NoSQL Databases

Daniel Rutz and Leon Schürmann (Eds.)

March 25, 2019

Contents

1 GraphDB

Thore Krüss, Lennart Purucker, Johanna Sommer

1.1 Motivation/Introduction

First paragraph to state the general context and make important points for the motivation and why its important. Refer to Wood Paper here.

But Graph Database research has its beginnings already in the early 90s. During this time, numerous proposals came up, describing a semantic network to store data about the database. That was, because contemporary systems were failing to take into account the semantics of a database. (letzten Satz ändern) The Locial Data Model [3] was proposed, trying to combine the advantages of relational, hierarchical and network approaches in that they modeled databases as directed graphs, with leaves representing attributes and internal nodes posing as connections between the data.
Similar to that, the Functional Data Model [6] was proposed with the same goal, focusing specifically on providing a conceptually natural database interface [1].
During this period, most of the underlying theory of Graph Databases was created. It was most likely because of insufficient hardware support for big graphs that this research declined, only to be picked up again now, powered by improved hardware. Todays focus in Graph Theory research lies primarily on actual practical systems and on the theoretical analysis of graph query languages [1].

Especially practical implementations of Graph Database Theory have gained traction, as real world problems are more often than not interrelated - hence graphs are extremely useful in unterstanding the wide diversity of real-world datasets.[5]
The emerging of social networks have naturally helped graph database models come up, with big players like Twitter and their implementation FlockDB entering the field. In those social network situations, a so-called social graph can effortlessly model attributes of a person as well as relationships between people. While in traditional RDBMS the apparent friend-of-a-friend-problem would be solved with a join over all relevant tables, in graph database technology this can be achieved with a traversal, which is far more cost inexpensive (letzten Teilsatz ändern) [4].

Another meaningful topic today are recommender systems, where most work focusing on optimizing machine learning algorithms. But also in database theory, this specific context poses challenges. However again, the graph model gracefully maps item similarities and correlations between user behaviour [2].

These application fields bring very distinct workloads that require specific query languages to process. There are two different kinds of workload: in social network transactions low-latency online graphs are processed while for example link analysis algorithms evaluate high-throughput offline graphs [1]. Many query language proposals have come up recently, differing mainly in the underlying graph data structure/model and the functionality provided [7].

A deeper description of the theory behind graph databases will be given in subsection 2, aiming to connect the data model to its fields of application as well as comparing it to RDBMS. This comparison will be picked up in subsection 3, where a use case from the SQL world will be applied to Graph Database Theory, focusing in particular on Neo4j. Lastly, our findings will be stated in subsection 4 with a general conclusion.

1.2 Graph Database Theory

1.2.1 Description of data model and functionality

1.2.2 fields of application

1.2.3 CAP Theorem

1.2.4 Comparison: Graph Databases and Relational Databases

As the comparison between Graph Databases and Relational Databases is a known field, a lot of literature exists on this topic already. Throughout the comparisons, the two methods are always assessed under the same aspects: performance, flexibility, security and maturity.
For those comparison points it makes sense to focus on specific implementations of the technologies, hence in this section Neo4j will be chosen as a concrete precedent for Graph Databases, whereas MySQL will be the example implementation for Relational Databases.

performance

Detailed surveys on performance of both technologies already exist in literature, for example from Vicknais et al. [**?**]. In this specific instance, MySQL version 5.1.42 and Neo4j-b11 were compared. The queries chosen for the experiments were similar to types that are used in real world provenance systems. Typically in this scenario, for one dataset/node one traverses the graph to find its origin. Another use case in this context is, if a data object or node is deemed incorrect, this information needs to be propagated to all its descendants/child nodes [**?**].
Further on, the queries were partitioned into structural queries referencing the graph but not the payload itsself, and data queries using the actual payload data. It is important to note that the payload data in this case was integer payload data, as different types are handled seperately depending on framework.
In the traversal queries, Neo4j clearly outperformed MySQL, sometimes even being faster by the factor of 10. Though that was expected, as Relational Databases are not designed for traversals. MySQL for this part of experiment falls back to a standard Breath First Search, which is not optimal for this scenario. Neo4j on the other hand has a built in framework for traversals, making it superior in terms of performance for the traversal queries [**?**].
Contrary to that, in the data queries MySQL turned out to be more efficient. This result was partly due to the fact that Neo4j uses Lucene for querying, which treats all payload as text, eventhough in this scenario the payload is of type integer. But also when the payload changes to text, MySQL had better performance in the experiments [**?**].
The researchers also took into account a special case for the experiments, trying the data queries with payload that is closer to actual real world data - text with spaces inbetween the words. Surprisingly, at a large enough scale Nao4j outperformed MySQL by a large amount for those queries.

flexibility

The flexibility aspect compares both database technologies in their behaviour when taken out of the environment that they were created for [**?**].
For Relational Databases an uncommon environment would for example be ad-hoc data schemes that change with time, whereas for Graph Databases a less typical dataset would be one without many connections between the individual nodes [**?**]. MySQL is optimized for a large-scale multi-user environment, hence trying to use it for smaller applications comes with a large overhead of functionality that has to be implemented with it but that may not even be needed for this specific application. Neo4j is typically targeted towards more lightweight applications, but manages to scale really well, having a scalable architecture that also accounts for speed [**?**]. Its easily mutable schema makes it more flexible with data types that are rather untypical for Graph Databases.

Security

Neo4j does not have built in mechanisms for managing security restrictions and multiple users [**?**]. MySQL on the other hand natively supports multiple users as well as access control lists. [**?**]

Maturity

it makes sense to talk about the database implementations in general now. Maturity refers both to how old a particular system is and to how thoroughly tested it is [**?**]. It is important to note taht all relational databases - including mysql- use the same query language SQL, which means that support is equal over all rdbms implementations. Support for one implementation is applicable to all others. [**?**] Neo4js version 1.1 was released in February 2010. While neo4j is a for-profit framework and has decent support from its parenty company, this does not apply to all graph database implementations [**?**]. Furthermore, the query languages differ from implementation to implementation, separating them in that aspect and not sharing support??.

1.3 Implementation with Neo4j

1.3.1 Use Case from the SQL world

1.3.2 Installation

1.3.3 modelling

1.3.4 usage, query language

1.3.5 short conclusion, summary

1.4 Reflection

1.4.1 alternative popular graphdbs

1.4.2 conclusion

reflect on advantages disadvantages with implementation references

Bibliography

[1] Renzo Angles and Claudio Gutierrez. An introduction to graph data management. In *Graph Data Management*, 2018.

[2] Zan Huang, Wingyan Chung, Thian-Huat Ong, and Hsinchun Chen. A graph-based recommender system for digital library. In *Proceedings of the 2Nd ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '02, pages 65–73, New York, NY, USA, 2002. ACM.

[3] Gabriel Kuper. The logical data model : a new approach to database logic /. 09 1985.

[4] Justin J. Miller. Graph database applications and concepts with neo4j. 2013.

[5] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

[6] David W. Shipman. The functional data model and the data language daplex. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 59–59, New York, NY, USA, 1979. ACM.

[7] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41:50–60, 2012.