomie	1426	De '''Astronomie''' ([[Greeksche Spraak|greeksch]] '''αστ...
2	2	Albanien	Albanien	2455	{| border="1" cellpadding="2" cellspacing="0" style="flo...
3	3	Andorra	Andorra	1846	{{Infobox Land | INAAM_INHEEMSCH = Fürstendom An...
4	4	Adam vun Bremen	Adam_vun_Bremen	2899	'''Adam vun Bremen''' (Adamus Bremensis) weer en Scho...
5	5	Afrika	Afrika	2174	[[Bild:Location Africa.png|thumb|Afrika]] '''Afrika''' is en [[E...
6	6	Amerika	Amerika	634	[[Bild:Location Americas.png|duum|Laag vun Amerika]] D...
7	7	Asien	Asien	423	[[Bild:Location Asia.png|thumb|right|Asien]] '''Asien''' is de ...
8	8	Australien	Australien	1233	{{Anner Bedüden|Australien (Eerddeel)}} [[Bild:Location...
9	9	Afghanistan	Afghanistan	4364	{| border="1" cellpadding="2" cellspacing="0" align="rig...
10	1..	Anatolien	Anatolien	23	#REDIRECT [[Lüttasien]]
11	1..	Ankara	Ankara	324	'''Ankara''' is de [[Höööftstadt]] vun de [[Törkie]]. Is al ut de...
12	1..	Athen	Athen	3094	'''Athen''' (greeksch "Αθηναι" = Athinai) hett sien Naam v...

	src	dst
10	6	1644
11	6	191
12	6	178
13	6	268
14	6	300
15	6	5222
16	5	69
17	5	50461
18	5	19
19	5	1994
20	5	1982

	src	dst
1	6	20135
2	5	69
3	2	1865
4	11	40930
5	10	46935
6	9	1865
7	13	69
8	12	9162
9	15	129
10	14	88
11	16	7926
12	8	4012

5 Graph Databases

Graph databases are databases where data is represented as a graph. A *graph* contains *nodes* which are connected by *relationships*. Each node and relationship can have *properties* for storing unstructured information.



In comparison, a *relational database* management system (RDBMS) represent data as *columns* in *tables* and relationships are implemented via *foreign keys*. To navigate using foreign-keys, join operations are required. A join operation is cost-intensive because the RDBMS has to match two columns in order to properly link them.

A graph database is a NoSQL database which is specialized for highly connected information like social media (social graph), online retailer (customers who bought this item also bought), etc. Navigating relationships does not require any special operations as relationships are first class members of a graph database. No joins are required.

Just as *relational algebra* is the mathematical foundation of a RDBMS, *graph theory* is the foundation of a graph database. This allows graph databases to fully integrate and optimized graph algorithms like finding a shortest path, spanning tree or run Dijkstra's algorithm.

5.1 Neo4j

Neo4j is an implementation of a graph database used by well-known companies like Ebay, HP, Cisco, Walmart, etc. For our evaluation we used the Neo4j Community edition, which runs local on a single machine. For use in production the Neo4j Enterprise edition offers enterprise-grade availability, management and scale-up & scale-out capabilities.

To connect to a Neo4j database, a wide range of technologies are offered. These include the following:

- HTTP Post Requests
- Java
- .NET
- JavaScript
- Python

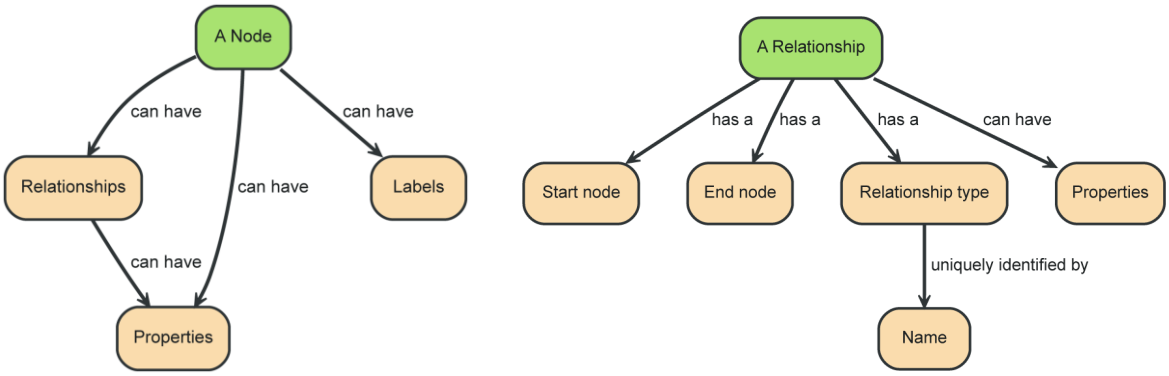
Even for less common languages like Clojure and Haskell, a Neo4j client is available.

Neo4j and its query language (see section Cypher) is very well documented online. The documentation contains general concepts of graph databases, the features and specifics of Neo4j, an easy to understand tutorial and a detailed reference about the Cypher query language.

5.1.1 Graph Model

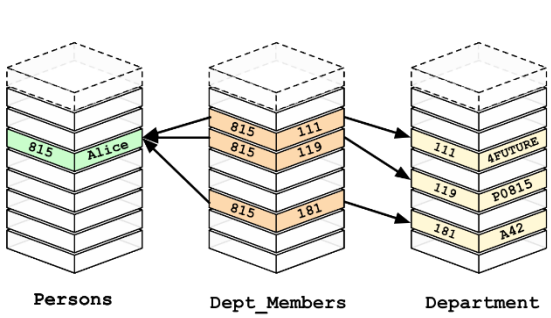
A Neo4j *graph* contains *nodes* and *relationships*. Both nodes and relationships can have *properties* where primitive values (boolean, int, float, …), arrays and strings can be stored.

A node can have properties, relationships and a label. With labels, nodes can be grouped into certain sets, for instance types of objects. A relationship has a start node, an end node, a relationship-type and can also have properties. The relationship-type is used similar to a label for nodes. Neo4j also specifies a *path*, which has a start and an end node and can contain one or more relationships.

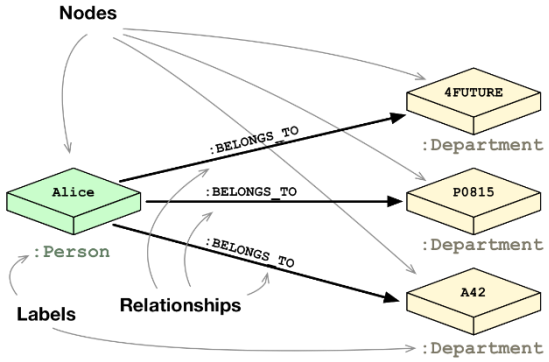


5.1.2 Data Representation

In Neo4j, each entity physically stores a list of relationships. Therefore instead of costly joins, Neo4j can navigate along these relationships in constant time. In addition Neo4j can materialize relationships into database structures for a more efficient access.



Storage of relationships in a RDBMS.



Storage of relationships in Neo4j.

5.2 Cypher

Neo4j provides an easy to use query language, called *Cypher*, which is used for creating, updating and deleting nodes, relationships and properties as well as for queries. Similar do SQL, Cypher is a declarative language where the required information is described and not the way it is obtained.

Cypher is designed to be a “humane query language” making it very natural and easy to use. The syntax of Cypher contains visual elements for relationships (arrows) to allow a visually appealing interpretation of the query.

5.2.1 CRUD Operations

For the creation of nodes and relationships in Cypher, the CREATE clause is used. A node is enclosed by parentheses and the properties within a nodes are enclosed by braces. Relationships are written as arrows (`-->`, `<--`) between matched nodes and the type and properties are enclosed by brackets. Despite Neo4j is storing both ends of a relationship physically enabling efficient navigations in both direction, a logical direction can be stated.

```
CREATE (p:Page {id:1, title:"Philosophy"})
CREATE (l:Page {id:2, title:"Logic"})
CREATE (l)-[:links_to]->(p);
```

The most important clause of Cypher is the MATCH clause which performs pattern matching. A matching node can have a name and a label and can specify properties that have to match. With the RETURN statement, all matching nodes are retrieved.

```
MATCH (p:Page {title:"Philosophy"})
RETURN p;
```

Properties of matched nodes can be changed and created with the SET clause. When setting a non-existing property, the property is created. Properties can be deleted with the REMOVE clause.

```
MATCH (p:Page {title:"Philosophy"})
SET p.test="Hello Graph"
REMOVE p.length;
```

Nodes can be delete with the DELETE clause.

```
MATCH (p:Page {title:"Philosophy"})
DELETE p;
```

With the WHERE clause, the matching node can be further restricted.

```
MATCH (p:Page)
WHERE p.length > 100
RETURN p;
```

5.2.2 Advanced Queries

Cypher supports advanced graph navigation in queries. Where in SQL, joins are necessary, Cypher provides an easy to use navigation using the arrow operator.

```
MATCH (p:Page {title:"Philosophy"})
MATCH (p)<--(a)
RETURN a;
```

Note that in the query above it is possible to only specify a single match and use the page-match and the link in a single MATCH statement. When combined in a single MATCH, the page will be matched multiple times, resulting in a less performant query.

Relationships can also be named and further restricted within brackets.

```
MATCH (p:Page {title:"Philosophy"})
MATCH (p)<-[l:links_to]-(a)
RETURN a, l;
```

When multiple “hops” are desired for a match, the multiplicity can be easily stated.

```
MATCH (p:Page {title:"Philosophy"})
MATCH (p)<-[l:links_to*1..3]-(a)
RETURN a, l;
```

Even arbitrary long paths are easily described. In SQL such queries require the definition of recursive statements, which are often hard to write, read and maintain.

```
MATCH (p:Page {title:"Philosophy"})
MATCH (p)<-[:links_to*]-(a)
RETURN a;
```

Typical predicates like ALL, ANY, EXISTS as well as aggregations like COUNT and statistic functions like SUM, AVG and MEAN are also supported.

```
MATCH (p:Page {title:"Philosophy"})
MATCH (p)<-[:links_to]-(a)
RETURN COUNT(DISTINCT a);
```

Neo4j also supports graph algorithms for complex graph evaluation. Among others, Neo4j supports

- shortest path,
- Dijkstra and
- A*.

For instance, the shortest path algorithm requires two matching nodes and a variable length relationship to obtain the shortest path between them.

```
MATCH path=shortestPath(
  (p:Page {title:"Philosophy"})
  <-[:first_links_to*]-
  (g:Page {title:"Graph"})
) RETURN path;
```

The Dijkstra and A* algorithm requires additional information, like the “costs-property”. In a RDBMS, these graph algorithms would be very hard or impossible to write.

6 Benchmarked queries

Each of the following queries is executed 10 times after the database server has started. The first query therefore runs without any cached results, resulting in a longer duration. In one test the results (all pages with title, text, etc.) of the query are transferred to the caller, while in the other test only the number of the pages are transferred. **All timings are measured in seconds.**

6.1 Find all pages that link to “Jesus”

Cypher

```
MATCH (p:Page {title:'Jesus'})
MATCH (p)
    <-[:links_to*1..1]-
      (a:Page)
RETURN COUNT(DISTINCT(a));

...

MATCH (p:Page {title:'Jesus'})
MATCH (p)
    <-[:links_to*1..5]-
      (a:Page)
RETURN COUNT(DISTINCT(a));
```

SQL

```
SELECT DISTINCT *
FROM (
SELECT a.*
FROM Page p
  JOIN Link l ON p.id=l.dst
  JOIN Page a ON a.id=l.src
WHERE p.title='Jesus'
 UNION ALL
...
 UNION ALL
SELECT a.*
FROM Page p
  JOIN Link l ON p.id=l.dst
  JOIN Link h1 ON h1.dst=l.src
  JOIN Link h2 ON h2.dst=h1.src
  JOIN Link h3 ON h3.dst=h2.src
  ...
  JOIN Page a ON a.id=h3.src
WHERE p.title='Jesus'
...
) a;
```

	MS-SQL (just count)						Neo4j (just count)				
Hops	1	2	3	4	5		1	2	3	4	5
(cold start) 1	0,380	0,552	0,653	1,043	3,356		0,459	0,525	0,718	3,916	> 600
2	0,038	0,103	0,212	0,419	2,969		0,041	0,071	0,089	1,969	> 600
3	0,037	0,105	0,211	0,417	2,999		0,039	0,071	0,088	2,870	> 600
4	0,036	0,103	0,210	0,418	3,304		0,039	0,066	0,089	1,854	> 600
5	0,036	0,103	0,210	0,419	2,862		0,038	0,055	0,105	1,861	> 600
6	0,036	0,104	0,211	0,423	2,875		0,039	0,049	0,088	1,900	> 600
7	0,036	0,105	0,208	0,414	2,876		0,039	0,052	0,088	1,763	> 600
8	0,037	0,101	0,211	0,422	2,899		0,039	0,050	0,087	1,925	> 600
9	0,038	0,103	0,209	0,417	2,891		0,039	0,042	0,095	1,827	> 600
10	0,037	0,102	0,212	0,436	2,967		0,051	0,040	0,088	1,903	> 600
average (run 2 - 10)	**0,037**	**0,103**	**0,210**	**0,421**	**2,960**		**0,040**	**0,055**	**0,091**	**1,986**	
stddev (run 2 - 10)	0,001	0,001	0,001	0,007	0,138		0,004	0,012	0,006	0,337	

	MS-SQL (with data)						Neo4j (with data)				
Hops	1	2	3	4	5		1	2	3	4	5
(cold start) 1	0,322	0,582	2,145	0,916	3,469		0,408	0,676	1,658	6,612	> 600
2	0,035	0,103	1,240	0,459	2,971		0,038	0,179	0,702	4,943	> 600
3	0,035	0,108	0,558	0,452	2,925		0,039	0,168	0,708	4,697	> 600
4	0,035	0,103	0,224	0,457	2,875		0,038	0,153	0,698	5,100	> 600
5	0,035	0,102	0,229	0,447	2,838		0,040	0,144	0,647	5,630	> 600
6	0,035	0,103	0,224	0,464	2,892		0,037	0,145	0,654	5,659	> 600
7	0,035	0,102	0,223	0,451	2,942		0,050	0,138	0,691	4,151	> 600
8	0,035	0,104	0,226	0,444	2,979		0,032	0,128	0,666	4,382	> 600
9	0,035	0,108	0,219	0,447	2,977		0,033	0,106	0,660	4,109	> 600
10	0,035	0,107	0,232	0,448	2,921		0,034	0,118	0,710	4,618	> 600
average (run 2 - 10)	**0,035**	**0,105**	**0,375**	**0,452**	**2,925**		**0,038**	**0,142**	**0,682**	**4,810**	
stddev (run 2 - 10)	0,000	0,002	0,343	0,007	0,049		0,005	0,023	0,025	0,576	

6.2 The number of pages that reach Philosophie by always following the first link

Cypher	SQL

`MATCH (p:Page {title:'Philosophie'})`
`MATCH (p)`
`<-[:first_links_to*]-`
`(a:Page)`
`RETURN COUNT(DISTINCT(a));`
	`WITH temp (id) AS (`
	`SELECT p.id`
	`FROM Page p`
	`WHERE p.title = 'Philosophie'`
	`UNION ALL`
	`SELECT fl.src`
	`FROM FirstLink fl`
	`JOIN temp t ON fl.dst = t.id`
	`)`
	`SELECT COUNT(DISTINCT(t.id)) - 1`
	`FROM temp t;`

	MS-SQL	Neo4j
(cold start) 1	0,296	2,167
2	0,033	0,075
3	0,032	0,068
4	0,032	0,048
5	0,032	0,046
6	0,032	0,050
7	0,032	0,039
8	0,032	0,038
9	0,032	0,038
10	0,032	0,047
average (run 2 - 10)	**0,032**	**0,050**
stddev (run 2 - 10)	0,000	0,013

6.3 Solutions on initial problem questions

On the Platdüütsch Wikipedia by the date of 2015-01-08, from 31814 articles

1. 33 articles lead to *Philosophie* by always following the first link and
2. 26986 articles lead to *Jesus* by using only up to 5 links inside the main texts.

7 Conclusion

In conclusion we can say that graph databases, especially Neo4j, are a very promising technology. Compared to relational databases, graph databases are much easier to understand and use. Writing queries in Cypher is a lot easier, even as a beginner, than writing relationship-intensive queries in SQL. Not only writing queries for a graph database, even thinking in graphs and relationships is very natural and easy.

Unfortunately the performance turned out to be not as good as expected. For a database system specialized on relationships performing worse than a relational database for relationship-intensive queries is pretty sad. We expected Neo4j to be a lot faster than MS-SQL. Maybe the query optimizer of Neo4j's community version is not as good as Microsoft's one, or our problem was not well suited for graph databases.

Nevertheless, with ongoing maturity graph databases are very capable and seem to be a promising alternative for traditional relational databases. Additionally, graph databases may scale better in a cloud environment, but no further investigations have been done on this subject.