# NoSQL Databases

Daniel Rutz and Leon Schürmann (Eds.)

April 29, 2019

# Contents

**Glossary** **4**

**1. Test** **6**

Foo, Bar

**2. Key Value Introduction** **7**

Vanessa Jörns, Tobias Schiffmann and Victor Veal

**3. Hazelcast** **9**

Vanessa Jörns, Tobias Schiffmann and Victor Veal

3.1. Hazelcast Introduction . . . . . 9
3.2. Specification . . . . . 10
3.3. Hazelcast and the CAP theorem . . . . . 11
3.4. Implementation . . . . . 12
3.5. Cheat Sheet . . . . . 15
3.6. Conclusion . . . . . 17

**4. Key-value databases** **19**
4.1. Redis . . . . . 19

Laura Khaze, Leon Schürmann

4.1.1. Primary characteristics . . . . . 19
4.1.2. Data Types . . . . . 21
4.1.3. Multi-node setups / Redis Cluster . . . . . 23
4.1.4. Typical use cases . . . . . 25
4.1.5. Conclusion . . . . . 27
4.2. The Riak Key-Value Store . . . . . 28

Daniel Rutz, Paul Thore Flachbart

4.2.1. Introduction . . . . . 28
4.2.2. Characteristics of Riak . . . . . 29
4.2.3. Placement inside CAP Theorem . . . . . 29
4.2.4. Advantages and Disadvantages . . . . . 30
4.2.5. Comparison to Redis . . . . . 30

4.2.6. Test Implementation . . . . . 31
4.2.7. Conclusion . . . . . 31

**5. Graph Databases – Neo4j** **32**

Thore Krüss, Lennart Purucker, Johanna Sommer

5.1. Abstract . . . . . 32
5.2. Introduction . . . . . 32
5.3. Graph Database Theory . . . . . 34
5.3.1. Description of Data Model and Functionality . . . . . 34
5.3.2. Advantages of Graph Databases . . . . . 37
5.3.3. Fields of Application . . . . . 39
5.3.4. Comparison: Graph Databases and Relational Databases . . . . . 41
5.4. Implementing a Graph Database Model . . . . . 44
5.4.1. Converting a Relational Database Model . . . . . 44
5.4.2. Implementing a Sample Project with Neo4j . . . . . 44
5.4.3. Conclusion . . . . . 49
5.5. Reflection . . . . . 50
5.5.1. Alternative Graph Databases . . . . . 50
5.5.2. Conclusion . . . . . 50

**A. Cassandra query example** **56**

# Glossary

**API** application programming interface 26, 28, 39

**ASCII** American Standard Code for Information Interchange 21

**BSD** Berkeley Software Distribution 19

**CAP theorem** The CAP theorem (also called Brewer's theorem) states, that in a distributed database system, it is not possible to achieve more than two characteristics out of *consistency*, *availability* and *partition tolerance* Brewer, 2000. 19

**CPU** central processing unit 20

**CRUD** Create, Read, Update, Delete 28, 34, 37, 47, 49

**DBMS** database management system 26

**DDR4** double data rate 4 20

**ECC** error correcting code 20

**FOOBAR** fully operational organization building acronym references 6

**HDD** hard disk drive 20

**HTTP** HyperText Transfer Protocol 28, 30

**IAM** identity and access management 41

**IOMMU** input/output memory management unit 20

**IT** information technology 41

**JPEG** Method for compression of image data developed by the *Joint Photographic Experts Group*. 22

**MDM** Master Data Management 40

**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. 19, 28, 32, 36, 37

**OGM** object-graph mapping 47, 49

**OLTP** online transaction processing 34

**ORM** object-relational mapping 47

**RAM** random access memory 20

**RDBMS** relational database management system 32, 33, 37

**RESP** Redis serialization protocol 21

**SATA** serial ATA 20

**SQL** Structured Query Language 28, 33, 38, 43, 47–50

**SSD** solid state drive 20

**TCP** transmission control protocol 21, 24, 27

# 1. Test

Foo, Bar

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

# 2. Key Value Introduction

Vanessa Jörns, Tobias Schiffmann and Victor Veal

Key-value databases or key-value stores are a classification of NoSQL databases. The idea of key-value stores is to collect a key for every data set. Each set that is stored in the database can be accessed by the key. Therefore the key needs to be distinct, whether in a namespace or in the whole system. The database system has no pattern for the values which is why it is not necessary to know about the type of the values that are stored. This feature enables easy storage of any kind of data like serialized structures, XML, text data, files... (Kudraß, 2015).

However, there are also disadvantages of this database management type. In terms of operational actions like querying through the data, as one would do in relational database management systems, key-value databases only provide simple operations like get, put and delete. As a result of this constraint, data querying must be handled at the application level.

Another difference to relational databases are the use cases. For simple applications, which only require a system that is able to store and manage data (e.g. update entries, join tables), is capable of representing real-world entities and describes relationships, relational databases should be used. Key-value databases should rather be chosen if the application or the system requires a good performance since key-value solutions are faster than relational database systems (Mendis, n.d.).

For instance, an online application that is only responsible to enable quick access to a profile, does not need to interact with entries of the profile itself. It should rather guarantee that the user of the application can easily find the profile by providing the corresponding key. To enable a quick access of the profile, it would be enough to only search for the unique ID of the profile instead of querying across several attributes of the data set.

Nevertheless, not all key-value solutions are similar designed. There are a lot of different systems today. In the following chapters some database management systems will be introduced. The focus will be on providing the reader a quick overview about the different systems, how they distinguish from each other and how to use them.

The first system that will be explained is Hazelcast which gives an overview of the the basic characteristics including a cheat sheet with needed commands. The next section talks about Redis and the basic features as well as in-memory computing. Last but not least the Key Value chapter is finished with a section about Riak.

# 3. Hazelcast

Vanessa Jörns, Tobias Schiffmann and Victor Veal

## 3.1. Hazelcast Introduction

Hazelcast is a company that is developing an in-memory computing platform consisting of Hazelcast IMDG, Hazelcast Jet and Hazelcast Cloud. "Hazelcast Jet is an embeddable, distributed computing platform for fast processing of big data sets". It is built on the foundation of Hazelcast IMDG on which this chapter focuses ("Hazelcast Jet", n.d.).

An in-memory data grid (IMDG) is a rather new concept where data is stored in the main memories of a computing cluster. One of the main aspects is the ability to automatically scale and rebalance the cluster when decreasing or increasing in size. Here the data is partitioned equally across the cluster nodes. IMDGs are usually used for implementing distributed and scalable applications since they provide distributed versions of the basic data structures (Tasci & Demirbas, 2015). Hazelcast IMDG is open source and implemented in Java. However, there are also existing API's for C/C++, .NET, REST, Python, Go and Node.js.

As Hazelcast is designed to be lightweight and easy to use, it can be downloaded as a compact library (JAR) and can be used by simply adding this JAR file to the classpath. With Java as the only dependency there is no need to install any software ("Hazelcast IMDG Reference Manual", n.d.).

Firstly, the features of Hazelcast are specified and distinguished from other key-value solutions. The next section talks about the CAP theorem and how it applies to Hazelcast. After that a short manual for implementing Hazelcast for own applications is provided. This inlcudes a basic set up as well as a configuration. For reference a cheat sheet with the most important commands is included. In the end, the whole chapter about Hazelcast is concluded and the advantages and disadvantages are analyzed.

## 3.2. Specification

Hazelcast consists of many same or similar features like other key value databases and IMDGs. However, there are some major features which describe Hazelcast's distinctive strengths. The first and one of the main characteristics is that Hazelcast completely computes in-memory rather than storage based. This makes Hazelcast extremely fast but also volatile.

Another major feature is that Hazelcast relies on clustering with the approach of a "masterless nature" of the nodes. This means that each node in the cluster has the exact same functionality and operates in a peer-to-peer manner (Johns, 2015). So unlikely other NoSQL databases, there is no master and slave hence there is no single point of failure. All nodes are responsible for the same proportion of processing and storing ("Hazelcast IMDG Reference Manual", n.d.). The oldest node of the cluster automatically becomes the "de facto leader" and manages the distribution of the data for joining nodes. Since the data is redistributed for every joining or exiting node, the cluster rebalances automatically and thus makes Hazelcast simple to set up and configure.

As Hazelcast consists of the basic features of an in-memory data grid, the data and therefore the load is equally spread across the cluster. Here each node is the owner and holds a number of partitions of the overall data. Therefore, saturation of a cluster can simply be overcome by adding more nodes to the cluster. The cluster is then rebalanced and the load for each node decreases. This means scaling is easy and fast which makes Hazelcast suitable for handling big amounts of data. Nevertheless, since Hazelcast persists data entirely in-memory, it has the main drawback that data will be lost with a node shutting down. To prevent the overall cluster of losing the data a failing node has held, Hazelcast distributes backups of each data partition among multiple other members. This means in case of a node shut down, another node will have a backup of the data and the cluster can be rebalanced without suffering any data loss and downtime (Johns, 2015).

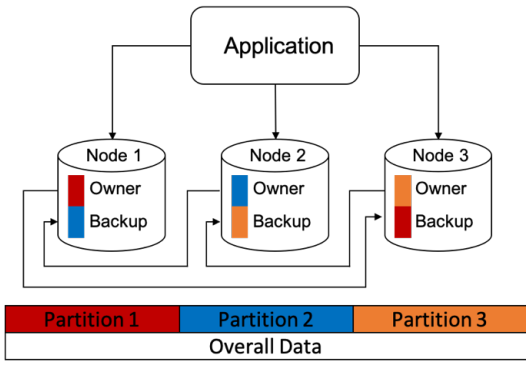


Figure 3.1.: Hazelcast Nodes (Johns, 2015)

This illustration shows the cluster structure as described above. It explains how the data is partitioned in equal parts and spread across the cluster as well as the replication of the data for backup. This is just a simplified version of the structure with only three nodes. In practice each node would be responsible for multiple subsets of the data and not just one data partition. So, for instance in order to get the data of Partition 1, the application has to communicate with Node 1. The distribution of the data is dynamic and which node is responsible for which subset of data usually changes over time. Hence, the allocation is an internal operational detail and the application as well as the user usually does not need to know it. Moreover, Hazelcast supports various distributed collectors, features and processors. Besides storing the data in-memory distributed on many nodes, it is possible to load data from diverse sources into varying structures, communicate across the cluster by sending messages and to use the stored data for analytical processing (Johns, 2015).

## 3.3. Hazelcast and the CAP theorem

As Hazelcast enables data storage for distributed systems, it may be interesting how the CAP theorem applies to it. According to the chapters before, Hazelcast offers a storage mechanism that distributes data across several nodes. Therefore the first aspect out of the CAP theorem is network partitioning. According to the article *Jepsen Analysis on Hazelcast 3.8.3* (Luck, n.d.), Hazelcast is a PA system which means that it favours availability over consistency. Due to the fact that data is partitioned, the problem of keeping the data consistent over the whole system occurs. For example, if the user inserts a new entry to a cluster, the whole systems needs to update this info so that the same information can be provided from all nodes. This problem of having several nodes storing inconsistent data is called split brain in an In-Memory Data Grid system.

Hazelcast offers a method that should avoid split brain. In the so called "Split Brain Protection" a minimum number of nodes is set on which write or read operations are prevented. If a split brain happens, any sub-clusters that have a lower number of nodes than the minimum number are prevented from accepting write operations. Nevertheless, this method only reduces the time of inconsistency. Therefore it does not completely avoid the inconsistent state of the system (Luck, n.d.).

Recently, the Hazelcast company has announced the ability to provide a solution that supports both PA (availability over consistency) and PC (constistency over availability). This feature should allow the user to adapt more flexibly to the requirements of the application. So far, there are too less resources that can describe this new feature of Hazelcast which is why this paper is currently not able to report about it ("What CAP Theorem Means to a Business Leader", n.d.).

## 3.4. Implementation

### Getting Started

As already mentioned Hazelcast is designed to be easy to use and therefore only require a few steps to set it up running. First of all, the Hazelcast package has to be downloaded, for example from the official website ("Hazelcast Downloads", n.d.). The package is offered in a compressed data format and has to be extracted afterwards.

Hazelcast does not need any software installations. It is written in Java and therefore the platform to run Hazelcast on needs to be able to execute Java code. To do so a Java Development Kit has to be ready to use which can be gathered from the Oracle website. After ensuring Java code to run, Hazelcast is ready to be used.

The console application is an easy way to get in touch with Hazelcast and experience the software. It can be started by executing the `console` scripts in the `demo/` directory from a terminal. Hazelcast creates a new member which will either create a new cluster or will join a cluster if there already exists one. The cluster can than be filled by typing the commands as the following example shows.

```
hazelcast[default] > m.put foo bar
null

hazelcast[default] > m.get foo
bar

hazelcast[default] > m.entries
foo : bar
Total 1

hazelcast[default] > m.remove foo
bar

hazelcast[default] > m.size
Size = 0
```

Figure 3.2.: Hazelcast Basic Commands (Johns, 2015)

When executing the scripts in another window, Hazelcast shows how the new member

joins the cluster and displays the two members and there addresses after a short period of time.

```
Members [2] {
    Member [127.0.0.1]:5701
    Member [127.0.0.1]:5702 this
}
```

Figure 3.3.: Example Member List (Johns, 2015)

Now the data is spread across the members. They both own a particular partition of the data and store the other part as backup.

When closing one terminal to test Hazelcast's reaction to a cluster node failure, the remaining member tries to reestablish the connection to the closed one, but without success. The terminal will than display the process of repartitioning the cluster and prints a statement when it finishes.

## Members and Clients

Besides the console application Hazelcast also provides the opportunity to include the Hazelcast package in your code. For Java there is a package for members and one for clients provided. Other programming languages only have clients to work with. The difference between clients and members is that clients do not hold data. They connect to Hazelcast cluster members and access the data on that way for read and write operations ("Hazelcast IMDG Reference Manual", n.d.). If Hazelcast has to be set up for an application in C++ for example, the scripts in the `demo/` directory can be used to start a cluster member without needing to create a java application. The C++ application than has to include the Hazelcast package and can access or create the data via the member.

## Configuration

Hazelcast offers an XML file to adjust certain configurations in the `bin/` directory. It is possible to change the amount of backups and also the types of backups. There are normal backups which are synchronized and therefore lock the data in case it is

manipulated. All other nodes have to wait until the data is changed in all backups and the changes are confirmed by the nodes which hold the backups. Additionally, there are asynchronous backups. They do not lock any data and therefore bring a performance increase because the nodes do not have to wait for them to confirm the changes. On the other hand, it brings the risk of inconsistent data. Nodes could access data which are no longer valid because the changes were not made on all backups after a manipulation took place.

In general, increasing the number of backups will increase the read performance because the data can be read on different nodes in parallel. The costs for this advantage are either bad write performance in case of normal backups or inconsistency in case of asynchronous backups.

Furthermore Hazelcast provides configurations about how big a particular data structure can grow and how to act when there is no more space left.

```
<map name="capitals">

  <max-size policy="PER_NODE">10</max-size>
  <eviction-policy>LFU</eviction-policy>
  <eviction-percentage>20</eviction-percentage>

  <backup-count>1</backup-count>
  <async-backup-count>1</async-backup-count>

  <time-to-live-seconds>86400</time-to-live-seconds>
  <max-idle-seconds>3600</max-idle-seconds>

</map>
```

Figure 3.4.: Hazelcast Configuration (Johns, 2015)

In this example the map called *"capitals"* has a maximum size of 10 items per node. When reaching this maximum, 20 percent of the data will be freed according to the principle of Least Frequently Used (LFU). One synchronous and one asynchronous backup will be created and the time to live for data sets is set. This means that data will be deleted after this amount of seconds goes by. In contrast to this, setting the maximum idle time will only delete a data set when it is not accessed after a certain amount of seconds. Johns and the Hazelcast websites provide more information about configuration and go more in detail.

# 3.5. Cheat Sheet

## General commands

| | |
|---|---|
| `echo true\|false` | turns on/off echo of commands (default false) |
| `silent true\|false` | turns on/off silent of command output (default false) |
| `# <number> <command>` | repeats `<number>` time `<command>`, replace $i in `<command>` with current iteration (0...`<number-1>`) |
| `& <number> <command>` | forks `<number>` threads to execute `<command>`, replace $t in `<command>` with current thread number ((0...`<number-1>`)) |
| `jvm` | displays info about the runtime |
| `who` | displays info about the cluster |
| `whoami` | displays info about this cluster member |
| `ns <string>` | switch the namespace for using the distributed queue/map/set/list `<string>` (default value = "`default`") |
| `@ <file>` | executes the given `file` script. Use '//' for comments in the script |

## Queue commands

| | |
|---|---|
| `q.offer <string>` | adds a string object to the queue |
| `q.poll` | takes an object from the queue |
| `q.offermany <number> [<size>]` | adds indicated number of string objects to the queue ('`obj<i>`' or `byte[<size>]`) |
| `q.pollmany <number>` | takes indicated number of objects from the queue |
| `q.iterator [remove]` | iterates/displays the queue, remove if specified |
| `q.size` | adds a string object to the queue |
| `q.clear` | clears the queue |

## Set commands

| | |
|---|---|
| `s.add <string>` | adds a string object to the set |
| `s.remove <string>` | removes the string object from the set |
| `s.addmany <number>` | adds indicated number of string objects to the set ('`obj<i>`') |
| `s.removemany <number>` | takes indicated number of objects from the set |
| `s.iterator [remove]` | iterates/displays the set, removes if specified |
| `s.size` | size of the set |
| `s.clear` | clears the set |

## List commands

| | |
|---|---|
| `l.add <string>` | adds string to the list |
| `l.add <index> <string>` | adds string at a certain index |
| `l.contains <string>` | checks if list contains `<string>` |
| `l.remove <string>` | removes `<string>` from list |
| `l.remove <index>` | removes element at `<index>` from list |
| `l.set <index> <string>` | replaces value at `<index>` with `<string>` |
| `l.iterator [remove]` | iterates/displays the list |
| `l.size` | size of list |
| `l.clear` | clears list |

## Map commands

| | |
|---|---|
| `m.put <key> <value>` | puts an entry to the map |
| `m.remove <key>` | removes the entry of given key from the map |
| `m.get <key>` | returns the value of given key from the map |
| `m.putmany <number> [<size>] [<index>]` | puts indicated number of entries to the map ('`key<i>`':`byte[<size>]`, `<index>+(0..<number>)`) |
| `m.removemany <number> [<index>]` | removes indicated number of entries from the map ('`key<i>`', `<index>+(0..<number>)`) |
| `m.keys` | iterates/displays the keys of the map |
| `m.values` | iterates/displays the values of the map |
| `m.entries` | iterates/displays the entries of the map |
| `m.iterator [remove]` | iterates/displays the keys of the map, remove if specified |
| `m.size` | size of the map |
| `m.localSize` | local size of the map |
| `m.clear` | clears the map |
| `m.destroy` | destroys the map |
| `m.lock <key>` | locks the key |
| `m.trylock <key>` | tries to lock the key and returns immediately |
| `m.trylock <key> <time>` | tries to lock the key within given seconds |
| `m.unlock <key>` | unlocks the key |
| `m.stats` | shows the local stats of the map |

### MultiMap commands

| | |
|---|---|
| `mm.put <key> <value>` | puts an entry to the multimap |
| `mm.get <key>` | returns the value of given key from the multimap |
| `mm.size` | size of the multimap |
| `mm.clear` | clears the multimap |
| `mm.destroy` | destroys the multimap |
| `mm.iterator [remove]` | iterates the keys of the multimap, remove if specified |
| `mm.keys` | iterates/displays the keys of the multimap |
| `mm.values` | iterates/displays the values of the multimap |
| `mm.entries` | iterates/displays the entries of the multimap |
| `mm.lock <key>` | locks the key |
| `mm.trylock <key>` | tries to lock the key and returns immediately |
| `mm.trylock <key> <time>` | tries to lock the key within given seconds |
| `mm.unlock <key>` | unlocks the key |
| `mm.stats` | shows the local stats of the map |

## 3.6. Conclusion

Hazelcast is categorized as a key-value NoSQL solution, but since it is an in-memory data grid there are some main features that should be emphasized which set Hazelcast apart from the ordinary key-value databases. First of all, one of the key characteristics is its simplicity. As mentioned above, Hazelcast's only dependency is Java and therefore it can be used by simply downloading the JAR file and including it in the classpath. Furthermore, the cluster is structured as a peer-to-peer network, meaning there is no master-slave relation which is usually common for NoSQL databases. Each node is responsible for the same amount of data.

Another characteristic is the speed of Hazelcast since it relies on in-memory computing ("Hazelcast IMDG Reference Manual", n.d.). Knowingly in-memory computing also comes with two main downsides: volatility and scalability. Hazelcast, however addresses these issues. Volatility is solved by keeping the data of the nodes redundant. This means that Hazelcast stores the data of each node on multiple members. So, if one member fails, there is a backup and the whole cluster can be rebalanced and there is no overall loss of the data. Scalability is achieved by just adding more nodes to the cluster, the data is then automatically rebalanced and the work load for each member decreases.

These main features of Hazelcast also directly conclude some major advantages. To summarize the already mentioned ones, Hazelcast is very easy and fast to install and

it is designed to provide fast computing. Additionally, since there is no master-slave concept, there is also no single point of failure. It is easy to scale either up or down and redundant data storage protects from unexpected data loss ("In-Memory NoSQL with Hazelcast IMDG", n.d.). Furthermore, in contrast to ordinary key-value databases, Hazelcast is designed for a distributed environment and therefore it is possible to provide an unlimited number of maps and caches per cluster. Another advantage is that Hazelcast can be implemented using multiple threads and thus benefits from all available CPU cores ("Redis Replacement", n.d.).

Nevertheless, Hazelcast relying entirely on in-memory processing still comes with the drawback that this kind of storage is temporary. So, in case there is an overall system shut down the data is lost since the backups are stored in the same cluster. In addition, RAM is usually expensive which should be kept in mind when considering scalability.

Regarding the CAP theorem, as discussed above, Hazelcast can be either implemented as an PA or PC system. When using Hazelcast as a PA system, it is neglecting consistency. Meaning the system is not fully consistent all the time. On the other hand, in case of a PC system, it fails to provide continuous availability. Furthermore, as a PC system the speed of the data grid system relies on the slowest node. If backups are kept synchronously, data is locked until the consistent state is achieved again (Johns, 2015). The other nodes will have to wait until this particular block of data is unlocked again.

Hazelcast provides several and detailed manuals online as well as understandable tutorials which makes it easy to adapt Hazelcast for own applications. However, scientific research and benchmarking is very limited.

# 4. Key-value databases

## 4.1. Redis

Laura Khaze, Leon Schürmann

Redis is a key-value data store. It was invented by Salvatore Sanfilippo in April 2009 and is released under the Berkeley Software Distribution (BSD) 3-clause (*new / revised*) license. Therefore, it is a free and open source software product. Redis – originally an acronym for *remote dictionary server* – is primarily used as a database, caching-solution or a publish/subscribe message broker.

Redis stands out in the field of key-value data stores because of its simplicity and speed. A part of the high performance can be attributed to the use of in-memory data structures, while the use of C – a low-level systems programming language – provides some advantages as well. Because of its unique properties, Redis is very popular. According to `db-engines.com`, it is currently on rank seven of the most popular databases, and on rank one of all measured key-value data stores ("DB-Engines Ranking", n.d.).

The goal of this section is to show the primary characteristics and available data types of Redis. Clustered Redis setups will be evaluated in the context of the CAP theorem. Finally, typical usage scenarios for this software will be evaluated, and the most important facts are reiterated in the conclusion.

### 4.1.1. Primary characteristics

Redis has a few distinctive characteristics that make it unique in the set of NoSQL databases covered by this book. This section will evaluate these characteristics in regards to the overall influence on the software.

### In memory

Redis is designed to run completely *in-memory* ("Introduction to Redis", n.d.). Traditional databases rely on their data being stored on a hierarchical file system, typically on top of a mass storage medium like hard disk drives (HDDs) or solid state drives (SSDs). While these media often come in much larger sizes than random access memory (RAM) modules, and have a significantly less cost per gigabyte, storing data on them has a few drawbacks. For instance with HDDs, accessing data at a specific position on the disk requires waiting for a data seek operation to complete – essentially letting the physical platter spin to the sector where data is stored and moving the read/write head into position. This makes random read and write operations slow. Even with flash-based storage like SSDs, where seeking data is not an issue, there are many layers of abstraction between the virtual file system and the physical data storage. Accessing a file on file systems typically involves performing a so-called *context switch* into the operating system kernel, accessing a hardware controller, serializing data over a wire according to standards like SATA, and the disk controller finally accessing the data. All of these operations take a considerable amount of time. (Edgar, n.d.) However, when an application accesses its own main memory region in the system's RAM, these operations get processed natively in the central processing unit (CPU) and input/output memory management unit (IOMMU) hardware, without involvement of the operating system kernel or any peripherals.

In summary, storing data in RAM has advantages to system load, seek times, response times and available bandwidth. However, at the time of writing, RAM is significantly more expensive than traditional storage media. The cost per gigabyte ratio could be higher then 238x (when comparing recent prices of a $512GB$ DDR4 ECC memory stick with a $4TB$ $7200rpm$ enterprise HDD).

Because RAM is a form of volatile memory, after a power loss or system reset, the data is cleared. To prevent data loss with Redis, two types of persistence modes can be used ("Redis Persistence", n.d.):

- `RDB` **files**:
  Redis can dump its entire data set into a binary `RDB` file that is sufficient to restore a full and consistent snapshot of a Redis instance. However, creating this dump takes time and memory – Redis forks its primary process and therefore duplicates the entire in-memory data set. Copy-on-write techniques can reduce system load with this process. It is unfeasible to use this method for continuously storing the database's state. ("Redis Persistence", n.d.)

- `AOF` **files**:
  Redis logs all of its transactions into an `AOF` file which can then be used to reconstruct a full snapshot of a Redis instance. This file has the advantage of being

append-only, reducing random accesses and seek times. In addition to that, new transactions can be constantly written to this file without interrupting the primary Redis thread. However, as new transactions can make old ones irrelevant, these files are often not as compact as `RDB` database dumps. Therefore, they can be compacted to contain only required transactions to rebuild the current database state. ("Redis Persistence", n.d.)

#### RESP: Redis serialization protocol

To communicate with a Redis instance, a client has to use the Redis serialization protocol (RESP). The primary goals of this protocol are to be simple to implement, fast to parse and to be human readable. While it only relies on a bidirectional communication channel with some guarantees regarding safety and packet order, it is currently only implemented on top of transmission control protocol (TCP) or UNIX sockets. ("Redis Protocol specification", n.d.)

RESP is designed to adhere to a request-response pattern. Both the requests to the server instance, as well as the responses have a well-defined human readable format. Different parts of the protocol are always terminated with a carriage-return and new-line character (`CR LF` or `\r\n`). ("Redis Protocol specification", n.d.)

Each request is an array of a Redis command and additional string arguments. The length of the array, as well as the size of all strings is sent as a prefix to the respective element. This has the advantage of being both simple to construct, and Redis being able to allocate a fixed size chunk of memory for each element. Therefore, once the data is received, no post-processing is required. ("Redis Protocol specification", n.d.)

The response for a request always starts with an ASCII-byte indicating the response data-type. For instance, this could be "`-`" for a string error, or "`:`" for a string-encoded integer that is guaranteed to be a valid 64 bit signed integer value. Following this byte, the payload is encoded as a (depending on the type *binary safe*) string. ("Redis Protocol specification", n.d.)

Overall, the Redis protocol achieves its goals of speed, simplicity and human readability. Its properties make it easy to develop libraries for communication with Redis.

### 4.1.2. Data Types

Redis is not only a key-value data store but rather a data structures server. In a classical key-value data store a string value is accessed via a string key while Redis supports

several other data structures (hashes, sorted sets, etc.). The basic data structures as well as some extraordinary ones will briefly be described in this section. ("An introduction to Redis data types and abstractions", n.d.)

### Strings

Strings are the most basic data type used in a Redis data store on which all complex data structures are built. Strings are binary safe, which means it is possible to save any kind of data (max 512MB per key), like a JPEG image or a serialized Ruby object, as well as simple text.

Moreover, it is possible to use strings as atomic counters using commands in the `INCR` familiy (`INCR`, `INCRBY`, etc.). Since it is not possible to declare an integer in Redis, strings are used for those purposes. Furthermore, it is possible to use strings as random access vectors due to commands like `GETRANGE`, `SETRANGE` or `GETBIT`. ("An introduction to Redis data types and abstractions", n.d.; "Commands", n.d.; "Data Types", n.d.)

### Lists

Redis lists are a collection of strings sorted by insertion order with a maximum length of $2^{32} - 1$ strings. Among other operations, it is possible to insert and delete elements within a list (either from the head or tail), as well as getting a subset of a list. A list is created when a push operation is performed on an empty key and conversely a list is deleted (key clearance) if the list is emptied by an operation.

Due to the combination of some operations, it is possible to create a customized list for specific use cases. For instance, it is possible to use `LPUSH` and `LTRIM` to create a list with a defined length which will never exceed a certain number of elements. Moreover lists can be used to model a timeline in a social network like Instagram or Facebook. In this example it would be possible to add new elements in the time line (`LPUSH`) and receive only the most recent events (`LRANGE`). ("An introduction to Redis data types and abstractions", n.d.; "Commands", n.d.; "Data Types", n.d.)

### Sets

Unlike lists, sets are an *unordered* collection of strings with a maximum number of $2^{32} - 1$ elements. Elements within a set are called members. Members can be added, removed and returned from a set (`SADD`, `SREM`, `SPOP`). If a string is already contained within the set, it is not possible to add it again. In this case, Redis will simply not add

the member, without indication of an error. Thus, it is not necessary for an application to use `SISMEMBER` before calling the `SADD` operation on a set. Moreover it is possible to display all members of a set and check whether a specific member is contained within a set (`SMEMBER`, `SISMEMBER`).

Due to the characteristics of the `SADD` function, sets can be used to track unique things like students ids lending a specific book in the library. ("An introduction to Redis data types and abstractions", n.d.; "Commands", n.d.; "Data Types", n.d.)

#### Hashes

Hashes are the most suitable data type to represent objects, since they are maps between string fields and string values. Every hash can store up to $2^{32} - 1$ field value pairs.

It is possible to set fields and retrieve the value of fields, both either individually or simultaneously (`HSET`, `HMSET`, `HGET`, `HMGET`). ("An introduction to Redis data types and abstractions", n.d.; "Commands", n.d.; "Data Types", n.d.)

#### Sorted Sets

Similar to sets, sorted sets are non repeating collections of strings ordered by a non-unique score (smallest to greatest). Members within a sorted set can be added, removed and returned (`ZADD`, `ZPOPMIN`, `ZPOPMAX`). Moreover it is possible to return members with a certain score or at a certain position/index within the sorted set. Also, scores can be increased and thereby updated.

Thus sorted sets can be used to keep track of any kind of ranking like a competition. In this case, scores can be initially inserted and later updated using `ZADD`. Due to operations like `ZRANGE` or `ZRANK`, it is possible to receive the top or bottom half of the ranking, or receive the rank of a specific member. ("An introduction to Redis data types and abstractions", n.d.; "Commands", n.d.; "Data Types", n.d.)

### 4.1.3. Multi-node setups / Redis Cluster

Originally, Redis only supported single-node and non-clustered setups. In combination with its mostly single-threaded architecture, this allowed it only scale vertically. However, with the introduction of both external clustering mechanisms (where a so-called proxy would distribute and balance requests across different Redis instances) and internal

clustering support, Redis can now scale horizontally as well. Because of the variety of clustering solutions, and focus on Redis itself, external proxies are out of the scope of this evaluation.

The integrated clustering solution of Redis is called *Redis Cluster*. According to its documentation, *"[it] is a distributed implementation of Redis"* ("Redis Cluster Specification", n.d.) and has three primary goals:

- **High performance and linear scalability** up to 1000 nodes
- **Write safety**
- **Availability**

However, by specification, these criteria do not have to be guaranteed at all times – altogether or even on their own. ("Redis Cluster Specification", n.d.)

Nodes in a Redis Cluster setup are connected over TCP bus connections. These connections (bus) are used to propagate information to all nodes in the cluster. Client requests to nodes in the cluster are not forwarded to the data-holding node. Instead, the client is redirected to the correct node by the use of `MOVED` or `ASK` error return codes. The data distribution is decided by the CRC-16 value of the string key. Each unique CRC-16 value is called a keyslot. All keyslots have one master and $n \geq 0$ slave nodes. ("Redis Cluster Specification", n.d.)

### Positioning in the CAP theorem

In the following, different criteria of Redis Cluster regarding typical usage guarantees in a distributed setup are evaluated:

**Consistency**:

According to the official Redis website, *"Redis is not able to guarantee strong consistency"* ("Redis Cluster Tutorial", n.d.). This can be implied from a few scenarios.

For instance, when a client writes a key to the respective master node for this keyslot, the operation is acknowledged instantly. The master then replicates this write to all slave-nodes for this keyslot. However, when these slave nodes are not reachable, the write is not fully synchronized across the network. In the case of a network partition, a slave that has not yet received this write operation may be promoted to become a master. The write is then lost, although it has been acknowledged. ("Redis Cluster Tutorial", n.d.)

In another example, the network is partitioned into a master-majority and a master-minority partition. For a short amount of time, both partitions accept write operations which are also acknowledged. However, the minority-partition will eventually completely block any write operations. After the network is reunited, the previous writes to the minority-partition are simply discarded. Redis avoids merge operations, as these provide architectural challenges and might not work well on large data ("Redis Cluster Specification", n.d.). "Redis Cluster Tutorial", n.d.

**Availability**:
As already stated regarding the consistency of Redis Cluster, the minority-side of a network partition will refuse write-operations after a timeout. Therefore, Redis Cluster is not available. ("Redis Cluster Specification", n.d.)

Even on the majority-side of the network, some write operations might be refused for a short amount of time. After an initial detection of a network partition and a timeout, slaves of missing keyslots get promoted to be master nodes. As soon as a master exists for all keyslots, the majority-partition is available again. ("Redis Cluster Specification", n.d.)

Depending on the kind of setup and test scenario Redis *tends* to be either AP or CP. Since this is only a small propensity, for systems or applications requiring the characteristic AP or CP, a database designed for a clustering solution should be used. This opinion is, in addition to the official documentation, shared by Davis, 2015 and others. Redis was initially designed as a single node solution with the primary focus on performance.

### 4.1.4. Typical use cases

Redis can be used for a variety of different purposes and use cases. In the following, three typical use cases are stated.

Redis can be used as a general purpose data store, especially if the data and application requires simple data structures and high performance. Nevertheless, Redis is not suitable for every use case requiring a general purpose data store. Since there are no complex data types, besides the ones mentioned in section 4.1.2, and it is not possible to model relations between different data objects (as in a relational database), use cases with these requirements can not be implemented using Redis. Moreover, the high memory usage of Redis can either exceed the capacity of preexisting infrastructure, or increase the cost of purchasing infrastructure drastically.

Due to these characteristics, Redis is often used to store volatile data, as a caching mechanism or as a message broker. ("Commands", n.d.; "Introduction to Redis", n.d.)

### Caching

Based on the characteristics of Redis, Redis is ideal for a caching mechanism. Common database management systems (DBMSs) usually have high latencies and response times which could make the user interface of an application feel sluggish.

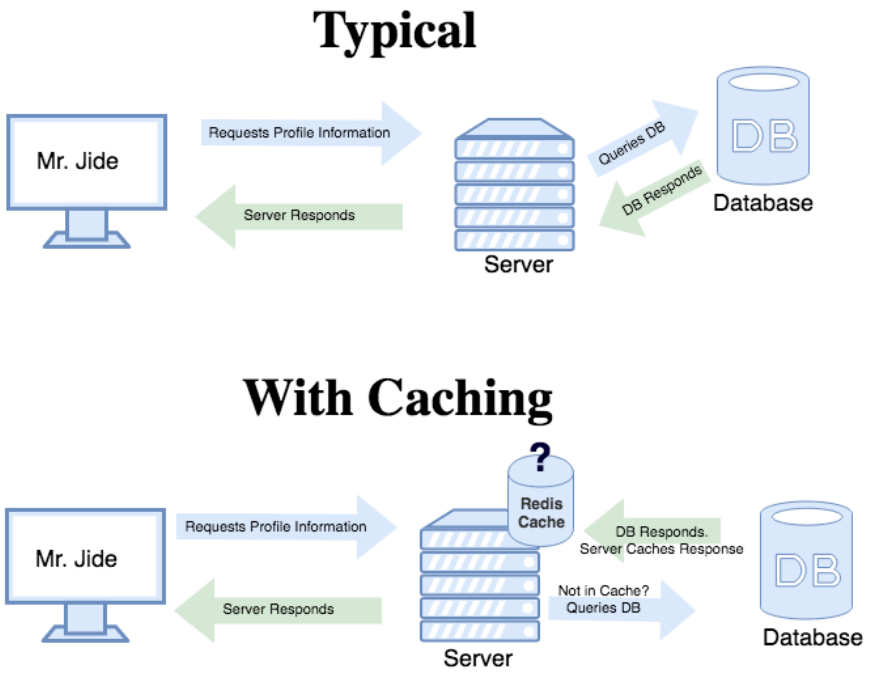


Figure 4.1.: Redis Caching Solution (Akinseye, 2018)

To solve this issue, a caching mechanism can be used. Already prepared and computed data, which is used several times, is stored in the caching mechanism with the result of less interaction between the user interface and the DBMS.

In *figure 4.1*, the difference between server queries to a database with and without Redis as a caching solution is pictured. The second option (with a Redis cache) reduces the response time since only queries, whose data is not already cached, are forwarded to the DBMS.

Redis-keys can be marked for deletion after a specified timeout (`TTL`), and display the time that has elapsed since the key was last modified (`OBJECT IDLETIME`). This enables automatic deletion of seldom used data. Thus Redis can be used as a caching solution for image previews, fetched data from APIs, as well as session data. (“Commands”, n.d.)

#### Message broker functionalities

Since Redis implements a publish-subscribe pattern, it is possible to use Redis as a message broker.

A publish-subscribe pattern is a mechanism where subscribers can receive information/messages from publishers. A typical publish-subscribe system consists of several subscribers and several publishers, where one application can be both subscriber and publisher. Publishers can provide information on specific issues without any knowledge of possible subscribers. Each message is assigned to a topic, which subscribers can subscribe to. In turn, these do not know if and which publisher published on a specific topic. ("Publish/Subscribe", n.d.)

Redis offers the typical publish-subscribe pattern features. Clients can subscribe (`SUBSCRIBE`, `UNSUBSCRIBE`) to a topic as well as push messages to a specific topic/channel (`PUBLISH`). The payload of the message is a binary-safe string, which enables the clients to exchange any kind of data.

This message broker functionality is especially useful for event notifications. For instance the clients can be notified about changes within the data store. Moreover, Redis' publish-subscribe implementation can be used to exchange arbitrary data. However, because this is essentially a routed protocol, it is less performant compared to peer to peer connections such as TCP or UNIX sockets. ("Pub/Sub", n.d.)

### 4.1.5. Conclusion

In this section, the key-value data store Redis was introduced by explaining the main characteristics and some of Redis' data types. In addition to that, a clustered Redis setup is analyzed in regards to characteristics from the CAP theorem. Finally, some of the most popular use cases were reiterated.

To summarize, Redis is much more than a traditional key-value data store. Different data types, carefully chosen architectural decisions and a speed-optimized implementation make it flexible and better suited for some applications. For instance, Redis is an excellent data store for caching purposes. While the publish- / subscribe feature does not necessarily have a great influence on the storage features, it can be used in combination with those to signal other Redis clients that some keys changed.

However, having a purely in-memory data store means that it is expensive to operate with vast amounts of data. Also, data persistence is possible, but only with a few disadvantages. Last but not least, Redis can be used in a clustered setup. The internal

clustering mechanisms (Redis Cluster) do not provide strong guarantees regarding availability as well as consistency. This severely limits its use-cases to applications, where both the correctness and presence of distributed data is not a strict requirement.

## 4.2. The Riak Key-Value Store

Daniel Rutz, Paul Thore Flachbart

### 4.2.1. Introduction

The Riak KV authors describe Riak KV as "a distributed NoSQL database designed to deliver maximum data availability by distributing data across multiple servers. As long as your Riak KV client can reach one Riak server, it should be able to write data." ("Riak KV", n.d.) Actually, this sentence already describes most of Riak's characteristics:

- Riak has been developed with availability in mind. It constructs a distributed system of nodes without master node. Even though this system can't guarantee consistency, it makes sure that the database is available as long as one node is accessible.

- Riak is a NoSQL database. Instead of using the Structured Query Language (SQL) language, Riak provides an HyperText Transfer Protocol (HTTP) ("HTTP API", n.d.) and a Protocol Buffers ("Protocol Buffers Client API", n.d.) interface for Create, Read, Update, Delete (CRUD) operations on key-value pairs.

There are two different databases besides Riak KV:

- Riak TS has been developed for time series data. It is not scheme-free: You have to describe tables in a way similar to SQL ("Riak TS", n.d.).

- Riak CS is a cloud storage solution. It has been developed to be compatible to the Amazon S3 application programming interface (API) ("Riak Cloud Storage", n.d.).

This chapter will deal with advantages and disadvantages of Riak KV. Furthermore, we will compare Riak KV to Redis and categorise it according to the CAP theorem. The other Riak[1] variants are not in the scope of this text.

[1] From now on, Riak KV will be referred to as Riak.

### 4.2.2. Characteristics of Riak

Riak is a key-value store written in Erlang. According to Kuznetsov and Poskonin (2014), it is inspired by the Amazon Dynamo whitepaper (DeCandia et al., 2007). Its main focus is distributivity: By using concepts such as consistent hashing and synchronisation using vector clocks, it does not need a master node to distribute data across multiple nodes.

Riak can be used in a *eventually consistent* or in a *strongly consistent* mode: When used with eventual consistency, an accessible Riak node will always answer to a request, but it cannot guarantee the response to be up-to-date. With strong consistency, Riak internally tries to solve the Byzantine Generals problem by achieving a distributed consensus between the nodes about the current value. If less than half the replications of a value are not available, however, Riak will not be able to return a response as the required quorum will not be reached. It should be noted that strong consistency is flagged as experimental; the Riak authors discourage its usage in production environments. ("Strong Consistency", n.d.)

Riak splits its keyspace into so-called *buckets*. A key can be used multiple times as long as all the usages are in different keys. Access to a value must be done with bucket and key.

A really interesting feature of Riak is the possibility to use MapReduce for queries of data: By sending a special query to the cluster, it can distribute the collection of requested data to all nodes. Every node now only needs to process only a subset of all key-value pairs. This allows distribution of computing power over multiple nodes ("Using MapReduce", n.d.).

### 4.2.3. Placement inside CAP Theorem

The CAP Theorem as stated by Fox and Brewer (1999) says that a database system is not able to be consistent, available and partition tolerant at the same time. Riak has been designed with this principle in mind. In its standard configuration, Riak tries to be available under every circumstances, even when parts of the cluster are not available. This leads to eventual consistency because a node might not know about changes inside a key-value pair yet. This makes Riak an **AP** database. If Riak is configured for strong consistency, it gets unavailable if the node can not reach a distributed consensus about a value. However, Riak can guarantee the answer to be the lastest. That means that Riak in strong consistency mode is a **CP** database.

### 4.2.4. Advantages and Disadvantages

Riak has several advantages and disadvantages:

- Its main advantage is the high level of availablility. Every node of a Riak cluster is able to answer queries, and even in the case of other node being unavailable, a Riak node will still try to answer. Writing data to a Riak node is always possible as long as the node is available.

- Riak has been developed for high scalability. Because Riak does has a masterless structure where data automatically get redistributed when node are added or removed, it is able to handle bigger amounts of data without problems. In order to support this, adding and removing nodes in a cluster has been designed to be very easy.

- Its main disadvantage is the very low consistency guarantee. In some cases, it might be better to get no result at all instead of getting wrong or old data. When using Riak, this problem must always be addressed.

- The original development company of Riak is out of business. That means that there is no official commercial support for Riak. Instead, it is developed by the community. Especially in large production environments, this uncertainty about further support can lead to problems.

- The Riak developers state that Riak is not suitable for small deployments because they do not need distributivity. For smaller databases, several alternatives are available.

### 4.2.5. Comparison to Redis

As Redis and Riak both are key-value stores, a comparison is very interesting in order to show their main differences:

- While Riak uses HTTP or Protocol Buffers for access, Redis has a custom query language (“Redis Protocol specification”, n.d.).

- Riak and Redis have different main focuses: While Riak is optimised for high availability, Redis is optimised for speed (“Introduction to Redis”, n.d.).

- Riak is a persistent database. Redis offers persistency, too, but has been optimised for usage as an in-memory database.

- Riak offers masterless replication. Redis however uses a client-server model for replication of data ("Redis Cluster Specification", n.d.).

These points show that Redis is tailored for in-memory caching in special, while Riak has been developed for actual persistent business data.

### 4.2.6. Test Implementation

In order to demonstrate the usage of a Riak database instance, a test application for NodeJS has been written. There is a NodeJS client for Riak that can be used ("The Riak client for Node.js." n.d.). It offers a special function `fetchValue`, which takes the bucket that holds the data, and the specific key the user wants to access. It will then do the query and call a callback afterwards. In our case, we store user data inside the Riak database. The username is used as the key. With the key, we get a user object from Riak that contains the password. We compare to the password given by the user. If it does not match, an error is shown. Otherwise, we display the user's name as saved in the database. This access together with error handling took 10 lines, showing that it is easy to retrieve data from Riak.

### 4.2.7. Conclusion

We have introduced Riak, a distributed key-value store optimised for availability. We have shown different advantages and disadvantages of the database. Afterwards, we stated its main differences to Redis as an example for another key-value store. In the end, we have shown a test implementation for an application using Riak.

Our research shows that Riak is ideal for big deployments with large databases where availability is really important. It should be noted, however, that Riak does not offer high consistency, which might be a problem in several use cases.

# 5. Graph Databases – Neo4j

Thore Krüss, Lennart Purucker, Johanna Sommer

## 5.1. 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.

## 5.2. 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 5.3, 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 5.4, where an implementation example will be given, focusing in particular on Neo4j and also explaining how an SQL example would be transformed to fit Graph Databases. Lastly, our findings will be stated in subsection 5.5 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.

## 5.3. 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 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).

### 5.3.1. 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 5.1 for an example.

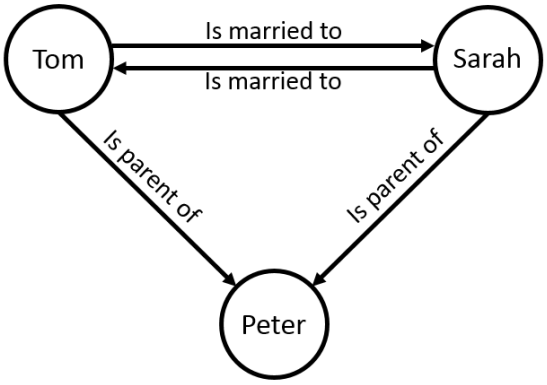


Figure 5.1.: A graph representing a small family. Tom, Sarah and Peter are entities and thus represented as nodes (circles). The two-way relationship between Tom and Sarah, their marriage, is depicted as two directed edges. Lastly, Tom and Sarah are the parents of Peter and thus both have the relationship "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 5.2 shows a labeled property graph.

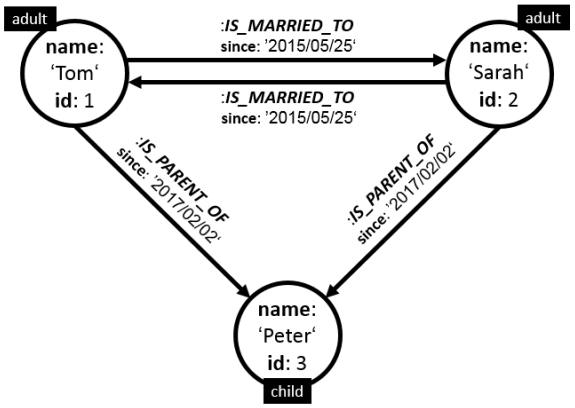


Figure 5.2.: A labeled property graph representing a small family (Eifrem, Webber, & Robinson, 2015; Lal, 2015). Compared to figure 5.1 additional information can been stored.

### Concepts of the property graph

**Nodes:** Like a normal graph, the property graph represents entities as nodes. The nodes in figure 5.2 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 5.2, 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 5.2 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 5.2, `"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) .

### 5.3.2. 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 (5.3.4).

#### 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", 5.3.3). 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 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.

### 5.3.3. 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.

### 5.3.4. 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.

## 5.4. 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.

### 5.4.1. 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.

### 5.4.2. 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 5.3 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.

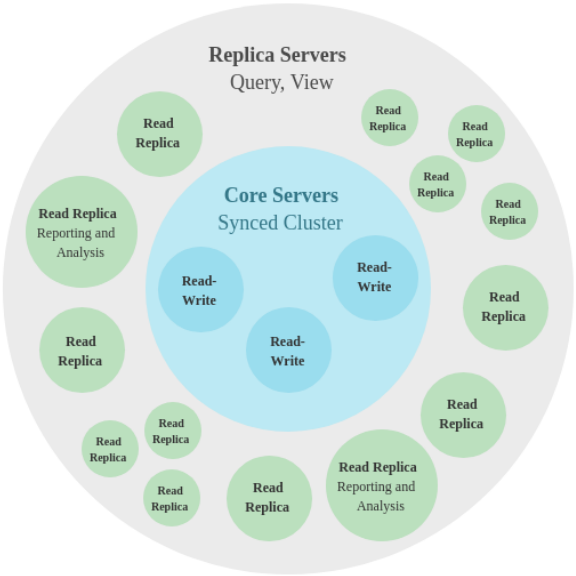


Figure 5.3.: Neo4j causal Cluster Architecture ("The Neo4j Operations Manual v3.5", 2019, Chapter 5.1)

**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 5.4.2 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 5.4 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.

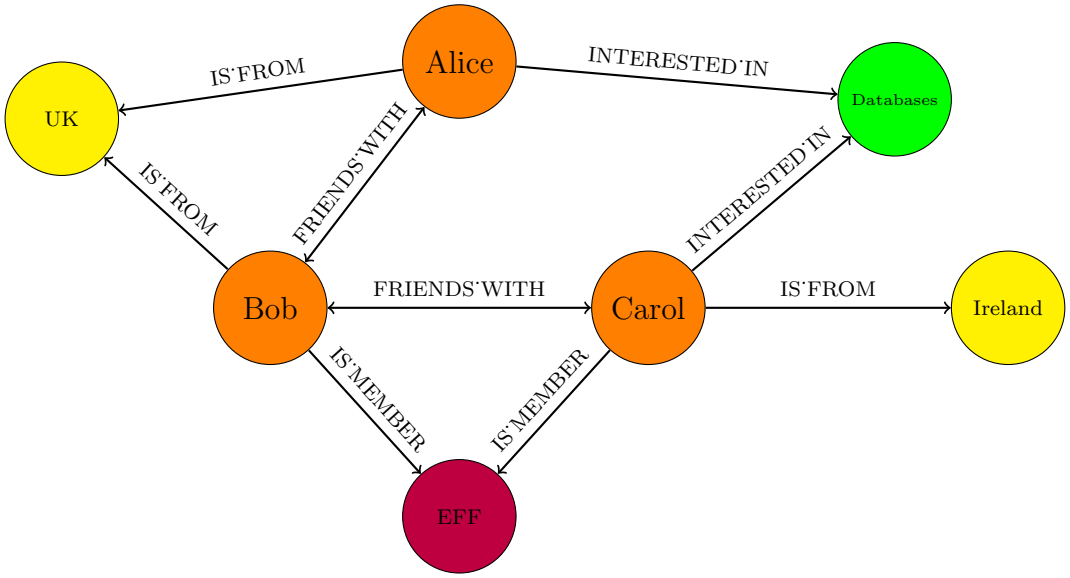


Figure 5.4.: Sample database schema

### Implementation in Python

While it is possible to manage the database utilizing the CRUD-functionality from Neo4js own query language Cypher (see 5.4.2) 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.

```
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 5.4 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).

### 5.4.3. 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.

## 5.5. Reflection

### 5.5.1. 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 5.3.4 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.).

### 5.5.2. 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


Akinseye, O. (2018). Introduction to caching and redis. Retrieved April 15, 2019, from https://www.codementor.io/brainyfarm/introduction-to-caching-and-redis-h6o16p4qx

An introduction to Redis data types and abstractions. (n.d.). Retrieved March 27, 2019, from https://redis.io/topics/data-types-intro

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*.

Brewer, D. E. A. (2000). Towards robust distributed systems. Retrieved April 15, 2019, from https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Commands. (n.d.). Retrieved March 27, 2019, from https://redis.io/commands

Data Types. (n.d.). Retrieved March 27, 2019, from https://redis.io/topics/data-types

Davis, M. (2015). An introduction to redis cluster. Retrieved March 31, 2019, from https://www.credera.com/blog/technology-insights/open-source-technology-insights/an-introduction-to-redis-cluster/

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

DB-Engines Ranking. (n.d.). Retrieved April 15, 2019, from https://db-engines.com/en/ranking

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first acm sigops symposium on operating systems principles* (pp. 205–220). SOSP '07. doi:10.1145/1294261.1294281

Edgar, J. (n.d.). CMPT 454: Data storage and disk access. Retrieved April 12, 2019, from http://www.cs.sfu.ca/CourseCentral/454/johnwill/

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.

Fox, A., & Brewer, E. A. (1999). Harvest, yield, and scalable tolerant systems. In *Proceedings of the seventh workshop on hot topics in operating systems* (pp. 174–178). doi:10.1109/HOTOS.1999.798396

Hazelcast Downloads. (n.d.). Retrieved April 5, 2019, from https://hazelcast.org/download/

Hazelcast IMDG Reference Manual. (n.d.). Retrieved March 27, 2019, from https://docs.hazelcast.org/docs/3.11.2/manual/html-single/index.html#hazelcast-imdg-reference-manual

Hazelcast Jet. (n.d.). Retrieved March 27, 2019, from https://hazelcast.com/products/jet/

HTTP API. (n.d.). Retrieved March 31, 2019, from https://docs.riak.com/riak/kv/2.2.3/developing/api/http/index.html

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

In-Memory NoSQL with Hazelcast IMDG. (n.d.). Retrieved March 27, 2019, from https://hazelcast.org/use-cases/in-memory-nosql/

Introduction to Redis. (n.d.). Retrieved March 27, 2019, from https://redis.io/topics/introduction

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

Johns, M. (2015). *Getting started with hazelcast - second edition* - (2nd ed.). Birmingham: Packt Publishing.

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

Kudraß, T. (2015). *Taschenbuch datenbanken*. Packt Publishing.

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

Kuznetsov, S. D., & Poskonin, A. V. (2014). Nosql data management systems. *Programming and Computer Software*, *40*(6), 323–332.

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

Luck, G. (n.d.). Jepsen analysis on hazelcast 3.8.3. Retrieved April 1, 2019, from https://hazelcast.com/blog/jepsen-analysis-hazelcast-3-8-3/

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

Mendis, W. S. (n.d.). From rdbms to key-value store: Data modeling techniques. Retrieved April 2, 2017, from https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46

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/

*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

Protocol Buffers Client API. (n.d.). Retrieved March 31, 2019, from https://docs.riak.com/riak/kv/2.2.3/developing/api/protocol-buffers/index.html

Pub/Sub. (n.d.). Retrieved March 27, 2019, from https://redis.io/topics/pubsub

Publish/Subscribe. (n.d.). Retrieved April 15, 2019, from https://www.ibm.com/support/knowledgecenter/de/SSCGGQ_1.2.0/com.ibm.ism.doc/Overview/ov00030.html

Redis Cluster Specification. (n.d.). Retrieved March 31, 2019, from https://redis.io/topics/cluster-spec

Redis Cluster Tutorial. (n.d.). Retrieved April 14, 2019, from https://redis.io/topics/cluster-tutorial

Redis Persistence. (n.d.). Retrieved March 27, 2019, from https://redis.io/topics/persistence

Redis Protocol specification. (n.d.). Retrieved March 26, 2019, from https://redis.io/topics/protocol

Redis Replacement. (n.d.). Retrieved March 27, 2019, from https://hazelcast.org/use-cases/redis-replacement/

Riak Cloud Storage. (n.d.). Retrieved March 31, 2019, from https://docs.riak.com/riak/cs/2.1.1/

Riak KV. (n.d.). Retrieved March 17, 2019, from https://docs.riak.com/riak/kv/2.2.3/index.html

Riak TS. (n.d.). Retrieved March 31, 2019, from https://docs.riak.com/riak/ts/1.5.2/

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

Strong Consistency. (n.d.). Retrieved March 31, 2019, from https://docs.riak.com/riak/kv/2.2.3/learn/concepts/strong-consistency/index.html

Tasci, S., & Demirbas, M. (2015). Employing in-memory data grids for distributed graph processing. In *2015 ieee international conference on big data (big data)* (pp. 1856–1864). doi:10.1109/BigData.2015.7363959

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

The Riak client for Node.js. (n.d.). Retrieved April 13, 2019, from https://github.com/basho/riak-nodejs-client

Using MapReduce. (n.d.). Retrieved April 12, 2019, from https://docs.riak.com/riak/kv/2.2.3/developing/usage/mapreduce.1.html

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

What CAP Theorem Means to a Business Leader. (n.d.). Retrieved April 1, 2019, from https://hazelcast.com/blog/what-cap-theorem-means-to-a-business-leader/

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" }
        ]
      }
    ]
  }
]
```