NoSQL Databases

Daniel Rutz and Leon Schürmann (Eds.)

April 16, 2019

Contents

Glossary

API Application Programming Interface 12

CRUD Create, Read, Update, Delete 7, 10, 20, 22

FOOBAR fully operational organization building acronym references 4

IAM identity and access management 14

IT Information Technology 14

MDM Master Data Management 13

NoSQL *No SQL* or *Not only SQL*. NoSQL is a loosely defined term, grouping different databases which either do not only support data accesses via the Structured Query Language (SQL), or do not support it at all. 5, 9, 10

OGM object-graph mapping 20, 22

OLTP online transaction processing 7

ORM object-relational mapping 20

RDBMS relational database management system 5, 6, 10

SQL Structured Query Language 6, 11, 16, 20–23

1. Test

Foo, Bar

First use: fully operational organization building acronym references (FOOBAR). Second use: FOOBAR.

2. Graph Databases – Neo4j

Thore Krüss, Lennart Purucker, Johanna Sommer

2.1. Abstract

This chapter of the book gives an overview of Graph Databases as part of the NoSQL landscape, focusing on Neo4j as a specific implementation. The goal of this work is to give a timely overview of Graph Databases today as well as assessing recent events and additions to this technology. The reader will be given a comprehensive introduction to the field and can find suggestions on how Graph Databases can help easier model data structures and in which scenarios it is superior to relational database models.

After a detailed theoretical presentation of Graph Theory and how it is applied to Graph Databases, a comparison to relational database management system (RDBMS) as well as the prevalent advantages of Graph Databases are given. Next, Graph Databases in practice are shown by the example of Neo4j, giving a comprehensive overview about setup and characteristics specific to this implementation. After a summarizing conclusion about Neo4j, an overall reflection of Graph Databases including personal experience and possible future work closes this chapter.

2.2. Introduction

The hype around Graph Databases in todays NoSQL-landscape can not be disregarded. The popularity for Neo4j has been steadily increasing and with its connection-first approach and close to reality data model Neo4j has been gaining fans from all over the database community ("Neo4j: Product", n.d.).

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 consider the semantics of a database. The Logical Data Model (Kuper, 1985) 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 (Shipman, 1979) was proposed with the same goal, focusing specifically on providing a conceptually natural database interface (Angles & Gutierrez, 2018).

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. Today's focus in Graph Theory research lies primarily on actual practical systems and on the theoretical analysis of graph query languages (Angles & Gutierrez, 2018).

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 understanding the wide diversity of real-world datasets (Robinson, Webber, & Eifrem, 2013).

The emerging of social networks has naturally contributed to the development of graphical database models, 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 (Miller, 2013).

Another meaningful topic today are recommender systems, where most work focuses on optimizing machine learning algorithms. This specific context also poses challenges in database theory. However again, the graph model gracefully maps item similarities and correlations between user behaviour (Huang, Chung, Ong, & Chen, 2002).

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 (Angles & Gutierrez, 2018). Many query language proposals have come up recently, differing mainly in the underlying graph data structure/model and the functionality provided (Wood, 2012).

A deeper description of the theory behind graph databases will be given in subsection 2.3, 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 2.4, where an implementation example will be given, focusing in particular on Neo4j and also explaining how an Structured Query Language (SQL) example would be transformed to fit Graph Databases. Lastly, our findings will be stated in subsection 2.5 with a general conclusion. Since the topic of Graph Databases contains extensive theory, this chapter of the book will explain the theory and Neo4j in equal parts, to give an easy-to-understand introduction into the topic.

2.3. Graph Database Theory

A graph database is a unique type of database designed to store data without transforming it into predefined structural models, whereby accessing and storing of relationships between data is as important as accessing and storing the data itself ("Neo4j Website: What is a Graph Database?", n.d.). Graph databases offer Create, Read, Update, Delete (CRUD) methods as an online, operational database management system. They focus on operation availability, transactional performance and integrity. Thus, graph databases are usually incorporated into online transaction processing (OLTP) systems (Eifrem, Webber, & Robinson, 2015).

A graph database can be implemented with different concepts, non-native or native, for storage and request processing. Additionally, a data model must be chosen. The most common graph models are property graphs, hypergraphs and triples (Eifrem et al., 2015). For this eBook, the (labeled) property graph model will be examined because it is the most popular model in industry practice (Eifrem et al., 2015) and the theoretical foundation of Neo4j (Lal, 2015).

2.3.1. Description of Data Model and Functionality

Before the explanation of the property graph model, a short recap of graphs is needed. There is no need for general graph theory, like search algorithms, to understand graph databases (Eifrem et al., 2015).

Graph Basics

A graph is a theoretical structure which represents a set of entities and their relationships, whereby entities are represented by nodes (vertices) and relationships by links between nodes (edges) (Eifrem et al., 2015; Lal, 2015). One-way relationships, like being the parent of someone, are represented as directed edges. On the contrary, two-way relationships, like being married to someone can be represented as two directed edges between both related nodes. Some literature tends to represent bidirectional (two-way) relationships as one undirected edge (e.g. an edge without arrows). It is more appropriate to use two directed edges because this is closer to an actual implementation where two physical pointers would exist. See figure 2.1 for an example.
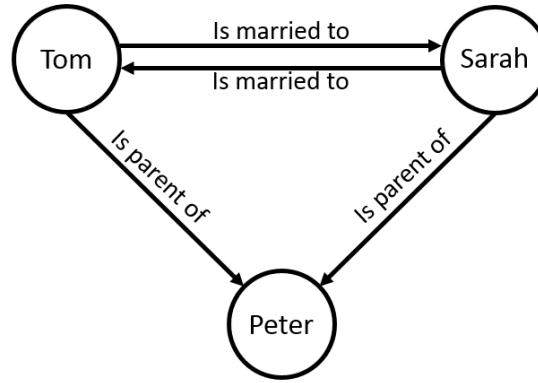
Figure 2.1.: A graph representing a small family. Tom, Sarah and Peter are entities and thus represented as nodes (circles). The two-way relationship between Tom and Sarah, their marriage, is depicted as two directed edges. Lastly, Tom and Sarah are the parents of Peter and thus both have the relationship "Is parent of" directed towards Peter.

The property graph model

The (labeled) property graph model is based on the theoretical graph from above (Lal, 2015). It increases the overall information that a normal graph can store. Two such extensions, as the model name illustrates, are additional labels for each node and properties for nodes and edges. Figure 2.2 shows a labeled property graph.
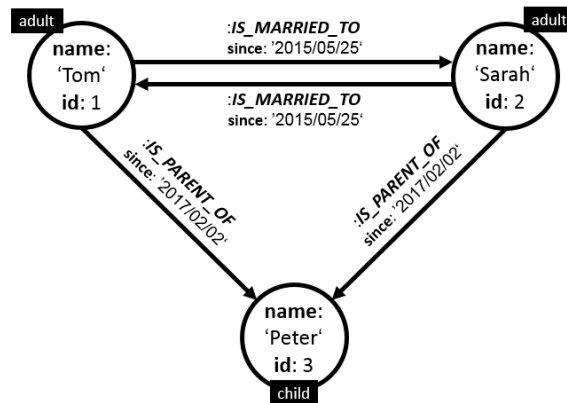

Figure 2.2.: A labeled property graph representing a small family (Eifrem, Webber, & Robinson, 2015; Lal, 2015). Compared to figure 2.1 additional information can been stored.

Concepts of the property graph

Nodes: Like a normal graph, the property graph represents entities as nodes. The nodes in figure 2.2 are the circles of Tom, Sarah and Peter. Each node can have any number of properties (e.g.: "name: 'Tom'" and "id: 1") and multiple labels (e.g. "adult") ("Neo4j Website: What is a Graph Database?", n.d.).

Labels: Nodes are tagged with labels which describe the role of the node within the system (Lal, 2015). In figure 2.2, the white text on black rectangles, "adult" and "child", are labels. Labels can also add metadata (constraints and indices) to the node (Lal, 2015; "Neo4j Website: What is a Graph Database?", n.d.).

Properties: Properties are attributes (key-value pairs) of nodes or relationships. They are used to sore further information about the entity or relationship (Lal, 2015). Each bold and non-italic written word in figure 2.2 is a key of a key-value pair property (e.g. "name:", "id:", "since:"). The content that follows the colon (e.g. "Peter", "3", "2017/02/02") is the value.

Relationships: Again, relationship are depicted as directed edges (links) between nodes. Relationships must have a name as well as a start and end node (Eifrem et al., 2015) . In figure 2.2, `"IS_PARENT_OF"` and `"IS_MARRIED_TO"` (the italic and bold written words) are the names of relationships. The text below ("since: [. . .]") is the property of the relationship. Arrows indicate the direction of the relationship. In practice, the direction is ignored and navigation through each edge (relationship) is possible (Lal, 2015; "Neo4j Website: What is a Graph Database?", n.d.). Generally, relationships store cost quantities according to the usage of the system (e.g. distance, ratings, etc.) ("Neo4j Website: What is a Graph Database?", n.d.).

Storage

"Storage deals with how the data is stored physically and how it is represented logically when retrieved"(Lal, 2015)

One of the main tasks of a graph database system is to traverse relationships. As mentioned earlier, storing and accessing the relationships in a graph database is crucial. They are stored explicit (per directed edge) instead of being inferred from stored attributes (like primary/foreign keys in a relational model) (Lal, 2015) . Thus, any kind of storage system needs to be able handle the relationships explicitly. Graph databases can either use native graph storage, systems that are built for storing and accessing graphs, or non-native graph storage, systems that store adequately transformed graph data in relational or non-graph NoSQL databases (Eifrem et al., 2015).

Non-native graph storage

Non-native graph storage transforms the data from the graph model (e.g. property graph) into relational or other non-graph NoSQL database models (document-oriented, etc.). When accessing the data, the responsible system must rebuild (infer) the relationships at runtime. This is mostly done by a query engine which is responsible for executing incoming queries and thus fetching or changing the data (CRUD methods) (Lal, 2015) . In the case of a RDBMS, the query engine would first make the relationships explicit by inferring them through utilizing foreign keys and join-statements before returning or processing the data. This preprocessing results in more costly operations and inefficient traversing of relationships (Lal, 2015). Non-native graph storage exists because it allows the use of well known, mature and well documented databases like MySQL (Eifrem et al., 2015).

Native graph storage

The key aspect of native graph storage is that it does not rely on actual indexes. The relationships between nodes within the graph are "natural adjacency"(Eifrem et al., 2015) indices. Thus, the nodes are stored in such a way that they are physically linked to each other on the disk. This is called "index-free adjacency"(Eifrem et al., 2015), which is in practice done by pointers. Accordingly, searching for a specific information in a native graph storage is implemented by traversing through pointers. This causes queries on native storage to be highly efficient compared to non-native storage which uses join-statements and index lookups (Eifrem et al., 2015) .

2.3.2. Advantages of Graph Databases

Graph databases offer substantial advantages when working with connected data (Lal, 2015). Its performance, flexibility and agility are the key differences to other databases (Eifrem et al., 2015). The following section will take a closer look at these three advantages. More details and references to actual research data on these advantages can be found in the section Comparison: Graph Databases and Relational Databases (2.3.4).

Performance

A graph database has much higher query processing performance compared to relational and other NoSQL databases. This advantage becomes more and more prevalent as the size/amount of stored data grows. In the relational world more data would mean a higher join-intensity and thus worse performance. Whereas in the graph database world the performance remains to be almost constant even for an exponential increase of size. This is the cases because queries are being restricted to parts of the graph (e.g. a subgraph) which contains the information of interest. Therefore, querying only needs

time proportional to size of the subgraph and not to the size of the whole graph (Eifrem et al., 2015; Lal, 2015).

Flexibility

The flexibility of a graph databases must be understood in the context of the graph database model. The model is flexible and so are graph databases. Furthermore, this flexibility is most apparent for an actual operational graph database in production environments.

The paradigm of fitting data to predefined data models, as in SQL, is neither efficient nor desired by developers. Instead fitting an easily extensible and changeable data model to newly emerging data is more appropriate for fields of graph database applications (See more in the section "Fields of Application", 2.3.3). Thereby the process of designing a complex and mostly final data model at the start of the database implementation, a point in time where it is impossible to predict all kinds of data that might be needed in the future, gets replaced by designing a basic data model with the expectation to change it in the future (Eifrem et al., 2015; Lal, 2015).

This process is natively supported by graph databases. All components of a graph model (e.g. for the property graph model: nodes, properties, labels and relationships) can be added to an existing model without invalidating queries already in use. This concept of graph databases also minimizes maintenance cost and risk because the need for migrations (e.g. the equivalent of schema migrations for a relational database) is reduced (Eifrem et al., 2015; Lal, 2015).

Agility

In today's agile software development world, where developers need to focus on a certain task for a short time before switching to a different task, software tools that fit this iterative approach are more favorable. Databases that offer data models which can grow with new data meet this requirement. Additionally, databases that offer data models which do represent the data closer to its actual format (e.g. not transferring it into tables) are also more favorable because they reduce the time between design and implementation which is appropriate for the short time a developer may have to implement a database. Lastly, modern test-driven development requires agile databases to be easily testable. (Eifrem et al., 2015; Lal, 2015).

A graph database is "schema-less" (Lal, 2015). It does not transform data (e.g. normalization in the SQL world) but rather tries to represent the data as close to its actual

format as possible. Furthermore, the Application Programming Interface (API) and query language design of graph database increase testability (Eifrem et al., 2015) . Finally, the flexibility of its data model enables the database to evolve with new data. As a result, a graph database has the characteristics to be agile software.

2.3.3. Fields of Application

When reading through use cases described by marketing teams of graph database management systems (for example: Neo4j ("Neo4j Website: Why Graph Database?", n.d.; Robinson; n.d.)), it may feel like any problem could be solved with a graph database. Solving any problem with a graph database may be possible but this fact alone is not a valid reason to do so.

Instead, cost efficiently, compliance with company polices, available developer skills and available time are the primary reasons to choose a graph database for a specific use case. Additionally, replacing existing well-working and established database management systems should have major and urgently needed advantages (Eifrem et al., 2015).

Graph databases can create these advantages for use cases which handle connected data. Below are some short examples of large companies that use graph databases. Subsequently, the top five use cases from the perspective of the graph database management system Neo4j are explained.

Enterprise Use Case Examples

Social Networks: Twitter, Facebook and LinkedIn use graph databases to manage user information and feed of users. This contains information like updates from friends, news and potential posts of relevance or interest (e.g. Jobs for LinkedIn users) (Lal, 2015).

Routing: Prominent navigation services like Google Maps, TomTom and Sygic utilize graph databases for map navigation (Lal, 2015).

Search: Google (Google Knowledge Graph) and Facebook (Facebook Social Graph) are also using graph databases for storing the connection of content for search functionalities (Lal, 2015).

Recommendation: Walmart and eBay are both using graph databases and value their performance for real-time product recommendations (Robinson; n.d.).

Neo4j Use Cases

Fraud Detection

Graph databases are well fit for fraud detection because good detection mechanisms need to analyze the relationships between data. In detail, if the relationships between certain data objects is conspicuously high, the risk of fraud is very high. As an example, take an E-commerce system. A normal user would use one or two credit cards to buy products. A fraudster would use a lot of different credit cards which are most likely stolen. This relationship density between users, credit cards and purchased products is modeled by a graph database and is therefore easily measurable and observable (Robinson; n.d.).

As mentioned before, graph databases are built for storing and rapidly traversing relationships, thereby supporting advanced detection mechanisms that need to perform relationship analysis between a lot of data.

Real-Time Recommendation Engines

A real-time recommendation engine can only be as effective as the database it is using because the engine needs information about the existence, quality and strength of data relationships. Information must either be computed from the database in a time-consuming manner or made available natively, as with graph databases (Robinson; n.d.). Graph databases model this information without any additional computation. Existence is modeled by edges (relationships), quality and strength by key-value pairs (properties) of edges.

In addition, the need to easily add and combine data (e.g. user behavior, demographics and their purchase history) and then analyze this new dataset in real time for possible recommendations is crucial for such an engine (Robinson; n.d.). A graph database supports simple addition and combination with its already mentioned flexibility. Furthermore, the performance advantage of graph databases in this context is again very favorable when analyzing this new dataset.

Master Data Management

In a company, master data is data such as users, customers, products, accounts, partners, sites and business units. Identifying, cleaning, storing and governing this data is called Master Data Management (MDM). Best practice for Master Data Management (MDM) is to create one master data store which contains the data of the entire company. As a result, any business application that might create or use this data only uses only the same storage system. Hence, the master data store is one storage system for a lot of different applications which still needs to fully function in real time and might need to adapt to new business requirements. Thus, it must provide a purpose-built, dynamic

and sometimes unconventional data model (Robinson; n.d.). These requirements are perfectly matched by the flexibility and agility of a graph database.

Network and Information Technology Operations

The structural representation of an Information Technology (IT) infrastructure network is a graph. Consequently, it should not be a surprise that a graph database is a good solution to model, store and serve requests for an IT infrastructure environment. Information in an IT infrastructure environment are for example device configurations, infrastructure interdependencies, any kind of event (log files, error messages, etc.) and administrative details. Systems that use such information can, in the event of a failure, inform the right administrator about what went wrong where and when in real time (Robinson; n.d.).

A graph database is not only able to model the network in its native representation but is also able to add all this information to its storage with ease. Network administrators are nodes with connection to their devices and field of responsibility (subgraphs). Configurations are properties of device nodes. Any interdependencies are represented by relationships. Lastly, events are nodes linked to the device that created it. This requirement for a native data model and sufficient performance is fulfilled by a graph database.

Identity and Access Management

The process of deciding which identity can access which resource is called identity and access management (IAM). This decision process also needs to utilize information about identities (e.g. administrators, users), resources (e.g. files, devices) and rules (e.g. "user X can access file Y") that must be stored somewhere. Conventional storage options, like directory services or application specific solutions, tend to be unsuitable because they cannot manage the required complex interconnection structures of big organizations or are too slow for bigger datasets (Robinson; n.d.). Whereas a graph database is a valid solution because its agility allows the developer to easily model the complex structures and its performance does not slow down for bigger datasets.

2.3.4. Comparison: Graph Databases and Relational Databases

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.

It is important to note that literature comparing the two is already rather old and there are no comparisons done on newer versions. There is no new version of mySQL, but two new releases for Neo4j. Those have included a new query engine and performance optimizations. It is hence expected that if such a comparison were to be done again today, the performance results of Neo4j would improve.

Performance

Detailed surveys on performance of both technologies already exist in literature, for example from Vicknair et al. (Vicknair et al., 2010). 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 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 (Vicknair et al., 2010).

Further on, the queries were partitioned into structural queries referencing the graph but not the payload itself, 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 separately depending on the 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 Breadth 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 (Vicknair et al., 2010).

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, even though in this scenario the payload is of type integer. But also when the payload changes to text, MySQL had better performance in the experiments (Vicknair et al., 2010). Lucene has since been dropped and replaced with Cypher in Neo4j version 3.

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 in between 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 (Vicknair et al., 2010).
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 (Jaiswal, 2013). 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 ("Neo4j Website: Why Neo4j? Top Ten Reasons", n.d.). 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 in their community edition (Jaiswal, 2013), but the fee-based enterprise edition provides such functionality. MySQL on the other hand natively supports multiple users as well as access control lists. ("MySQL Website: Security in MySQL", n.d.)

Maturity

For the comparison under the aspect of maturity it makes sense to talk about database implementations in general. Maturity refers both to how old a particular system is and to how thoroughly tested it is (Vicknair et al., 2010). Since all Relational databases - including MySQL - use the same query language SQL, support is equal over all implementations and support for one implementation is applicable to all others (Jaiswal, 2013). Neo4js version 1.1 was released in February 2010. While Neo4j is a for-profit framework and has decent support from its parent company, this does not apply to all graph database implementations (Vicknair et al., 2010). Furthermore, the query languages differ from implementation to implementation, separating them in that aspect and making support for one implementation not applicable to another one.

2.4. Implementing a Graph Database Model

This section shall outline the general approach, how to convert an existing relational database model into a property graph model. In the second part an introduction to Neo4j, the implementation of a database model in Python and basic querying in an application and with Neo4js own query language „Cypher" will follow.

2.4.1. Converting a Relational Database Model

There are a couple of guides available describing how to build a database model for Neo4j. Since the database itself is schema-less, multiple schemas can be used and implemented at the same time. Nevertheless an application needs a model of the data. Neo4j states in ("Neo4j Website: Model: Relational to Graph", 2019) that it is possible to transfer almost all existing relational models into a graph model. The general approach has been described in (Hunger, n.d.). The first step in this conversion is to consider the names of all Non-JOIN-Tables as labels. Foreign keys will become relations. JOIN-Tables will be converted into relations as well with additional properties added to the relation ("Neo4j Website: Concepts: Relational to Graph", 2019). The rows will be converted into nodes connected by edges based on the formerly converted relations. Attributes not covered in the previous steps will become properties of a node.

2.4.2. Implementing a Sample Project with Neo4j

In this section the modeling and setup with Neo4j will be lined out and a sample project be described utilizing available Python-Libraries to implement a sample project. There are two different available versions of Neo4j. First of all there is the Open Source community edition which is published under the GPL. Additionally Neo4j Inc sells licenses for an enterprise edition ("Neo4j Website: Neo4j Subscriptions", n.d.) including support and several additional features, such as replication, multiple users, several query performance optimizations and no limitation for the number of nodes in the database (the community version is limited at 34 Billion nodes). The following examples have been implemented and tested with the community edition.

Setting up Neo4j

Neo4j is available for Windows, Linux and Mac and can be installed via the provided packages. While there is a minimum requirement of 2GB of RAM, Neo4j recommends

16GB or more ("The Neo4j Operations Manual v3.5", 2019, Chapter 2.1). Since Neo4j is implemented in Java, starting it can be done by invoking it with a Java runtime of choice installed. The default configuration does not need to be modified to get started.

High availability The community edition of Neo4j does not allow to set up a cluster of multiple Neo4j instances. This feature is reserved for the enterprise version. The setup of a causally consistent cluster is explained in the documentation ("The Neo4j Operations Manual v3.5", 2019, Chapter 5.1). The reference architecture as shown in Figure 2.3 recommends an odd number of at least three „Core Servers" connected into a RAFT-Cluster handling mostly write requests. They ensure consistency of the data. Connected to this core cluster may be an arbitrary number of „Read Servers". They only handle the – sometimes resource costly – read requests but are not relevant to the clusters integrity. Information from Core Servers is replicated asynchronously to the read replicas.
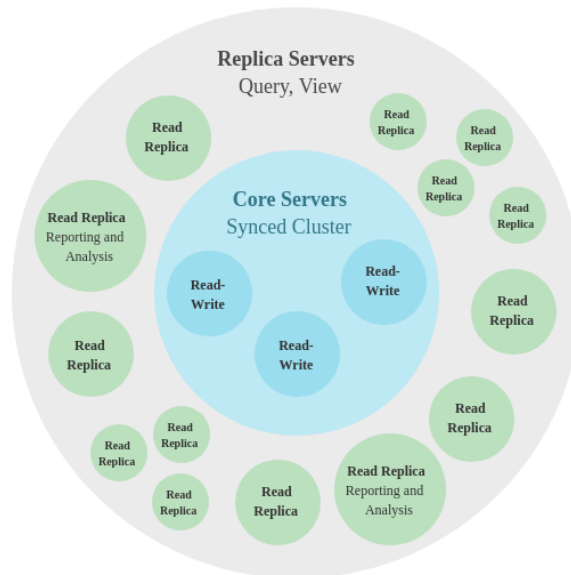

Figure 2.3.: Neo4j causal Cluster Architecture ("The Neo4j Operations Manual v3.5", 2019, Chapter 5.1)

Classification of Neo4j within the CAP Theorem For this kind of classification it is again necessary to differentiate between the community and the enterprise edition. Since it is not possible to set up a Neo4j-Cluster with the community edition it can not be considered a distributed system. Therefore the CAP-Theorem is not applicable (Mehra, 2017).

As mentioned in section 2.4.2 the enterprise edition can be set up as a causal cluster. Causal consistency though does not fulfill the criteria Brewer put out for consistency (Kleppmann, 2015a) so it can not be considered as „C" under the CAP Theorem. It is important to note that a causal cluster is still ACID compliant. Due to the nature of the core servers using a consensus-based protocol (RAFT) availability in case of a network partition only applies to the majority of the cluster (Penchikala, 2016). This does make them partition tolerant though, fulfilling all criteria for „P".

According to Michael Hunger, one of the Neo4j developers, the causal cluster architecture can be considered as a „CP" system (Penchikala, 2016). Considering the the concerns lined out by Martin Kleppmann in his blog post (Kleppmann, 2015b) Neo4j should be considered as „P" – if no alternative to CAP is considered as proposed by him (Kleppmann, 2015a).

Modeling the graph database

The sample project maps the relations within users in a social network. Figure 2.4 outlines a data model with circles representing nodes. They are connected by edges showing their relationships. For this example labels are represented as colours. In this sample model there may be an arbitrary number of persons (orange), who may be friends with other persons. Additionally, they can share interests (green) and may be a member of a group (purple). Furthermore they can state from which country (yellow) they are from.
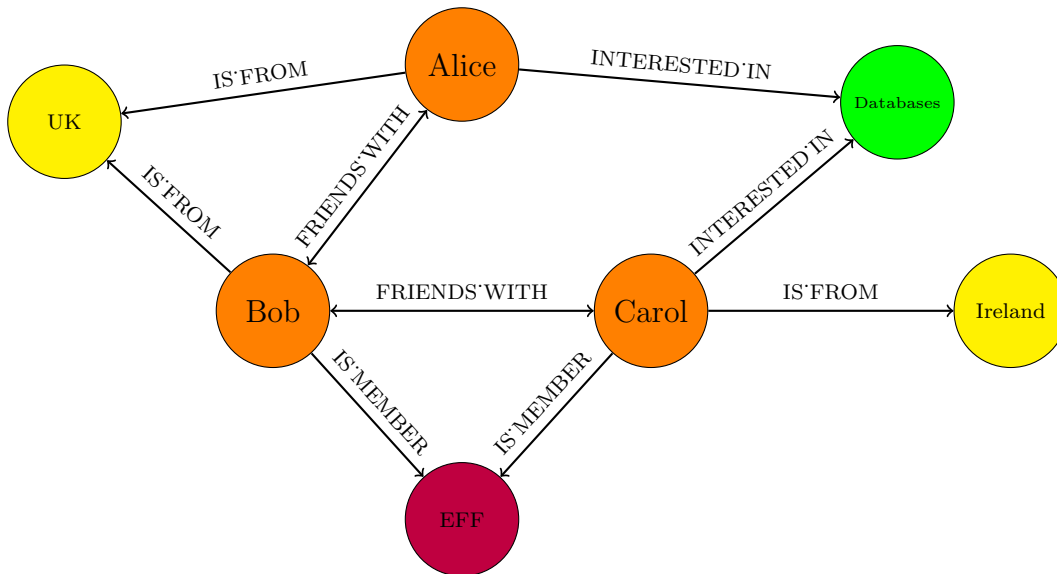

Figure 2.4.: Sample database schema

Implementation in Python

While it is possible to manage the database utilizing the CRUD-functionality from Neo4js own query language Cypher (see 2.4.2) this is not really suitable for an application. Developers familiar with object-relational mappings (ORMs) such as Hibernate for Java or SQLAlchemy for Python would prefer to define the different nodes and relations in classes providing the database elements as objects and abstracting actual SQL-Queries.

For Python there exists a community driven project called neomodel (Anastasiou, 2019) aiming to provide an object-graph mapping (OGM) for Python projects. Neomodel is published under the MIT License.

```python
class Partnership(StructuredRel):
    since = DateTimeProperty(
        default=lambda: datetime.now(pytz.utc)
    )


class Country(StructuredNode):
    name = StringProperty(unique_index=True, required=True)


class Interest(StructuredNode):
    name = StringProperty(unique_index=True, required=True)


class Group(StructuredNode):
    name = StringProperty(unique_index=True, required=True)


class Person(StructuredNode):
    uid = UniqueIdProperty()
    name = StringProperty(unique_index=True)
    age = IntegerProperty(index=True, default=0)

    # traverse outgoing relations
    country = RelationshipTo(Country, 'IS_FROM')
    interests = RelationshipTo(Interest, 'IS_INTERESTED_IN')
    groups = RelationshipTo('Group', 'IS_MEMBER')
    friends = Relationship('Person', 'FRIENDS_WITH', model=Partnership)
```

Listing 1: Example graph database model with neomodel

The implementation of the graph model mentioned in Figure 2.4 in Python has been realized in Listing 1 following the neomodel documentation (Edwards, 2019). With this model it is possible to create new nodes in the database by instantiating a new object of the given classes as in Listing 2.

```
lmeitner = Person(name='Lise Meitner', age=89)
lmeitner.save()
```

Listing 2: Creating a new person node in the database

To connect two nodes it is necessary to get both objects and to invoke the `connect()` method as seen in Listing 3 on one of them. The `get_or_create` method simplifies creation of nodes with no additional properties since it either returns an already existing node or creates it.

```
country = Country.get_or_create({'name': 'Austria'})
lmeitner.country.connect(country[0])
```

Listing 3: Connecting a person and a country node

Retrieving one or more existing nodes can be done by filtering as shown in Listing 4.

```
curie = Person.nodes.filter(name='Marie Curie')
```

Listing 4: Querying for a person node by the name attribute

Queries using Cypher

Neo4j provides its own query language Cypher. It is developed with an open source specification called openCypher ("openCypher – About", 2018). Thus it should be possible to use the same query language for graph processing in other databases – such as SAP HANA or Redis. Its syntax is oriented on SQL statements though there are quite some differences to better match with a graph model. Neo4j has an extensive introduction how to use Cypher ("Neo4j Website: Cypher Query Language", 2019). Therefore only a short introduction should be given here.

Cypher uses two basic patterns. First of all there are nodes, denoted by enclosing parentheses. The second pattern is used for relationships. They are expressed by two dashes and may have a direction utilizing the greater-than/less-than signs. Furthermore, the type of a relationship may be specified in brackets between the two dashes.

A few examples will make it easier to understand how these patterns can be used.

The simplest query would be to get all nodes and all relations between them without regard to their labels. This can be achieved by calling

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

It is important to note that Cypher uses `MATCH` as a keyword similar to SQL's `SELECT`. Contrary to SQL it is necessary in Cypher to `RETURN` the previously matched nodes to obtain them in the result. It is possible to declare the label of a node by calling

```
MATCH (p:Person) RETURN p
```

This would reduce the output to all persons and the relations between them.

As mentioned before Cypher supports a pattern for relations. To include them in a query – e.g. for all Persons who have one or more friends the query would look like

```
MATCH (a:Person) -[:FRIENDS_WITH]- (b:Person) RETURN a, b
```

The relationship type in the brackets may be ommited to get all types of relations between these nodes.

Similar to SQL, Cypher also supports a `WHERE` statement. To query for a specific Person where the attribute `name` equals „Otto Hahn" and all nodes connected to this person the Cypher query would look like this

```
MATCH (p:Person)-[r]-(n) WHERE p.name = 'Otto Hahn' RETURN p, r, n
```

Of course Cypher offers a complete keyword set for all types of CRUD operations. Interested readers should follow the introduction by Neo4j ("Neo4j Website: Cypher Query Language", 2019).

2.4.3. Conclusion

Getting started with Neo4j is relatively simple. There is plenty of documentation available helping to implement a database model and an application based on it. Especially the OGM Projects for Python are pretty advanced and suitable for production usage. Users familiar with SQL will find Cypher not that difficult to get used to.

There are two major downsides to Neo4j. The first one is the memory footprint. A newly set up instance already consumes more than 600MB of RAM – in comparison, a PostgreSQL instance storing a couple hundred MB of data still consumes less than half of that. The second downside is, that many features – especially regarding maintenance and clustering – are preserved for the enterprise edition and not available in the open source community edition. This makes it difficult if not impossible to use the community edition in a production environment.

2.5. Reflection

2.5.1. Alternative Graph Databases

OrientDB

OrientDB is one of the biggest competitors to Neo4j, developed by Callidus Software Inc. (owned by SAP) and published under Apache-2 License ("OrientDB Website: OrientDB vs Neo4j", n.d.). Like Neo4j it is implemented in Java. OrientDB is a multi model graph database, just as Neo4j, but also a document oriented database allowing relations between documents ("OrientDB Website: Why OrientDB", n.d.). As a query language OrientDB uses SQL with a custom dialect to include features for traversals ("OrientDB Website: OrientDB vs Neo4j", n.d.). In comparison to Neo4j they claim to be a lot faster ("OrientDB Website: OrientDB vs Neo4j", n.d.). This claim is based on a paper (Dayarathna & Suzumura, 2012), which has been released in 2012. As mentioned in section 2.3.4 Neo4j has reimplemented their query engine since then.

A major selling point is the possibility to set up a highly-available (multi-master) cluster of multiple nodes with the Open Source community version ("OrientDB Website: Support and Subscriptions", n.d.; "Setting up a Distributed Graph Database", n.d.).

2.5.2. Conclusion

Even though there is extensive literature on the topic of Graph Databases, our group was overall dissatisfied with the scientific resources we found. Most publications were written by the same few people, not providing a distinct enough reflection on the topic. Furthermore, while there were many publications around 2010 on this topic, literature did not provide updates or added benchmarks of e.g. comparisons with other database systems. We aim to close that gap with an updated summary on today's Graph Database theory and Neo4j.
In this chapter, we first gave an introduction into Graph Databases, providing an overview of its history. Next, the basic underlying theory was explained and assessed critically. A report on the implementation with Neo4j was given, stating its distinct characteristics and concluding its value as a Graph Database implementation.

To conclude, we would also like to share our personal experience of working with Neo4j and Graph Databases in general. Overall, we feel that with the concept of Graphs as a database model one can easier map real life datastructures than with just a relational structure. While the underlying theory is rather complex, we felt that there were sufficient resources to give a simple introduction into the topic. The implementation was

enjoyable, as the documentation for Neo4j was easily understandable and it did not take long until the basic setup was complete. We especially enjoyed using community driven libraries; when issues arose we were given an answer and help immediately, making our experience overall very pleasant.

Future work to extend this paper could include assessing the enterprise edition. We were unable to compare the community edition to the enterprise edition due to stellar pricing. The fee-based version allows for clustering which would have been interesting to take into account for our implementation. In addition, a deeper evaluation of alternatives like OrientDB could be valuable, especially since OrientDB is an open source project and provides clustering functionality in its community version. Lastly, Hypergraph and Triplet propose interesting approaches to graph modeling and an assessment of the differences and strengths would be valuable for the current literature landscape.

Bibliography

Anastasiou, A. (2019). Neomodel. Retrieved March 29, 2019, from https://github.com/neo4j-contrib/neomodel/blob/master/README.rst

Angles, R., & Gutierrez, C. (2018). An introduction to graph data management. In *Graph data management.*

Dayarathna, M., & Suzumura, T. (2012). Xgdbench: A benchmarking platform for graph stores in exascale clouds. In *4th IEEE international conference on cloud computing technology and science proceedings, cloudcom 2012, taipei, taiwan, december 3-6, 2012* (pp. 363–370). doi:10.1109/CloudCom.2012.6427516

Edwards, R. (2019). neomodel - Getting started. Retrieved March 25, 2019, from https://neomodel.readthedocs.io/en/latest/getting_started.htm

Eifrem, E., Webber, J., & Robinson, I. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.

Huang, Z., Chung, W., Ong, T.-H., & Chen, H. (2002). A graph-based recommender system for digital library. In *Proceedings of the 2nd acm/ieee-cs joint conference on digital libraries* (pp. 65–73). JCDL '02. doi:10.1145/544220.544231

Hunger, M. (n.d.). From relational to graph: A developer's guide. Retrieved March 25, 2019, from https://dzone.com/refcardz/from-relational-to-graph-a-developers-guide

Jaiswal, G. (2013). Comparative analysis of relational and graph databases. *IOSR Journal of Engineering, 03*, 25–27. doi:10.9790/3021-03822527

Kleppmann, M. (2015a). A critique of the CAP theorem. *CoRR, abs/1509.05393.* arXiv: 1509.05393. Retrieved April 2, 2019, from http://arxiv.org/abs/1509.05393

Kleppmann, M. (2015b). Please stop calling databases CP or AP. Retrieved April 2, 2019, from http://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html

Kuper, G. (1985). The logical data model : A new approach to database logic /.

Lal, M. (2015). *Neo4j graph data modeling.* Packt Publishing - ebooks Account.

Mehra, A. (2017). Understanding the CAP Theorem. Retrieved April 2, 2019, from https://dzone.com/articles/understanding-the-cap-theorem

Miller, J. J. (2013). Graph database applications and concepts with neo4j.

MySQL Website: Security in MySQL. (n.d.). Retrieved March 25, 2019, from https://dev.mysql.com/doc/mysql-security-excerpt/5.7/en/

Neo4j Website: Concepts: Relational to Graph. (2019). Neo4j Inc. Retrieved March 25, 2019, from https://neo4j.com/developer/graph-db-vs-rdbms/

Bibliography

Neo4j Website: Cypher Query Language. (2019). Neo4j Inc. Retrieved March 30, 2019, from https://neo4j.com/developer/cypher/

Neo4j Website: Model: Relational to Graph. (2019). Neo4j Inc. Retrieved March 25, 2019, from https://neo4j.com/developer/relational-to-graph-modeling/

Neo4j Website: Neo4j Subscriptions. (n.d.). Retrieved March 30, 2019, from https://neo4j.com/subscriptions

Neo4j Website: What is a Graph Database? (n.d.). Retrieved April 2, 2019, from https://neo4j.com/developer/graph-database/

Neo4j Website: Why Graph Database? (n.d.). Retrieved April 2, 2019, from https://neo4j.com/top-ten-reasons/

Neo4j Website: Why Neo4j? Top Ten Reasons. (n.d.). Retrieved March 25, 2019, from https://neo4j.com/top-ten-reasons/

Neo4j: Product. (n.d.). Retrieved April 7, 2019, from https://neo4j.com/product/

openCypher – About. (2018). Retrieved March 28, 2019, from http://www.opencypher.org/

OrientDB Website: OrientDB vs Neo4j. (n.d.). Retrieved April 5, 2019, from https://orientdb.com/orientdb-vs-neo4j/

OrientDB Website: Support and Subscriptions. (n.d.). Retrieved April 5, 2019, from https://orientdb.com/support/

OrientDB Website: Why OrientDB. (n.d.). Retrieved April 5, 2019, from https://orientdb.com/why-orientdb/

Penchikala, S. (2016). Neo4j 3.1 Supports Causal Clustering and Security Enhancements. Retrieved April 2, 2019, from https://www.infoq.com/news/2016/12/neo4j-3.1

Robinson; J. W. (n.d.). *White paper: The Top 5 Use Cases of Graph Databases.* Neo4j Inc. Retrieved April 2, 2019, from https://go.neo4j.com/rs/710-RRC-335/images/Neo4j_Top5_UseCases_Graph%20Databases.pdf

Robinson, I., Webber, J., & Eifrem, E. (2013). *Graph databases.* O'Reilly Media, Inc.

Setting up a Distributed Graph Database. (n.d.). Callidus Software Inc. Retrieved April 5, 2019, from https://orientdb.com/docs/last/Tutorial-Setup-a-distributed-database.html

Shipman, D. W. (1979). The functional data model and the data language daplex. In *Proceedings of the 1979 acm sigmod international conference on management of data* (pp. 59–59). SIGMOD '79. doi:10.1145/582095.582105

The Neo4j Operations Manual v3.5. (2019). Neo4j Inc. Retrieved March 30, 2019, from https://neo4j.com/docs/operations-manual/current/

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th annual southeast regional conference* (42:1–42:6). ACM SE '10. doi:10.1145/1900008.1900067

Wood, P. T. (2012). Query languages for graph databases. *SIGMOD Record, 41,* 50–60.

A. Cassandra query example

Create database, table, insert and select data

```
CREATE KEYSPACE people
    WITH REPLICATION =
      { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
USE people;

CREATE COLUMNFAMILY users (
    username varchar PRIMARY KEY,
    name varchar,
    lastname varchar,
    email varchar,
    age int
);

INSERT INTO users (username, name, lastname, email)
    VALUES ('john', 'John', 'Smith', 'john@gmail.com');
INSERT INTO users (username, name, lastname, age)
    VALUES ('jack', 'Jack', 'Sparrow', 33);
INSERT INTO users (username, name, lastname, email, age)
    VALUES ('kate', 'Kate', 'Austen', null, 25);

SELECT * FROM users;

 username | age  | email          | lastname | name
----------+------+----------------+----------+------
     kate |   25 |           null |   Austen | Kate
     john | null | john@gmail.com |    Smith | John
     jack |   33 |           null |  Sparrow | Jack

$ nodetool flush

$ sstabletump /var/lib/cassandra/data/people/*/mc-1-big-Data.db
[
  {
```

```
    "partition" : {
      "key" : [ "kate" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 45,
        "liveness_info" : { "tstamp" : "2019-04-14T16:21:05.014317Z" },
        "cells" : [
          { "name" : "age", "value" : 25 },
          {
            "name" : "email",
            "deletion_info" :
              { "local_delete_time" : "2019-04-14T16:21:05Z" }
          },
          { "name" : "lastname", "value" : "Austen" },
          { "name" : "name", "value" : "Kate" }
        ]
      }
    ]
  },
  {
    "partition" : {
      "key" : [ "john" ],
      "position" : 46
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 98,
        "liveness_info" : { "tstamp" : "2019-04-14T16:21:04.982207Z" },
        "cells" : [
          { "name" : "email", "value" : "john@gmail.com" },
          { "name" : "lastname", "value" : "Smith" },
          { "name" : "name", "value" : "John" }
        ]
      }
    ]
  },
  {
    "partition" : {
      "key" : [ "jack" ],
      "position" : 99
```

```
      },
      "rows" : [
        {
          "type" : "row",
          "position" : 144,
          "liveness_info" : { "tstamp" : "2019-04-14T16:21:05.002672Z" },
          "cells" : [
            { "name" : "age", "value" : 33 },
            { "name" : "lastname", "value" : "Sparrow" },
            { "name" : "name", "value" : "Jack" }
          ]
        }
      ]
    }
]
```