NoSQL Databases

Course STG-TINF16A of the Baden-Wuerttemberg Cooperative State Univers

May 28, 2019

Contents

Glossary

ACID atomicity, consistency, isolation, durability 36

API application programming interface 26, 29, 78

ASCII American Standard Code for Information Interchange 22

ATLAS ATLAS (A Toroidal LHC ApparatuS) is one of the seven particle detector experiments at the Large Hadron Collider (LHC), a particle accelerator at CERN. It generates 1 petabyte of raw data per second, even after filtering it still requires over 100 megabytes of disk space per second – at least a petabyte each year (DOE/Fermi National Accelerator Laboratory, 2010). 33

BSD Berkeley Software Distribution 19

BTRFS Better File System (modern copy on write filesystem) 65

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. 20

CERN CERN, the European Organization for Nuclear Research, is one of the world's largest and most respected centres for scientific research. 33

CPU central processing unit 20

CQL Cassandra Query Language 36, 40, 41

CRUD Create, Read, Update, Delete 29, 73, 76, 86, 88

DBMS database management system 7, 26

DDR4 double data rate 4 20

ECC error correcting code 20

HDD hard disk drive 20

HTTP HyperText Transfer Protocol 29, 31

IAM identity and access management 80

IOMMU input/output memory management unit 20

IT information technology 80

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

MDM master data management 79

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. 7, 8, 20, 28, 29, 71, 75, 76

OGM object-graph mapping 86, 88

OLTP online transaction processing 73

ORM object-relational mapping 86

RAM random access memory 20, 21

RDBMS relational database management system 71, 72, 76

RESP Redis serialization protocol 21

SATA serial ATA 20

SQL Structured Query Language 7, 29, 36, 40–42, 59, 61, 62, 72, 77, 82, 86–89

SSD solid state drive 20

TCP transmission control protocol 21, 24, 28

1. Introduction

In a world of unstructured data, NoSQL Databases are of never-ending popularity. While SQL databases are still the most popular, two out of the ten most used databases already are NoSQL ("DB-Engines Ranking", n.d.). The trends show that the interest in NoSQL increases a lot faster than in traditional databases ("DB-Engines Ranking", n.d., section `ranking_categories`). That shows that it is necessary to consider NoSQL databases as alternatives for relational databases and to discuss their advantages as well as their disadvantages.

The hype around NoSQL Databases in today's landscape can not be disregarded. The popularity for NoSQL databases has been steadily increasing and gaining fans from all over the database community.

But the term NoSQL was first used in 1998 already, introduced by Carlo Strozzi. While the term can mean either 'not only SQL' or 'no SQL', it is uniform in the meaning of not being a relational database system. Soon after, in the beginning of the next century, NoSQL really started gaining momentum and first implementations of different technologies started coming up. One of the first NoSQL technologies to come up was the document oriented database implementation Metakit, soon followed by the first ever graph database Neo4j in the year 2000. After that, many more followed and NoSQL databases have become a central point to the database landscape today.

While it is clear that NoSQL databases will not replace all other database management systems (DBMSs) in existence today, they are especially popular in a few distinct scenarios: For instance, when prototyping a new application while the data format is still undecided, NoSQL databases provide developers with flexibility unseen in usual relational DBMSs. In addition to that, the fast response and processing times can drastically increase an application's performance, if used correctly. Also, the ability to store data which does not adhere to a constant schema is significant with ever-changing and uncontrollable data sources.

With the increasing amount of databases offered, choosing the right one can be overwhelming. Not only does one need to decide for the right data structure, but also find an implementation that fits the needs of your application.

Due to this growing importance of NoSQL databases in today's IT-landscape and the

abundance of different technologies implementing NoSQL, it is of great importance to provide some guidance for this topic. For this reason this book delivers not only an overview over some of the most popular NoSQL databases but also analyses their features in the context of the CAP-theorem developed by Brewer.

This eBook gives an overview of different types of NoSQL technologies that are relevant today. First, it will talk about Key-Value databases including a quick general introduction and the selected implementations Hazelcast, Redis and Riak. After that, Column Wide Oriented databases are introduced, including the basics of Cassandra. Chapter 4 focuses on Documented Oriented databases like Couchbase and Rethink DB. Lastly, the principals of Graph databases are described with Neo4j as an example implementation.

2. Key-value databases

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.

2.1. Hazelcast

Vanessa Jörns, Tobias Schiffmann and Victor Veal

2.1.1. 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.

2.1.2. 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).

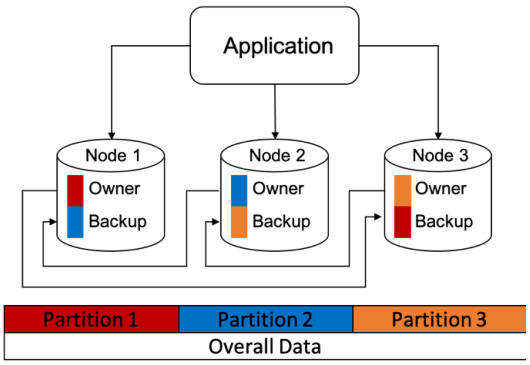

Figure 2.1.: Hazelcast Nodes (Johns, 2015)

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).

2.1.3. 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.).

2.1.4. 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 2.2.: 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 2.3.: 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 2.4.: 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.

2.1.5. 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

2.1.6. 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.

2.2. 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.

2.2.1. 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.

2.2.2. 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.)

2.2.3. 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.

2.2.4. 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 2.2.2, 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 DBMSs usually have high latencies and response times which could make the user interface of an application feel sluggish.

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 2.5*, 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.)

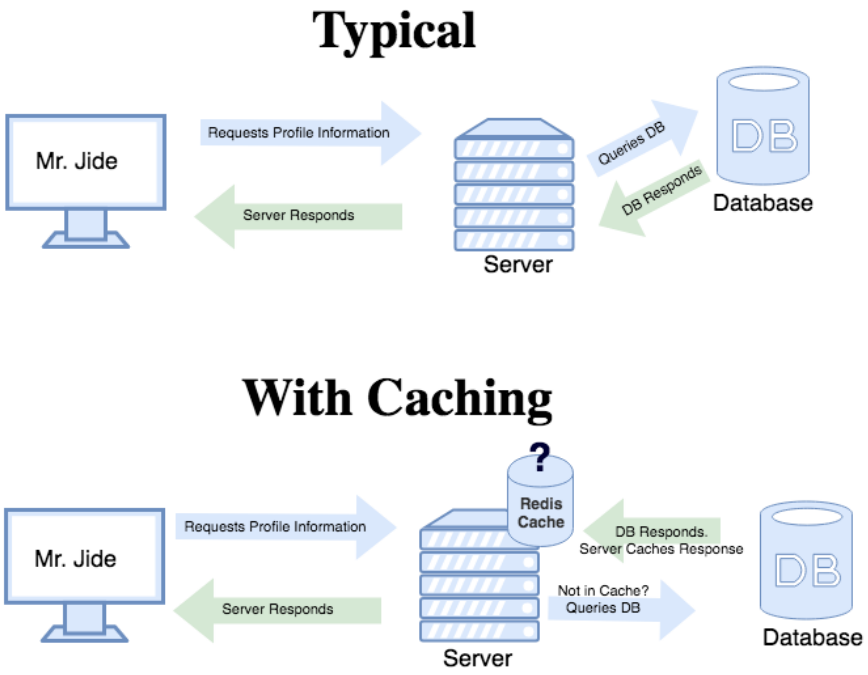

Figure 2.5.: Redis Caching Solution (Akinseye, 2018)

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.)

2.2.5. 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.

2.3. The Riak Key-Value Store

Daniel Rutz, Paul Thore Flachbart

2.3.1. 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.

2.3.2. 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

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

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.).

2.3.3. 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.

2.3.4. 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.

2.3.5. 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.

2.3.6. 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.

2.3.7. 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.

3. Wide Column Store – Apache Cassandra

David Marchi, Daniel Schäfer, Erik Zeiske

3.1. Introduction

The following chapter aims to bring an overview into the Cassandra data store as well as detailed information about how it works and its architecture. After that, a detailed guide on how to model data for Cassandra in order to make the best use out of the architecture will follow. Transitioning into technical details and examples on how Cassandra works, what features and pitfalls it brings with it.

Many large companies and organizations have deployed Cassandra clusters. Here is an overview of exceptional cases:

CERN	Storage backend for ATLAS detector (Sicoe, 2012)
Netflix	2,500 nodes, 420 TB, over 1 trillion requests per day Migrated from Oracle to Cassandra (Cockcroft, 2011)
eBay	6 billion writes and 5 billion reads daily Single Cassandra table of 40TB (Patel, 2012; Qu, 2014)
Apple	More than 75k Cassandra nodes with 10PB in production Several clusters with 1000+ nodes each (Kohli, 2014)

This shows that Cassandra is heavily used in large-scale deployments of big tech companies and other organizations. The significance of Cassandra and the problems it solves is, with an ever growing amount of data, high. For trying out Cassandra on a single node or a small local cluster however all of those settings can be left at their defaults. To keep this chapter at a manageable size we will not explain the topological features, only mention which are available. This chapter will be wrapped up with instructions and hints for trying out Cassandra on a single node or a small local cluster. Cassandra can be made aware of the relative physical location of all nodes, for example in which rack or data center they are. For real world usage it is recommended to do that and also adjust

the configuration to take that into account.

3.1.1. Overview of Cassandra

Cassandra is a Wide Column Store Database. It is table-like but not relational. The architecture and design make it a distributed and masterless data store for large data. It is written in Java and can be run distributed on commodity hardware. Since it is masterless there is no single point of failure. Nodes (connected instances running Cassandra) can be hot-swapped or be temporarily down without affecting the availability of the Cassandra cluster. Adding a new node and removing one will elastically increase or decrease the cluster size without any major impact on the distributed system. This can be refereed to as elastic scalability. All these features make Cassandra highly available and fault tolerance. Depending on the settings, a cluster of Cassandra nodes can be more or less consistent. From eventually consistent, the lowest setting for consistency, to highly consistent. This can be referred to as tunable consistency and allows for a configuration dependent classification in the CAP Theorem. More on that in section 3.7.

Cassandra was originally developed by Facebook in 2008, the co-author of Amazons DynamoDB was involved in the development process. Hence Cassandra shares similarities with the architecture of DynamoDB (Lakshman & Malik, 2010). Currently (Apr/19) it is a free software project of the Apache Foundation. Development is mainly driven by DataStax. A company designated for commercial Cassandra use and enterprise support.

3.2. Wide Column Store

A wide column store is a tabular but not relational approach to store data. It is not column oriented, since the rows are stored together. A row can have missing columns which are not stored on disk. This makes it sparse. It can be thought of like a key-value store where the value can have a subset of a predefined set of columns. Or, put in other words; "Sparse, distributed multi-dimensional sorted map" (Chang et al., 2008) The naming can be explained as a key value store with wide (complex) values that consist of columns.

Figure 3.1 shows in a graphical representation the structure of a wide column store. The name is the primary key and different people have different columns. The columns with no data are not stored as null but rather not even stored as a whole.

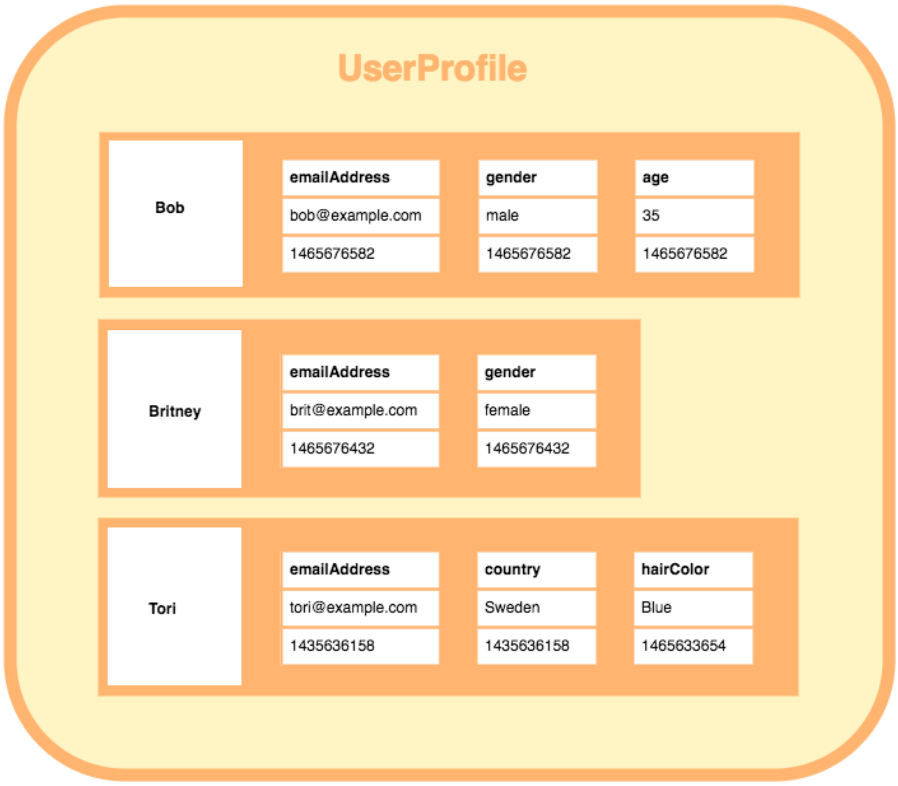

Figure 3.1.: Example entity relation model (database.guide, 2016)

3.3. Use-Cases Cassandra

Cassandra comes with strong benefits over other database systems. These benefits and extra features make it miss some features which one might expect as standard in database systems. It should be carefully considered whether to use Cassandra or not since it is not the jack of all trades. One benefit of Cassandra is fast writes which means it can handle a high throughput but the latency might not be too short. These writes include the operations of INSERT, UPDATE and DELETE. The way Cassandra handles those operations make them equal to a "regular" write operation. This will be explained in section 3.5. Another key benefit is high availability. Due to its architecture and depending on the settings, a Cassandra setup can be highly available. Since one of the base assumptions of a distributed system is the expensiveness of inter-node communication, it has an linear horizontal scalability. The communication between nodes does not increase with the size of node. Due to its distributed nature, the architecture does not rely on a master-slave setting and comes masterless. Furthermore it can be configured to work with several clusters, globally distributed, which makes for example multicloud Cassandra setups possible. This means that any client can read from and write to any node. As described previously Cassandra is a wide column store database which results in having a flexible schema where rows can have missing columns that

are not stored on disk. Cassandra has its own query language which is similar SQL and called Cassandra Query Language (CQL). This makes it easier to use Cassandra if knowledge from SQL is available.

If any of these points apply to a project or Use-Case, Cassandra is probably a good fit. But to be sure it is even more important to rule out the cases where Cassandra is not a good fit. The following paragraph will describe limits of Cassandra. If the use-case needs any of those features, Cassandra will most likely not suffice to fulfill requirements. First, a single system instance with Cassandra should be avoided. Most of the features and benefits come with multiple node setups. Use-Cases which would need a dozen nodes seem to find a great fit with the distributed storage system.

Equally important is the way data has to be stored and accessed. Due to its architecture, data has to be modeled different in Cassandra. ACID transactions are not possible and tables are fine-tuned for pre defined queries. Those queries should be known early on and not change during use. It is not easily possible to change or extent those queries. Furthermore there are no relations between tables. It is possible to link, connect and reuse IDs but this has to be handled on client side, there are no operations on those IDs available. Additionally column aggregation operations such as `GROUP BY` are also not possible, since such operations would be very inefficient in a distributed wide column storage.

Having a lot of updates and deletes interspersed with reads slow the system down due to the way of handling requests - just appending data, not changing records and merging them while reading. This has affects on data validation as well. It is not possible to check on write for data uniqueness, constraints or auto increments. (Carpenter & Hewitt, 2016)

To sum it up there are general Use-Case conditions where Cassandra is a great fit. In order to make an educated decision the following key-points should fit with the Use-Case and environment:

- Large Deployments
- High write throughput
  - "High performance at high write volumes with many concurrent client threads" (Carpenter & Hewitt, 2016)
- Geographical distribution of data and database clients
- Different columns per row

3.4. Data Modeling in Cassandra

Compared to other database systems like traditional relational database systems, data modeling is different in Cassandra. Here is a little example to understand the different way of thinking when modeling data for Cassandras wide column data store.

Figure 3.2 shows an example entity relationship diagram of data in the context of a hotel environment. The hotel has the attributes *address*, *phone*, *name* and an *id* as unique primary key. Each hotel has certain amounts of rooms which store information them-self and are connected to other entities.

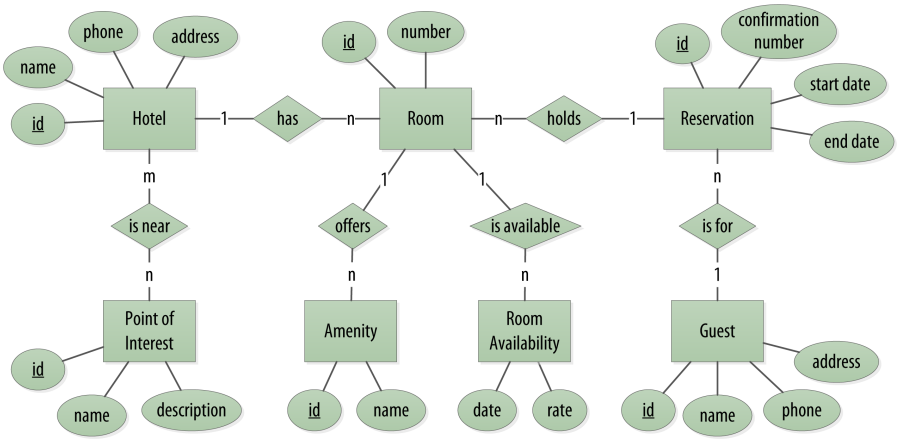

Figure 3.2.: Example entity relation model (Carpenter & Hewitt, 2016)

In figure 3.3 an example database model for an relational database of the ERP from figure 3.2. The connections and primary keys previously noted connect now the table to enable complex join queries over multiple tables later on.

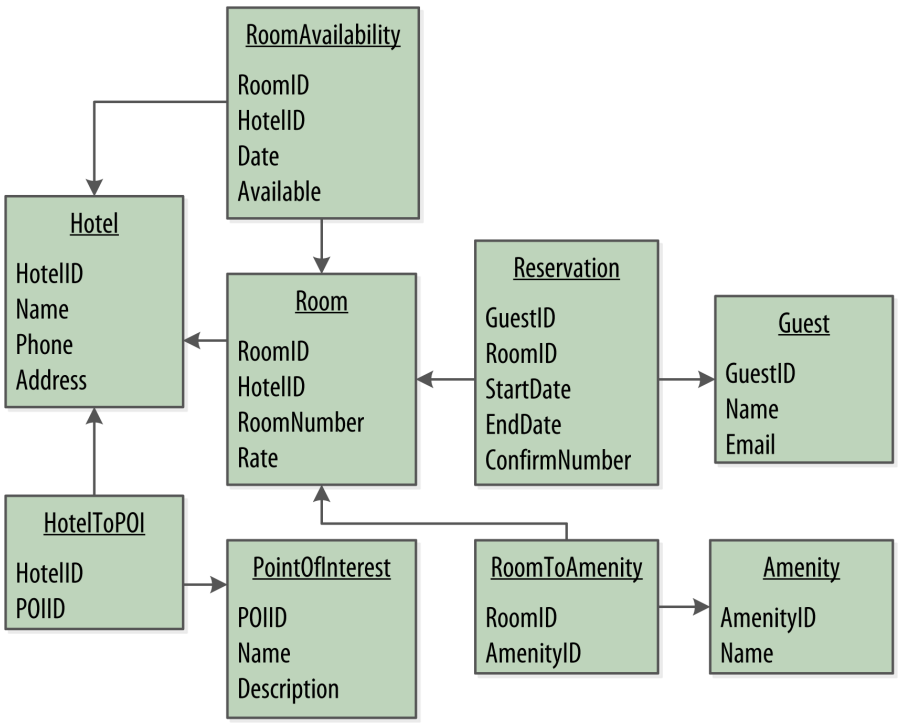

Figure 3.3.: Example RDMBS normalization transition (Carpenter & Hewitt, 2016)

The fundamental difference between those two modeling types is the starting point. In the example above it was all about what data has to be stored and how that data is connected to each other. In data modeling for Cassandra the queries are the starting points. This means for the architect that they first have to think about queries the database system has to answer.

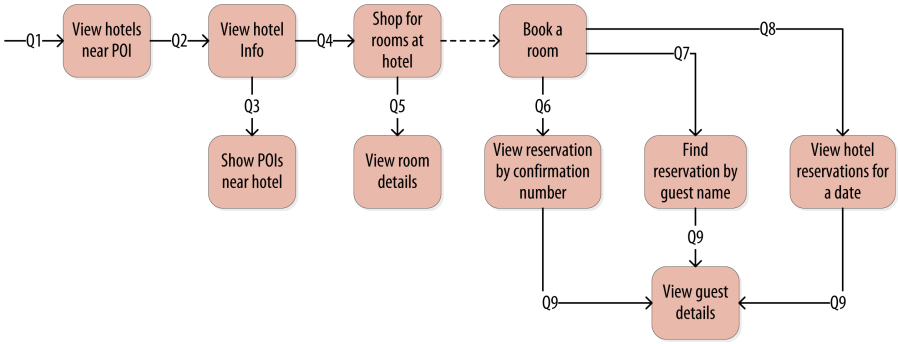

Figure 3.4.: Planned queries against database system (Carpenter & Hewitt, 2016)

Figure 3.4 show this circumstance; queries the system later has to answer to. This diagram describes a classical use-case for the hotel environment in which the database

would come to work. A user searches for hotels, checks information about it and reserves it.

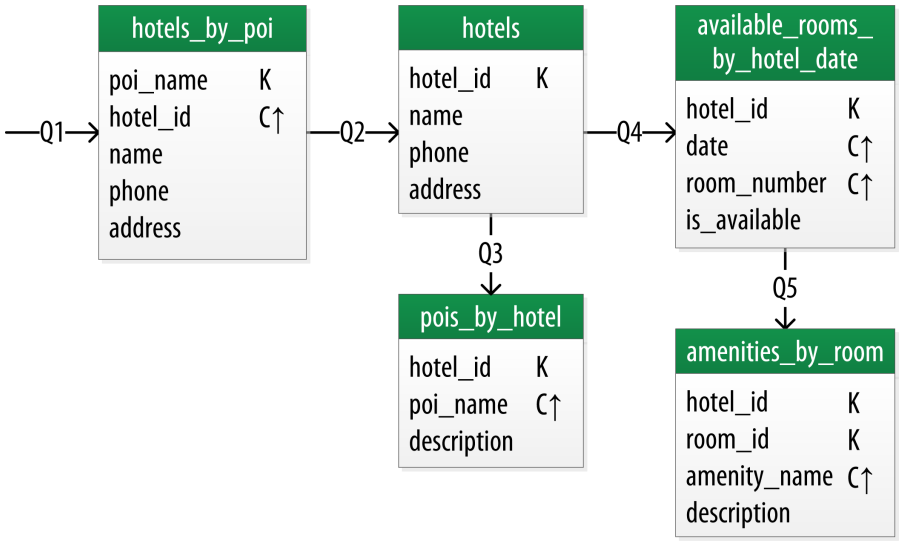

Figure 3.5.: Model and denormalize tables to fit queries (Carpenter & Hewitt, 2016)

Transitioning from the collection of queries in figure 3.4 to accessible tables, can be seen in figure 3.5. It can be seen that data is stored redundantly, available on multiple tables. This does not bother the modeling since queries later on will not use any sort of join operation and be limited onto one table request - which is precisely why data is stored in multiple tables. Each table fulfills one of the queries previously thought of. Furthermore since there is no referential integrity it is possible to store *ids* in tables but the database system does not provide any functionality to make use of and / or connect those *ids*.

In order to properly graph and document this type of modeling, a uniform and new way was proposed to write down data modeling for Cassandra and other similar systems with same needs. (Chebotko, Kashlev, & Lu, 2015) These diagrams are called Chebotko diagrams and include the way queries are planned. This can be seen in figure 3.4. Out of this table, the diagram can be concluded. Creating to each query an unique table with all necessary information. The arrows with the queries numbers (Q1, Q2 ..) will be kept to keep the context and relation to the previous diagram (3.5). This can be seen in a descriptive example section around figure 3.6

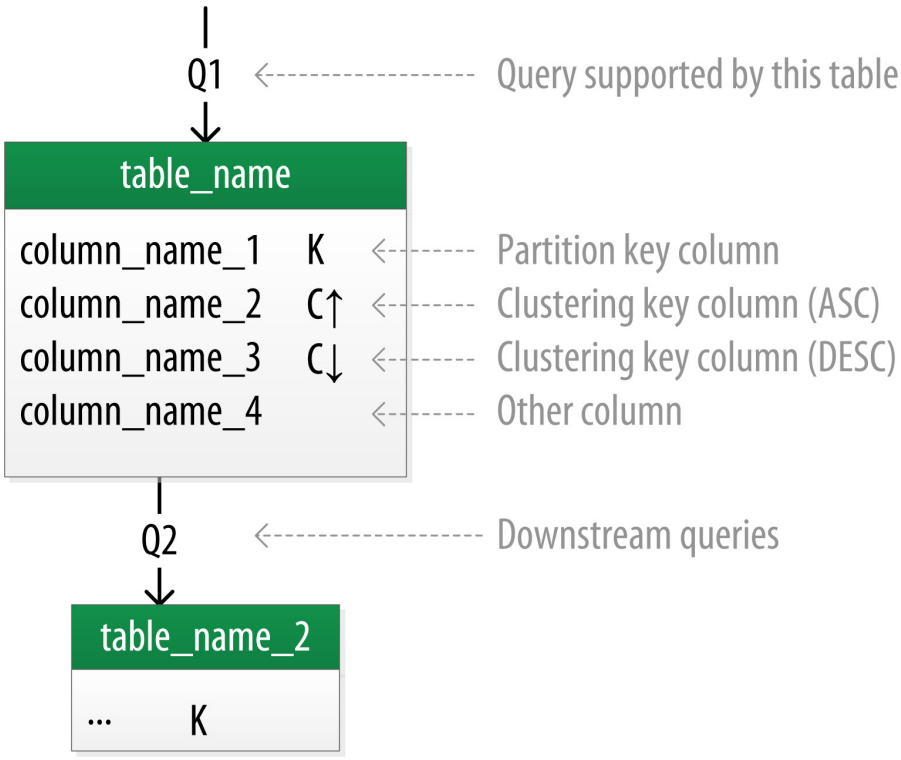

Figure 3.6.: Model and denormalize tables to fit queries (Carpenter & Hewitt, 2016)

3.5. Using the Cassandra Query Language

This section will give a short overview of how to interact with a Cassandra database using CQL, which is mainly inspired by SQL (Cannon, 2012; DataStax, Inc., 2019a; Meng, 2014).

3.5.1. Creating a keyspace

This similarity becomes clear by looking at how the creation of a keyspace is performed:

```
/* Create a new keyspace in CQL */
CREATE KEYSPACE data WITH replication =
    {'class': 'SimpleStrategy', 'replication_factor': 3};

/* Create a new database in SQL */
CREATE DATABASE data;
```

Hereby the only difference is that instead of creating a database, a keyspace is created and it is possible to specify which replication parameters should be used. What these parameter mean and how they should be used is explained later in section 3.7 (Meng, 2014).

3.5.2. Creating a table

After creating a keyspace a table has to be created in order to hold the data. As a database is always part of a keyspace it is either necessary to specify the keyspace in every query or to simple scope every subsequent query into a given keyspace by using the USE query (DataStax, Inc., 2019j):

```
USE data;
```

Using this keyspace a table can be created using the same syntax as in SQL (Cannon, 2012; DataStax, Inc., 2019b; Meng, 2014):

```
CREATE TABLE groups (
   group_name varchar,
   group_location varchar,
   added_date date,
   username varchar,
   PRIMARY KEY (...)
);
```

Hereby the only difference is how the primary key can be specified:

```
      partition key       clustering key  clustering key
       |       |                 |            |
((groupname, group_location), added_date, username)
```

Figure 3.7.: Parts of a primary key specification in CQL (J. Miller, 2014)

The first part of the definition will always be the partition key. If it is a compound of several columns they need to be surrounded by parentheses and separated by comma in order to state that they as a whole form the partition key. If necessary the partition key can be followed by several clustering keys. Keep in mind that the data will be ordered first by the first clustering key, after that by the second and so on. This means that an `ORDER BY` has to first be called on the first clustering key and a second ordering can be

done on the subsequent one. It will not be possible to only order by the second or other subsequent clustering keys when not ordering by the first. Any other non-primary key column cannot be used for ordering (DataStax, Inc., 2019b; J. Miller, 2014).

3.5.3. Interacting with data

In order to manipulate Cassandra only provides three possible methods (Lakshman & Malik, 2010):

- insert(table, key, rowMutation)
- get(table, key, columnName)
- delete(table, key, columnName)

All having in common that the entire primary key has to be specified in order to interact with the data. The only exception hereby is the getting of data where only the partition key has to be specified.

Important to note is that there is no interaction to update a data entry. The reason for that is that as Cassandra is optimized for high write throughput is is very costly to read any data before writing. This means that the update and insert known from SQL will perform the same action on the data (Cannon, 2012; Meng, 2014):

```
/* Inserting Data */
INSERT INTO Person (lastname, name, email)
  VALUES ('Doe', 'Jane', 'jane@example.com');

/* Updating Data */
UPDATE Person SET email = 'jane@example.com'
  WHERE lastname='Doe' AND name = 'Jane';
```

As getting and deleting data is also similar to SQL there is no need to go into it any further in this section (DataStax, Inc., 2019h; Meng, 2014):

```
/* Selecting Data */
SELECT * FROM Person
  WHERE lastname='Doe' AND name = 'Jane';
/* Deleting Data */
DELETE FROM Person
  WHERE lastname='Doe' AND name = 'Jane';
```

3.6. Local reads and writes

In order to perform the requested changed to the data they have to be written into the database. This section will take a look into how the changes will be written on a single node not taken into account the cluster.

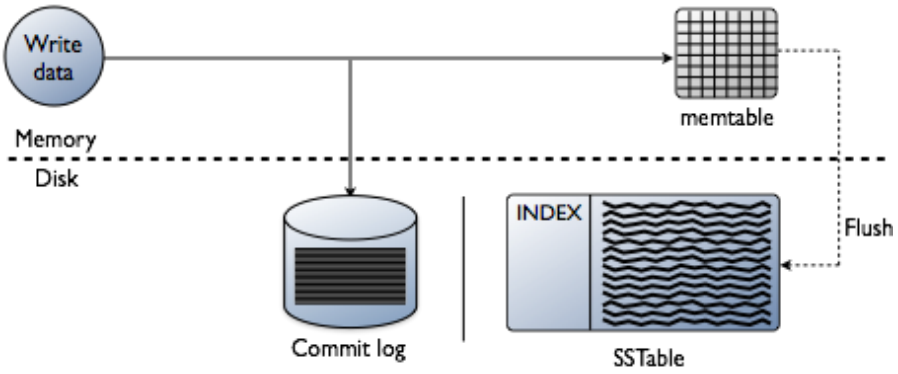

Figure 3.8.: Writing data to Cassandra node (DataStax, Inc., 2019f)

In figure 3.8 it can be seen that the write process to Cassandra involves three steps:

1. **Write to journal** Hereby the query is simple append to the journal on the disk, making it persistent even if the node goes down. As this action is a simple append it is very fast and leaves the data in a temporal order in the journal.

2. **Write to memtable** After writing to the journal the change is performed in the memtable putting the data into a Sorted String Table (SSTable). This form is the same form in which the data will be written on disk. Here it is important that only the required data is written if there are any columns not specified it will not be written to the memtable. In contrast to writing `NULL` to a column which will delete it by setting a tombstone on it (See Appendix A).

3. **Flush to disk when memtable is too big** This allows to simply flush the data and some metadata to the disk when it gets to big for the memory to hold it. Hereby a new data file is created, not touching any of the previously written files, making this action also quite fast as no lookups have to be performed.

4. **Compacting written datafiles** As writing to disk is only an append and will create a new datafile for every flush, the whole database will be scattered over multiple files with redundant data if entries were written after flushing them to disk. In order to merge these files it is possible to compact the files into one resulting in faster reads later (DataStax, Inc., 2019g).

After writing the data it also can be read again as shown by figure 3.9:

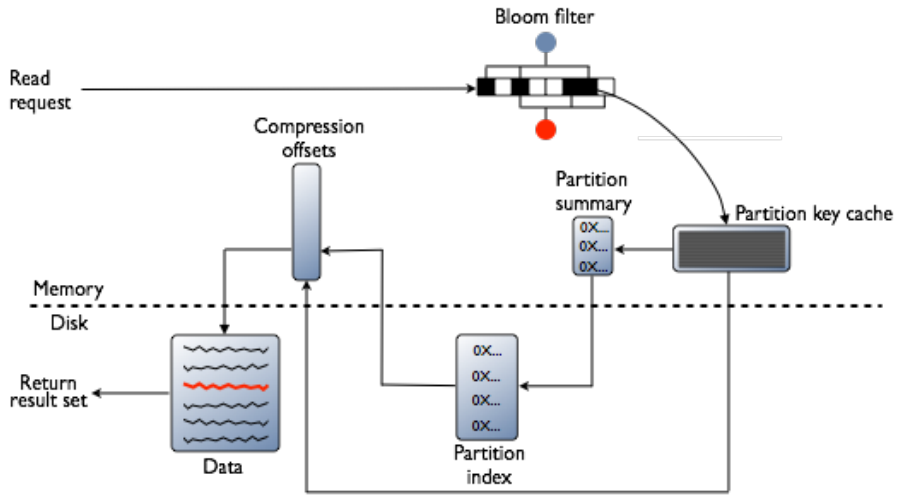

Figure 3.9.: Reading data from Cassandra node (DataStax, Inc., 2019e)

1. **Check caches** First the last query cache will be checked. Returning the data right away if it was requested in the near past.

2. **Check memtable** If not found in the caches the memtable will be checked whether it has the most recent activities on the requested data.

3. **Find SSTable and location** If no entry was found in the memtable the data on disk will be checked by firstly determining in which memtable dump the dataset will be and then retrieving it from there. For a detailed overview of this take a look at figure 3.9

4. **Merge with memtable** If it was necessary to retrieve the data from disk the data will be written to the memtable to allow later queries on the same data to succeed earlier.

3.7. Cluster Architecture

Which node has a certain piece of data is not determined by a master server. Any node has the ability to determine where a particular piece of data is or must be stored just by hashing the partition key of a row. The result of that calculation is called a token. All nodes are placed somewhere on a *token ring* (see figure 3.10) and store the data of the tokens they are assigned to.(Lakshman & Malik, 2010, p. 2) (DeCandia et al., 2007, pp. 209, 210)

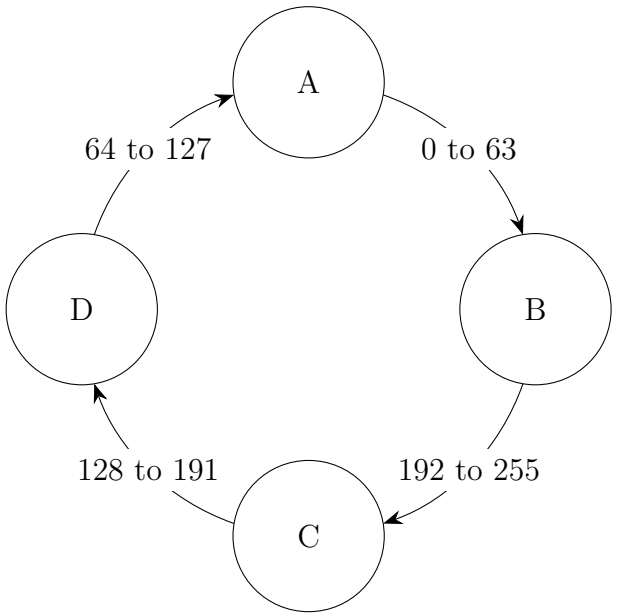

Figure 3.10.: Token Ring

The tokens from the node's location to the next node belong to it. That means if the hashing of a partition key results in a token between 0 and 63 the data will be written to or read from node A. Keep in mind: This doesn't however mean that this node is *in control* of that data - when replication is configured all replicas are equal. The node with that token is just the starting point to determine the first of the replicas.

Replication To ensure that all data continues to be available even if some nodes go down Cassandra replicates the data in the cluster. Replication can be defined for each keyspace when it is created. There is a simple strategy and a network topology aware strategy. (DataStax Inc., 2019a)
The `SimpleStrategy` places the n replicas of a piece of data on the next $n-1$ nodes[1] located clockwise on the ring after the node with the token. (Lakshman & Malik, 2010, p. 3)
The `NetworkTopologyStrategy` strategy tries to be smart about the physical placement of replicas. It needs to be taught about in which datacenter and rack the nodes are located. To avoid losing data when an entire rack or datacenter fails this strategy prefers to spread it out. It is recommended to use this strategy in production, unless all nodes are in a single rack, where both strategies are equivalent.

Rebalancing When a new node is added or a node is removed it is likely that the distribution of nodes on the ring becomes imbalanced. When the ring is imbalanced

[1]The first node that holds the data also counts as a replica.

nodes are responsible for different amounts of data and experience different amounts of load. Since nodes are recommended to be of the same performance level[2] this is not optimal. Some nodes are overloaded and others underutilized. To balance the ring the nodes are moved to equidistant positions on the ring. When this happens data is bound to be assigned to a new owner node - that data will be moved. (Bailey, 2012) The hash function used to transform partition key into is a so called *consistent hash function*. The difference from a regular (cryptographically secure) hash function is that the output wraps around and thus can be represented as a ring. It also has the property that it minimizes the amount of shifting data that is necessary when a node is added or removed. (Karger et al., 1997)

Figure 3.11 visualizes an imbalanced ring and how the green nodes can be repositioned.

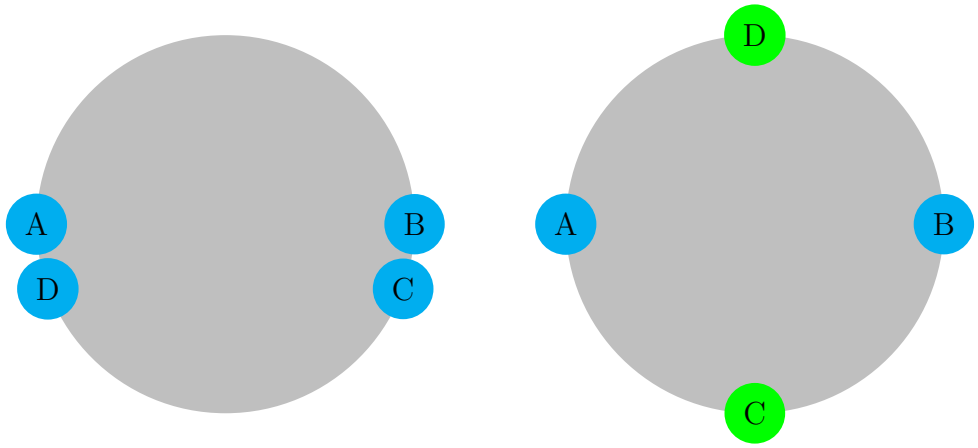

Figure 3.11.: Rebalancing

Nodes A and B can stay at their location, C and D are repositioned. C is moved further along the ring in a clockwise direction. It gives much of the data it was previously responsible to B and takes all of D's data. D is moved past A and takes approximately half of its data.

Virtual Nodes By default each node is placed on the ring just once, when using virtual nodes they each can be placed on the node multiple times. This brings four advantages: (Lynch, 2016; Williams, 2012)

- Nodes can occupy a specified proportion of the ring
- Tokens are automatically calculated and assigned to nodes *rightarrow* No manual rebalancing

[2]Or be balanced by dividing them up into virtual nodes according to performance

- Data hotspots on the ring are handled my multiple nodes
- Rebuilding of replacements is faster

1. When nodes have different processing capabilities and disk space that performance difference should be taken into account so that all nodes receive load which is proportional to their performance. With virtual nodes enabled a node can be assigned any number of tokens. Not the absolute number of tokens of an individual node is relevant it only has an effect on how much of the ring it occupies on relation to the others.
2. Without using virtual nodes the tokens for each node have to be calculated manually and assigned to each node. Cassandra also automatically rebalances the ring whenever a new node joins or a node leaves.
3. When data partitioning was done poorly or the data happens to cluster around a particular token range the node responsible for that data receives disproportionately much load. Virtual nodes remedy that because they not only make the individual partitions smaller but they also make any single note responsible for multiple areas on the ring.
4. When a node goes down and a replacement is brought up. This replacement needs the data of the node it replaced. If the replication factor is three it needs data from three different partitions. Without virtual nodes this can come from up to $3 \cdot (3-1) = 6$ different replicas. When using virtual nodes the partitions the same amount of data is spread across more nodes and thus the replication can transfer data from more nodes at once. If each node is split into k virtual nodes it can replicate from k times more replicas.

See figure 3.12 for a visualization.

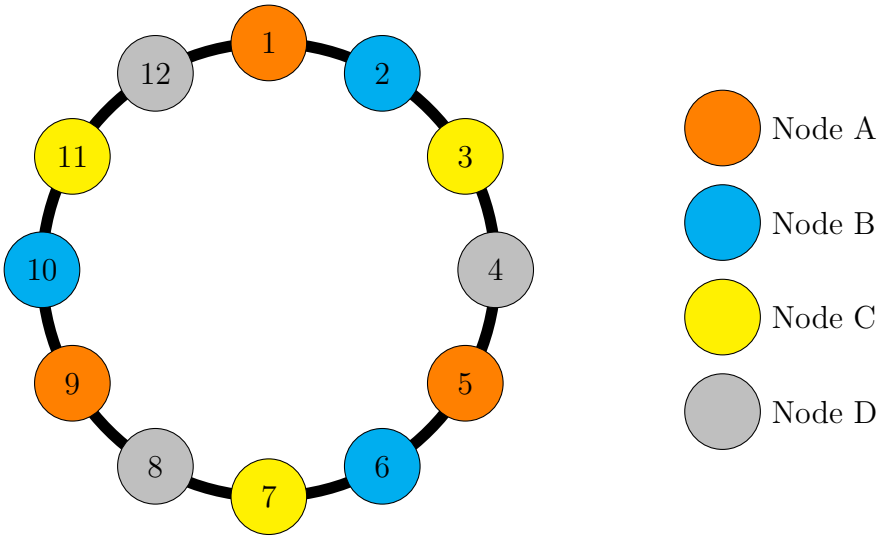

Figure 3.12.: Virtual Nodes

3.7.1. Distributed writes and reads (CAP Theorem)

Cassandra has a masterless architecture; no single node controls any particular piece of data (Lakshman & Malik, 2010, p. 5)[3]. A consequence is that a client can run queries against any node of the cluster. In practice the client determines, either by some heuristic of proximity/latency or a round-robin algorithm, which node to use. For one query that particular node becomes the *coordinator node*; it coordinates the execution and distribution of that query. First it hashes the partition key of the data to obtain the token and finds the node responsible for it. By using the replication strategy it can find out which replicas are responsible for that data as well.
When executing a read query the coordinator asks all replicas[4] for the data. When more than `CL.read` replicas have answered the client is given an answer. When the replicas don't agree on the value of the data the client is sent the newest copy. Once all answers have come in the replicas with outdated information are sent a message on how to update their data, this is called a *read repair*. (DataStax, Inc., 2019c)
When executing a write query the coordinator sends the write request to all responsible replicas. If a replica is not currently available the coordinator logs the request and retries it later when the replica is back up - this is called a *hinted handoff* (Featherston, 2010, pp. 6, 7). After more than `Cl.write` replicas have responded that they successfully completed the write the client is given a successful response. (DataStax, Inc., 2019d)

For an explanation of the consistency levels `CL.read` and `CL.write` see section 3.7.1.

In short, the process looks like this:

1. Client sends query to any node
2. That node becomes coordinator
3. Coordinator determines tokens by hashing
4. Coordinator sends write or read requests to all replicas determined by replication strategy
5. For writing
    a) Return success to client when more than `CL.write` nodes have acknowledged
    b) Write *hinted handoff* to log for nodes that are currently unavailable

[3] Unless explicitly configured to do so
[4] A replica is every node responsible for that piece of data - the one determined by hashing as well as the others determined by replication strategy.

5. For reading

    a) Return newest response when more than `CL.read` nodes have responed

    b) Send `read_repair` to nodes with outdated data

This process is also illustrated by figure 3.13. The client queries node 4 which becomes the coordinator and then itself queries all replicas (1, 8 and 11). When node 1 answers the default consistency level of `ONE` is satisfied and an answer is returned to the client.

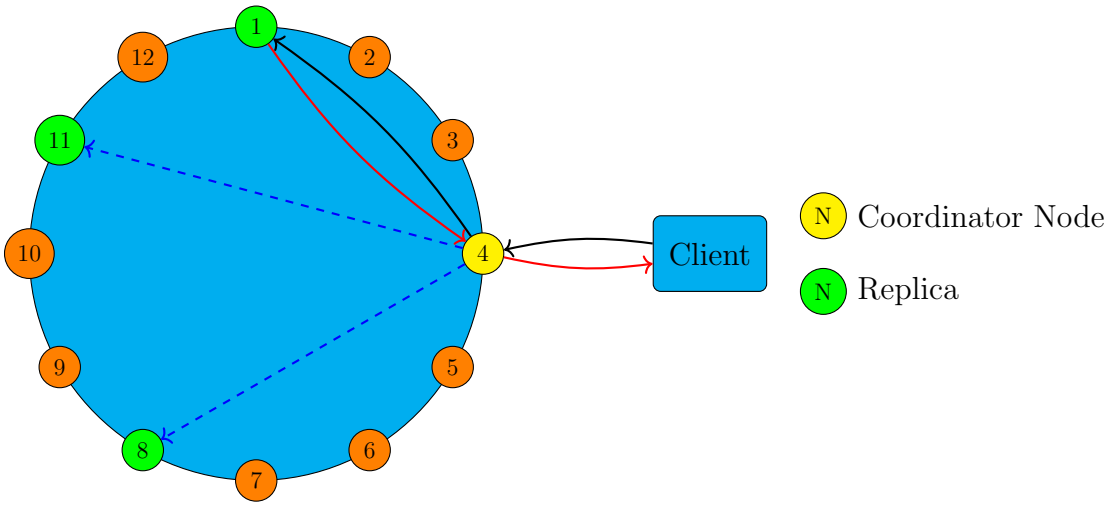

Figure 3.13.: Replication

Tuning Consistency By default writes and reads need to be acknowledged by only a single replica. Usually data is configured to be replicated over multiple nodes. The result is that queries to Cassandra are highly available - any particular node can fail and the request will receive a successful response. This comes at the cost that not all replicas always have the same data - a lack of consistency.
Because different applications have different requirements of availability and consistency Cassandra offers several parameters to adjust its alignment on that spectrum.

One of those is the consistency level. As previously mentioned in this section it determines how many nodes have to acknowledge a request until enough nodes have acknowledged completion. Consistency level for reading and writing are abbreviated as `CL.read` and `CL.write` respectively.

Cassandra offers these, but not limited to these, options. (DataStax Inc., 2019b)

- `ALL` replicas

- `QUORUM`: A majority of the replicas: half + 1
- `THREE` replicas
- `TWO` replicas
- `ONE` replicas (default)
- `ANY`: Only writing - At least one replica or a logged *hinted handoff* if all are unavailable

In addition to these levels Cassandra also offers others that also take into account how nodes are distributed into different datacenters.
For each query, but usually an entire client session, the consistency level can be chosen.

The `ALL` level yields the highest consistency. Only when all replicas have responded the response is given to the client. That means either all were updated or all were asked for their current dataset. Thus there is no uncertainty about what the result is - all replicas must agree. The `ANY` consistency level is the least consistent. The data does not have to be fully written to any SSTable, memtable or even commit log of a replica - a simple *hinted handoff* on the coordinator is enough. (Featherston, 2010, pp. 6, 7)

Replication Factor Whenever creating a keyspace you have to configure its replication strategy. Whatever strategy you choose you have to determine how often you want a piece of data to be replicated. If you go with the lowest possible value of 1 the failure of any node will inevitably lead to data loss (read and write failures). Increasing the replication factor means that more nodes can fail while requests can still be properly responded to - this means increased availability. It will, however, also mean that the additional replicas can get out of sync, which lowers the consistency. (Lakshman & Malik, 2010, p. 3)

We see that by default Cassandra is highly available and only eventually consistent but it can be gradually tuned to being highly consistent but more prone to failure. (Featherston, 2010, pp. 2, 3)

3.8. Setup and Configuration

Cassandra is designed to be run in a cluster of multiple machines. That has to be kept in mind, even when setting up a single instance. That single instance would form a single node cluster. This makes manual configuration unavoidable - every node needs to know

how to join the cluster. For the purpose of joining the cluster each node is configured with a list of *seed nodes*. When starting up for the first time it asks those nodes about the state of the ring and the new node is assigned a place on the ring. After joining the cluster is complete and the distinction of *seed nodes* is no longer relevant - all nodes are equally important. When a new node was added it is responsible for a chunk of the data the others were previously responsible for. They do not remove that, now unnecessary, data on their own - to do so run `nodetool cleanup` on each of the old nodes.

A simple but sufficient[5] configuration of the first node would look like shown in listing 1. (DataStax, Inc., 2019i)

```
# Open socket on this address
listen_address: "192.168.0.2"
# Tell other nodes its reachable on this address
broadcast_address: "3.14.1.59"

seed_provider:
  - class_name: org.apache.cassandra.locator.SimpleSeedProvider
    parameters:
      - seeds "3.14.1.59"

# Enable client communication
start_native_transport: true
```

Listing 1: Configuration of first node

For the masterless cluster to function all nodes need to be able to reach all other nodes. This is trivial if all are in the same subnet of the network. Then they can just reach each other by their IP addresses. When they are in different subnets however they each have a private (local to their subnet) and public (inter subnet) address. This is very often the case in public cloud offerings. There are just not enough addresses available to give every node a public one. Cassandra needs to know two things: 1. On what address to listen for oncoming TCP connections. That's what `listen_address` is for. This is the local/private address. 2. What address to tell other nodes it's reachable under. This is set by `broadcast_address` and it is the public address that all other nodes must be able to reach. Therefore in a local network (with no address translation between the nodes) `listen_address` and `broadcast_address` are set to the same value.

In order to create a single cluster instance the node has to have its seed set to containing only its own address. Since it doesn't need to listen on a public or even private IP this should, in most cases, be `localhost`. The first node of a cluster can be thought of as such a single instance node until others join.

[5]The other settings can be left at their defaults.

In order for additional nodes to join the cluster configure them analogous to the first, changing the address values but keeping the seed list. Upon starting them they will automatically communicate with the seed and join the cluster.

More configuration Cassandra has many more parameters that we are not going to mention here. They can be used to adjust everything from performance tuning, architectural changes or increasing security. To use virtual nodes explained in section 3.7 and give a node more than a single token the `num_tokens` parameter can be used. (DataStax, Inc., 2019i) In its default configuration Cassandra claims a few gigabytes of memory. On dedicated Cassandra nodes you would probably want to increase it to fully utilize the hardware. However for testing purposes you do not want it to consume a big chunk of your memory. Our experience has shown that Cassandra does not like being given too little memory and is prone to crashing if that happens. To adjust how much memory Cassandra uses the `MAX_HEAP_SIZE` and `HEAP_NEWSIZE` variables in `/etc/cassandra/casandra-env.sh` can bet set. (Carpenter & Hewitt, 2016, p. 281)

Setup Overview Setting up a service in a cluster architecture can be daunting because it involves many different components that all need to interact with eachother. The following list gives an overview over the tasks that need to be done:

1. Acquire enough capable nodes. DataStax the main contributor to Cassandra recommends to use at least 3 nodes, 8 cores and 32GB of RAM (Apache Foundation, 2016)
2. Set up a network between the nodes so that each one has a unique IP address and every node can reach every other node.[6]
3. Adapt the configuration files like shown above
4. Open the necessary port in the firewall: TCP/7000 for node to node communication (`storage_port`) and TCP/9042 for client to node communication (`native_storage_port`).
5. Start all seed nodes and once they're online start the other nodes.

When that's done and all nodes have fully joined the cluster you can interact with the cluster with the `nodetool` and `cqlsh` tools.

3.9. Summary and Conclusion

This chapter gave an overview about what Cassandra is, when to use it and detailed insights on how to use it and its architecture. It has shown that Cassandra is a complex

[6] Multiple nodes behind a single NATed public IP is not possible because Cassandra allows only configuration of the broadcast IP but not the port

database system with unique benefits compared to other database systems. It solves a lot of problems that come with big data environments. Due to its uniqueness it has to be assessed very carefully whether Cassandra is a fit for a certain environment or not. Although it comes with major benefits and extra features, some expected features and base assumptions about a database system are different in Cassandra. It could be seen, that it is a great fit for lots of fast writes environments but comes short in OLTP environments where traditional RDBMS databases shine. During the work on this chapter and the learning phase about Cassandra, it showed how detail rich and complex the solutions this distributed database storage brings with is. To truly understand Cassandra in all its different settings, possibilities and to be able to setup, run and maintain a production ready big sized cluster, more reading is needed. Especially the fine details how a Cluster will react in certain distress situations should be fully understood and can be assessed further.

3.10. RethinkDB

Anne Born, Dorian Czichotzki

3.10.1. Introduction

In today's data-driven world, the availability of information in real-time becomes more and more important. Many applications rely on an analysis and delivery of data in a time-frame that is considered as immediate by the user. Be it multi-player games, real-time analytics or connected devices in an IoT infrastructure (Wingerath, 2017). Data should not only be delivered quickly but also proactively, focusing more on a push-based architecture, giving the responsibility to keep the client informed to the server, instead of the conventional pull-based database were the client has to request information each time it is of interest. One kind of technology, implementing this immediate handling of information are so called real-time databases, which are discussed in this chapter by reference to RethinkDB, an example for such a push-based database. The chapter is structured as follows.

First, an introduction to real-time databases in general is provided and a concrete definition of the term in the context of RethinkDB is given. The following sections then discusses the real-time database RethinkDB in great detail, providing information on the architecture, use-cases it is suited for, and most importantly where the technology could be placed in the CAP-Theorem.

3.10.2. Real-time Databases

Like many terms in the IT, real-time databases have an abundance of different definitions. Some overlapping to a great extend, others having nearly no intersection at all. This makes for a need to introduce some of the most common definitions with the goal of establishing a precise interpretation of the term in the context of this book and - particularly - this chapter.

One of the first indications of the term real-time database was within a special issue of the ACM SIGMOD Record on Real-Time Database Systems in March 1988 (Eich, 1988). In an article of this journal called "Issues and Approaches to Design of Real-Time Database Systems" Mukesh Singhal defined the use-case for real-time databases as "applications which have severe performance constraints such as fast response time and continued operation in the face of catastrophic failures" (Singhal, 1988, p. 1). He further mentions timing constraints as a direct derivation of the use-cases mentioned above, laying a fundamental basis for one of the most common definitions of said technology.

This definition (referring to real-time databases as databases that fulfil time-constraints and must meet certain deadlines), also probably the most popular one, is not the one used in the context of RethinkDB.

Instead, the definition used in this book is based on a medium article by Wolfram Wingerath (Wingerath, 2017) who claims that "in recent years, people have come to expect reactivity from their applications, i.e. they assume that changes made by other users are immediately reflected in the interfaces they are using" (Wingerath, 2017). From this statement it can already be derived that he defines the use-cases for real-time databases differently from Singhal. Instead of applications that need fast response time for security reasons, like applications for power-plants or hospitals, he defines a general demand for reactive applications for the sole reason of improved user experience. However, similarly to Singhal, Wingerath also sees a need for new technology besides conventional databases. Conventional databases being pull-based applications that only represent the state of the system at a single point in time. As an alternative to those databases he defines real-time databases as technology that facilitates the push-based handling of changes, "taking view maintenance out of the application layer" (Wingerath, 2017). A real-time databases thus must provide some functionality for clients to 'subscribe' to certain events (changes), receiving updates when the event is triggered. This subscription-feature' is what he calls 'real-time queries'.

Real-time queries can fall into one of two different categories:

- Self maintaining queries
- Event stream queries

The former delivers not only the information that something has changed but also the old and new value. This of course only works if the query is executed again each time a change took place. The latter type of queries only provides the information that a change took place without any more details.

Both queries are push based, pushing information to the client.

Summarised, real-time databases, as defined in this book, are databases that facilitate a push-based method instead of only allowing pull-based query execution. This makes the development of reactive applications much more feasible, since it allows for data updates in a publish/subscribe-like manner.

3.10.3. RethinkDB

After giving an introduction to real-time databases as defined in this book, the document-based database RethinkDB is introduced and discussed in great detail. In order to understand not only the database itself but also its place in the CAP-Theorem this chapter is structured in the following sections: Firstly this section will provide an overview over RethinkDB and its core features, next a short section on the history of the database gives some more insight into the story of RethinkDB and its community, then the proprietary query language ReQL is introduced and explained briefly. Finally the architecture that this database is based on is outlined, including the distribution of RethinkDB over multiple nodes, the functioning of query executions and the storing of data.

Most of the information displayed in this chapter is from the official RethinkDB website (“RethinkDB”, 2019).RethinkDB defines itself as ”the open-source database for the

real-time web” (“RethinkDB”, 2019).

From this description it is already possible to derive two of the most important features of the database project. The first being that it is ’open-source’ meaning that its source code is publicly available[7]. RethinkDB itself is developed and enhanced by a combination of core team developers and database experts as well as over 100000 other contributors from the RethinkDB community, using the open-source aspect to provide their knowledge to the project.

The second feature that the RethinkDB project uses frequently to describe itself is being a database ’for the real-time web’. This means, that RethinkDB falls exactly in the definition of real-time databases established in subsection 3.10.2, providing a function

[7]https://github.com/rethinkdb/rethinkdb

called "changefeeds" (see section 3.10.3) that is push-based and thus automatically updates the client in case of changes.

This feature is already a pretty unique characteristic of RethinkDB. Another one is the fact that the ambitious project was written from the ground up in C++, not relying on and extending existing database implementations. This approach of creating something completely new and "rethinking" databases also lead to the development of a proprietary querying language, that is discussed in section 3.10.3.

History

RethinkDB is originally the product of the eponymous company founded in 2009 with the goal to "bring a breath of fresh air to the database world" (Akhmechet, 2009). At this point the database was not open-source, but also not released as a production ready version yet. With version 1.2 released in November 2012, the company first decided to open-source their project (Team, 2012).

After three more years of development in 2015, RethinkDB 2.0 was released and with it the first production ready version of the real-time database (Paul, 2015). One year later, however, the company was shut down, terminating the official support for the production versions that was offered until then (Akhmechet, 2016). In the blog article announcing the shut down of the company, it was also signalised that the RethinkDB team plans on continuing the development of their product with an open-source continuity plan, keeping it available under an open-source license. Due to this continuous effort of former RethinkDB employees and people from the community to "transition it to a community-driven endeavour" (Glukhovsky, 2017), the source-code was purchased by the Cloud Native Computing Foundation in 2017, who published it under the Apache License 2.0.

As of today the RethinkB project is extended by a large open-source community with version 2.3 being the latest release.

ReQL

As mentioned previously, one of the main features of RethinkDB is its powerful, proprietary query language called ReQL. Since this language is the base of all interaction between the client and the server (the database), it will be discussed in detail in the following sections, explaining its core attributes, the supported data-types and providing examples for different functionalities ("Introduction to ReQL", 2019).

Data Types To facilitate a deeper understanding of ReQLs mechanics, this section gives an insight into the different data types that are supported by the querying language. In the official documentation, RethinkDB distinguishes two main categories ("ReQL data types", 2019):

- Basic data types
- RethinkDB-specific data types

Each of the aforementioned categories is explained briefly.

Basic data types This category includes the data types supported by nearly every programming or query language.

- Numbers
  - This data type includes any real number. Internally values of this type are represented as double precision floating point numbers. This leads to a size of 64 bits.
  - allowed (examples): 2, 5.8276, -13.4
  - not allowed: infinity, NaN
- Strings
  - The data type of string is used the same way it is commonly used in other query languages as well. The encoding used by ReQL is UTF-8, which is why any valid UTF-8 string is also a valid ReQL string. This includes the usage of the null code point (U+0000).
- Booleans
  - A boolean can attain the values true or false.
- Null
  - Nearly every programming language has a data type similar to Null, however the naming may be different from language to language. Since ReQL can be embedded into many different programming languages (see 3.10.3) it might be called something different from Null (e.g.: nil or none). It is important to mention that Null does not equal the number zero (0), the empty set ()

or a string with a length of zero (""). Instead Null is used to emphasise the absence of a value. For example the parent of a root node could be Null, since it does not exist.

- Objects
  - Since RethinkDB belongs to the category of document stores and it more specifically is designed to store JSON documents, any valid JSON object is also a valid object in ReQL. This includes but is not limited to simple key-value pairs *(name: "Harry Potter")* or nested objects.
- Arrays
  - Equally to the documents mentioned above, the array data type of ReQL is also based on JSON arrays, meaning that any valid JSON array is also a valid ReQL array *([1, 2, 3] [] [name: 'Harry', age: 23, wizard: 'Dumbledore', age: 112])*. The maximum number of elements inside an array can be changed at runtime, the default, however, is 100000. But RethinkDBs internal handling of arrays makes them inefficient at large sizes

RethinkDB-specific data types Data types of this category are specific to the functioning of RethinkDB and ReQL. They are not supposed to be used as actual data but instead are either what RethinkDB uses internally to store data or as return values to queries. An exception to this are the so-called 'pseudo types'. Those can be used as actual data types for values defined by the user, but are specific to RethinkDB in a way that they are not commonly implemented in most programming languages.

- Databases
  - This data type describes RethinkDB databases. When using the '.db' operation, this is the return value. A database holds tables and administrative information.
- Tables
  - This data type describes RethinkDB tables. They can be used as selectors. Additionally documents can be added, updated or deleted making tables writable.
- Streams
  - Similar to the aforementioned arrays, streams are lists. However, in order to

make working with large result sets more feasible a Stream is represented by a cursor that points to the result set. This means, that instead of retrieving the entire result at once (like an array) a loop is employed to only retrieve one result set at a time, making it more performant. Another difference between streams and arrays is that the former is read-only and thus can not be changed. This limits the chainability of commands that return a stream.

- Selections
  - Selections are a subset of the table data type. Many table operations (for example `filter` and `get`) return a selection. Selections exist as a counterpart to three different data types: `Selection<Object>`, `Selection<Arrays>` and `Selection<Stream>`. Due to the chainability of ReQL commands the Selections are writable and can be passed to other ReQL operations
- Pseudotypes
  - As mentioned previously the pseudotypes form an exception to the other RethinkDB-specific data types since they can be used with user data similarly to the basic types. Generally they are either special cases of other types or a composition of multiple other types. The pseudotypes include binary objects, times and geometry data types.

ReQL provides its users with the 'typeOf' command, returning the data type of operations (see Listing 2).

```
r.table('wizards').get(1).typeOf()

# returns:
# "SELECTION<OBJECT>"
```

Listing 2: Usage of the TypeOf command

This short description of the data types supported by ReQL already gives an insight to all the possibilities of querying with RethinkDBs proprietary query language. The following sections will extend this insight by introducing some of the many features that ReQL provides.

Features ReQL itself is designed to offer an extensive amount of functionality, enabling users to not only manipulate json documents but also facilitating the usage of common SQL capabilities. To achieve this ambitious goal the language is based on functional programming languages like Haskell and Lips. However, a knowledge of those languages

is not necessary, since ReQL was developed to embed into an abundance of different programming languages. To allow for this natural integration into different languages so called 'drivers' are developed. The official RethinkDB Website groups the existing drivers into three categories:

1. Official Drivers
    - Drivers developed and maintained by the core team of RethinkDB developers
2. Community-supported drivers
    - Drivers developed and maintained by members of the large open-source community of RethinkDB. To be accepted as a community-supported driver, the json driver protocol has to be used and the features of at least Rethink 2.0 ReQL must be supported.
3. Drivers with limited features
    - Drivers developed by members of the open-source RethinkDB community, that do not provide full support of RethinDB 2.0 ReQL.

A list with all available drivers by category (as of April 2019) can be found in Appendix B. To use ReQL in one of the (partially) supported programming languages the user must import the driver the way that is specified by said programming language. Afterwards he/she can construct queries by calling methods from the driver.

Listing Listing 3 shows how this works exemplarily for python.

```
#import the RethinkDB package
import rethinkdb as r
#connect to the server on localhost and default port
conn = r.connect()
#create a table `users`
r.table_create('users').run(conn)
#get an iterable cursor to the `users` table
r.table("users").run(conn)
```

Listing 3: Importing ReQL as a package to python

Of course this embedding of ReQL has limitations. One can not use every feature offered by the 'host language'. This applies to those operations, that have side effects and control blocks. Listing Listing 4 shows the limitations of embedding ReQL with the help of an example. It illustrates two ways of getting all documents that match certain criteria,

one employing operations native to python (if - else) and one employing the 'r.branch' command included in the RethinkDB package.

In this listing only the latter query would work, since the first one is based on python operations with side effects. Other operations and control blocks that can not be used inside ReQL queries include but are not limited to print statements, switch-case statements and loops.

```
# WRONG: Get all wizards older than 30 using the `if` statement
r.db("wizards_world").table('wizards').filter(lambda wizard:
    True if wizard['age'] > 30 else False)

# RIGHT: Get all wizards older than 30 using the `r.branch` command
r.db("wizards_world").table('wizards').filter(lambda wizard:
    r.branch(wizard['age'] > 30, True, False))
```

Listing 4: Limitations of the embedding of ReQL

Additionally, this listing also gives an insight into another important functionality of ReQL: Chaining commands. To construct a query in this query language, a nearly arbitrary number of commands can be chained together, separated with '.'.

To emphasise on this feature and to really illustrate how it works, listing Listing 5 provides a good example for this. First, all documents of a specific table are returned, then by adding '.pluck("last_name")' only the last_name field of each document is returned. This can then be filtered further by adding '.distinct()' to it and so forth.

```
# Get all entries of a table
r.db("wizard_world").table("wizards")

# Return only the last names of the documents
r.db("wizards_world").table("wizards").pluck("last_name")

# Get all the distinct last names (remove duplicates)
r.db("wizards_world").table("wizards").pluck("last_name").distinct()

# Count the number of distinct last names
r.db("wizard_world").table("wizards").pluck("last_name").distinct()
↪   .count()
```

Listing 5: Chaining of commands in ReQL

As previously mentioned ReQL has many powerful features, including the SQL-like

combination of multiple tables with JOINs. For reasons of performance JOINs in ReQL are automatically distributed (see Query Execution).

Now anyone who is familiar with SQL knows that many different JOINs exist in the world of databases. RethinkDB supports four, called inner-join, outer-join, eq-join and zip. The first two work similar to their SQL-counterparts, which is why they are not discussed in greater detail in this book. The last and second last however, are ReQL specific and will be explained with the help of a short example.

```
r.db("demo").table("wizards").eqJoin("house",
↪  r.db("demo").table("houses"))
# Example output only eq_join (one document)

{

    "left": {
        "haircolor": "brown" ,
        "hobbies": "Collecting magical creatures" ,
        "house": "d20dda48-6112-499b-ac2f-5288faa6bb4f" ,
        "id": "a7892a90-a211-493a-a194-734d12387741" ,
        "name": "Newt Scamander"
    } ,
    "right": {
        "animal": "badger" ,
        "house": "Hufflepuff" ,
        "id": "d20dda48-6112-499b-ac2f-5288faa6bb4f" ,
        "score": 60
    }

}
```

Listing 6: The Eq-Join function in ReQL

```
r.db("demo").table("wizards").eqJoin("house",
↪  r.db("demo").table("houses")).zip()
#Example output with zip() (one document)

{

    "animal": "badger" ,
    "haircolor": "brown" ,
    "hobbies": "Collecting magical creatures" ,
    "house": "Hufflepuff" ,
    "id": "d20dda48-6112-499b-ac2f-5288faa6bb4f" ,
    "name": "Newt Scamander" ,
    "score": 60

}
```

Listing 7: The Zip function in ReQL

The 'eq_join', as any other join, has the goal of combining two tables with the help of a common indicator. In this case, a function can be applied to any field of the left-hand table and is then matched to the right-hand tables primary keys or secondary indexes. Since ReQL automatically adds a primary key for each document in a table, users have the option of declaring other fields as secondary indexes if needed, speeding up many read queries ("Using secondary indexes in RethinkDB", 2019). The 'eq_join' command is more performant than the inner or outer join and operates more efficient.

Considering listings Listing 6 and Listing 7 one can not only see an example of an eq_join query, but also an example output. This output of course only displays one entry of an array of documents that are returned as a result of this query. When taking a look at the first query of this listing, it becomes apparent, that the output is split into a left and right side. Each side contains the matching document of the respective table.

The second query of the listing Listing 6 shows how the two parts can be merged together with help of the zip function. This function is "used to 'zip' up the results of a join by merging the 'right' fields into 'left' fields of each member of the sequence" ("ReQL command: zip", 2019).

Of course, it is way out of scope for this chapter to provide a comprehensive guide on all of ReQLs many functions and features. However, RethinkDB has an extensive and well-written documentation, enabling users of all skill levels to write ReQL queries [8]

[8] https://www.rethinkdb.com/docs/

Architecture

RethinkDB is supposed to be used in a cluster configuration. Despite the database being capable of running on a single server, many of the replication features, described below, will only be available in a cluster. The Raft algorithm is used to distribute cluster configuration between nodes.

Sharding & Replication Data is partitioned into shards (see "RethinkDB Architecture FAQ", 2019). Each table in the database can have up to 64 shards. When a table gets sharded the data is divided into a specified amount of equally sized ranges. The primary key of each document determines the shard a document will be stored in. After shard creation there is no automatic re-balancing mechanism. Therefor re-balancing activities must be initiated manually.

Shards can be replicated over multiple nodes. A set of replicas always has one primary shard. The primary shard is responsible for all write and read operations on the documents it holds. When a table is configured for replication, RethinkDB applies a set of heuristics to distribute the shards over multiple systems. Alternatively the Database can be configured to distribute shards to specific servers in the cluster by facilitating server tags. Server tags are identifiers for one or a set of servers.

Query Execution Query Execution is a multi-step process that can be described as follows.

1. One server in the cluster receives a query from a client and creates an execution plan in form of a stack. Lower operations on the stack retrieve data. Higher operations transform the data.

2. The stack is distributed to every relevant server in the cluster for execution. All participating nodes can operate in parallel and stream their results upwards the execution stack.

3. The results are aggregated and sent to the client

To enable concurrent access, RethinkDB uses a copy-on-write mechanism for documents that are already read by an other client. A write operation will lead to consistent results just for a read operation that was started after the write process finished completely. If multiple clients try to write to the same document simultaneously, exclusive block-level locks are used. The client does not notice or manually control the lock mechanism.

Atomicity of operations Most operations changing a document are atomic in nature. An update operation can be finished in a single operation as long as the execution is proven to be deterministic. Exceptions in RethinkDB are operations that facilitate JavaScript operations, random values and sub-query execution.

Data Storage Data can be stored using many common file systems. The most performant solution is too use direct disk I/O. Data in the system is organised in B-Trees and managed by an integrated storage engine. The engine is capable of automatic garbage compaction and is inspired by Better File System (modern copy on write filesystem) (BTRFS).

Change feeds

The main feature of RethinkDB is to actively push changes to a client, when these occur. In ReQL this behaviour can be triggered by using the `changes()` operation in a query("RethinkDB Changefeed Docs", 2019).

To receive changes on a table called *Houses*, one can execute the following query:

```
r.table("Houses").changes();
```

The query would result in a constant stream of changes in the form shown in Listing Listing 8. For more efficient use only changes are transmitted.

```
{
"new_val": {
    "id":  "0539dcdb-f0f7-482c-9721-ead006dfef30" ,
    "name":  "Gryffindor" ,
    "points": 0
    },
"old_val": {
    "id":  "0539dcdb-f0f7-482c-9721-ead006dfef30" ,
    "name":  "Gryffindor" ,
    "points": 202
    }
}
```

Listing 8: Form of a single change response

The change feed feature can be used with many different other features of the query

language and does not have to be applied to the end of the query all the time. The next example plucks the output of the change feed to only show the *name* and *id* property of a changed document.

```
r.db('Demo2').table('Houses').changes().pluck('name', 'id');
```

More complex queries sometimes require some preconditions to be fulfilled. The `orderBy()` operation needs the property that is used for sorting to be indexed. Also the query needs a limit, because change feeds don't work on potentially infinite streams of data. A working query would look like this

```
r.db('Demo2').table('Houses').orderBy({index:'points'}).limit(3)
↪  .changes({includeOffsets:true});
```

The `changes()` function can take multiple options. One of them can be seen above. The *includeOffset* option leads to a response with offset parameters. These parameters can be used to keep local copies of the list in the right order.

Additionally change feeds can be squashed. Squashed feeds buffer change data and trigger change commands less frequent. This greatly improves performance on the client and server.

3.10.4. Reflection

Cap Theorem

In general RethinkDB does value Consistency over Availability (see Listing Figure 3.14). The database guarantees that a write operation is distributed to all systems before the change can be read, like described in section 3.10.3. Therefore any normal read operation will always return the cluster wide consensus at any give time.

The normal behaviour can be changed on a per query basis. A user can specify that the given request should favour availability. In such a case the database returns the next available record regardless of it being on a primary replica. A query using this behaviour can be created as follows

```
r.table("Houses").run({readMode: "outdated"});
```

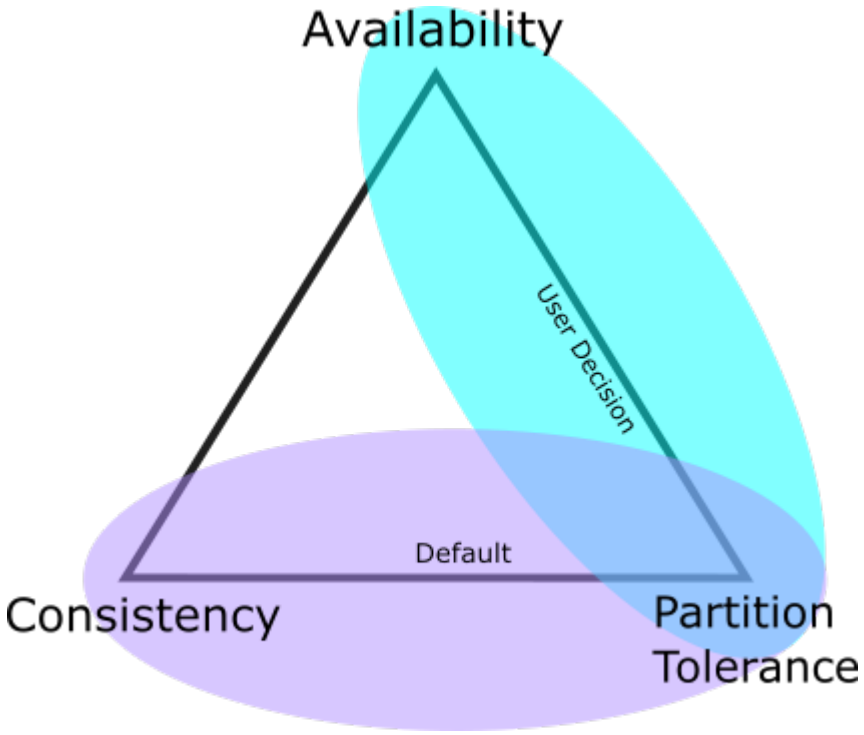

Figure 3.14.: Overview of the CAP Theorem in RethinkDB

Performance

In a test conducted by the RethinkDB core team the developers concluded that the database scales horizontally in a near linear fashion ("RethinkDB Performance Report", 2019) as can be seen in Figure 3.15. The developers further imply that there will be ongoing efforts to improve upon this results.

Comparison to real-time sync APIs

The functionality of RethinkDB can be compared to many real-time sync API services like Pusher[9] or Firebase[10]. There are some key differences between those kinds of systems and RethinkDB.

The capability to store information is the most noticeable difference. Sync APIs don't store any data. They are intended to send uniform event streams to a large amount of recipients. Other key differences can be taken from Table 3.1.

[9]https://pusher.com
[10]https://firebase.google.com/products/realtime-database/

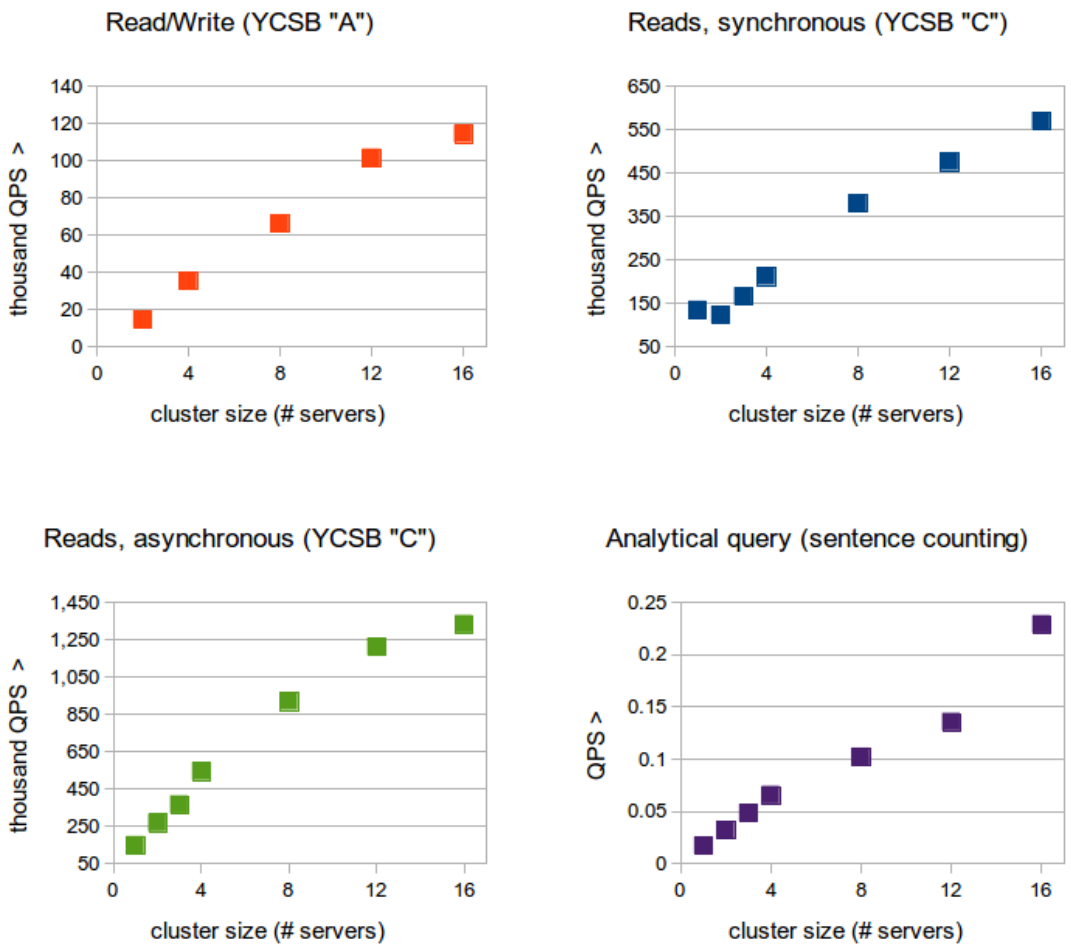

Figure 3.15.: Overview of performance test results("RethinkDB Performance Report", 2019)

Use Cases

RethinkDB is useful in any situation where data needs to be stored and distributed at the same time("RethinkDB FAQ", 2019). The following list contains major use cases for the database.

- Collaborative web and mobile apps
- Streaming analytics apps
- Multiplayer games
- Real-time marketplaces

Table 3.1.: Comparison of RethinkDB and real-time sync APIs

RethinkDB	Real-time sync APIs
Open source → Can be deployed on any infrastructure	Cloud Service
General Purpose Database System → Can run complex queries	Syncing documents → limited querying
Designed to be accessed from an application server → More complex to set up → More flexible → Works for sophisticated applications	Access directly over browser → Easy to setup → Limits flexibility

- Connected devices

There are some use cases RethinkDB should not be applied to, gathered in the following list.

- Applications that need full ACID support
- Strong schema enforcement is required
- Computationally intensive analytics
- High write availability is critical

Conclusion

Working with RethinkDB we found the documentation to be extraordinarily comprehensive. Every major feature could be found described in great detail and supported by multiple examples. Nevertheless there is some confusion concerning the term of real-time databases, since many different definitions exist. This made it harder to evaluate the topic in general and specifically in connection with RethinkDB. This was complicated even further by the fact that RethinkDB in fact 'rethought' databases, delivering a product that can hardly be forced into predefined categories.

However, due to this impossibility to classify the database, RethinkDB was very refreshing to work with, providing many new features and combining the advantages of multiple different database types.

Working with the Database proved to be very convenient. The documentation made it

easy to get used to the specifics of RethinkDB. We had no chance to use all the features ReQL gives the user, due to it having to many and specialised features. We also were not able to test the software in a cluster configuration. Therefore testing the behaviour with sharding and replication setting applied was not possible.

In general we found RethinkDB very useful and where able to identify fitting use cases while testing the database.

We think that not only real-time databases but also RethinkDB will become more and more important in the near future, with big data and IoT redefining the world of databases.

4. Graph Databases – Neo4j

Thore Krüss, Lennart Purucker, Johanna Sommer

4.1. 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.

4.2. 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 (J. J. 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 4.3, 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 4.4, 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 4.5 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.

4.3. 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).

4.3.1. 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 4.1 for an example.

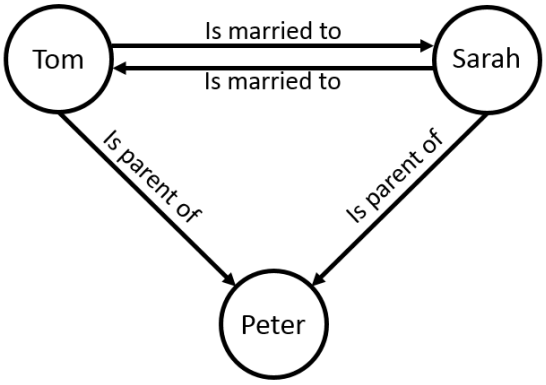

Figure 4.1.: A graph representing a small family. Tom, Sarah and Peter are entities and thus represented as nodes (circles). The two-way relationship between Tom and Sarah, their marriage, is depicted as two directed edges. Lastly, Tom and Sarah are the parents of Peter and thus both have the relationship "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 4.2 shows a labeled property graph.

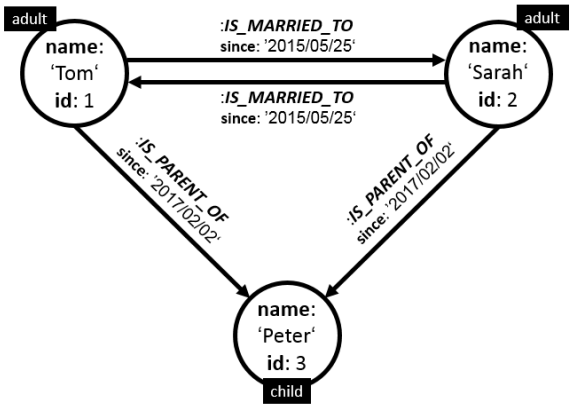

Figure 4.2.: A labeled property graph representing a small family (Eifrem, Webber, & Robinson, 2015; Lal, 2015). Compared to figure 4.1 additional information can been stored.

Concepts of the property graph

Nodes: Like a normal graph, the property graph represents entities as nodes. The nodes in figure 4.2 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 4.2, 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 4.2 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 4.2, `"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) .

4.3.2. 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 (4.3.4).

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", 4.3.3). 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.

4.3.3. 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.

4.3.4. 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.

4.4. 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.

4.4.1. 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.

4.4.2. 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 4.3 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.

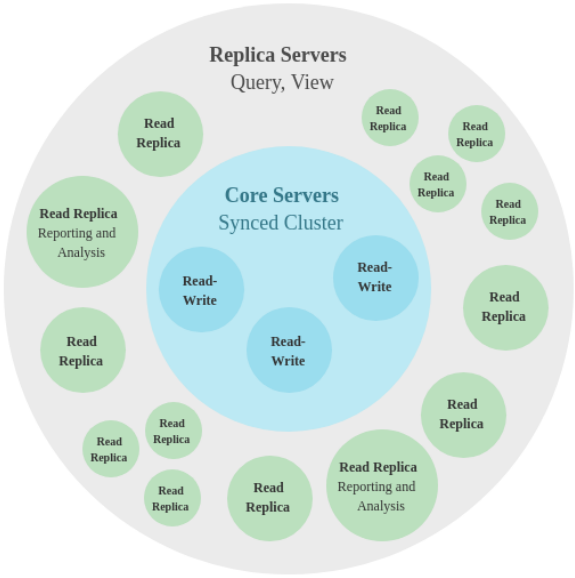

Figure 4.3.: Neo4j causal Cluster Architecture ("The Neo4j Operations Manual v3.5", 2019, Chapter 5.1)

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 4.4.2 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 4.4 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.

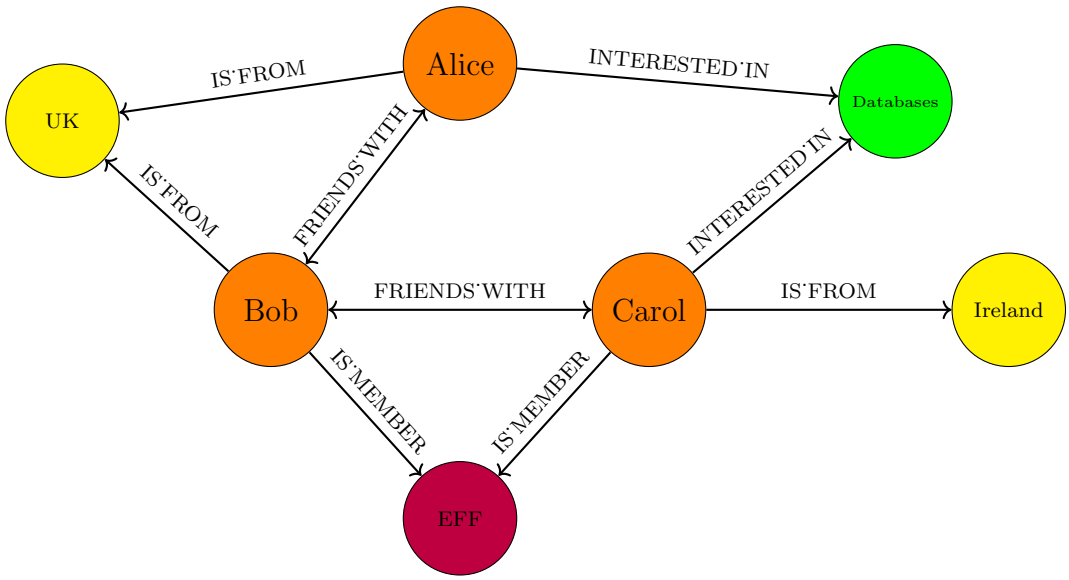

Figure 4.4.: Sample database schema

Implementation in Python

While it is possible to manage the database utilizing the CRUD-functionality from Neo4js own query language Cypher (see 4.4.2) 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 9: Example graph database model with neomodel

The implementation of the graph model mentioned in Figure 4.4 in Python has been realized in Listing 9 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 10.

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

Listing 10: 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 11 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 11: Connecting a person and a country node

Retrieving one or more existing nodes can be done by filtering as shown in Listing 12.

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

Listing 12: 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).

4.4.3. 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.

4.5. Reflection

4.5.1. 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 4.3.4 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.).

4.5.2. 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

Akhmechet, S. (2009). Rethinkdb: A new kind of database. Retrieved April 9, 2019, from https://www.rethinkdb.com/blog/rethinkdb-a-new-kind-of-database/

Akhmechet, S. (2016). Rethinkdb is shutting down. Retrieved April 9, 2019, from https://www.rethinkdb.com/blog/rethinkdb-shutdown/

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*.

Apache Foundation. (2016). Hardware choices. Retrieved April 17, 2019, from http://cassandra.apache.org/doc/latest/operating/hardware.html

Bailey, N. (2012). Balancing your cassandra cluster. Retrieved March 30, 2019, from https://www.datastax.com/dev/blog/balancing-your-cassandra-cluster

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

Cannon, P. (2012). What's New in CQL 3.0 — DataStax. Retrieved March 30, 2019, from https://www.datastax.com/dev/blog/whats-new-in-cql-3-0

Carpenter, J., & Hewitt, E. (2016). *Cassandra: The Definitive Guide*. O'Reilly Media. Retrieved March 30, 2019, from http://shop.oreilly.com/product/0636920043041.do

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, *26*(2), 4.

Chebotko, A., Kashlev, A., & Lu, S. (2015). A big data modeling methodology for apache cassandra. In *2015 ieee international congress on big data* (pp. 238–245). IEEE.

Cockcroft, A. (2011). Replacing Datacenter Oracle with Global Apache Cassandra on AWS. Retrieved March 30, 2019, from https://www.slideshare.net/adrianco/migrating-netflix-from-oracle-to-global-cassandra/28-Remote_Copies_Cassandra_duplicates_across

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

database.guide. (2016). What is a Column Store Database? Retrieved March 30, 2019, from https://database.guide/what-is-a-column-store-database/

DataStax Inc. (2019a). Data replication. Retrieved April 16, 2019, from https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbArch/archDataDistributeReplication.html

DataStax Inc. (2019b). How is the consistency level configured? Retrieved April 16, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlConfigConsistency.html

DataStax, Inc. (2019a). CREATE KEYSPACE — CQL for Apache Cassandra 3.0. Version 3.0. Retrieved April 14, 2019, from https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateKeyspace.html

DataStax, Inc. (2019b). CREATE TABLE — CQL for Apache Cassandra 3.0. Version 3.0. Retrieved April 14, 2019, from https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateTable.html

DataStax, Inc. (2019c). How are read requests accomplished? — Apache Cassandra 3.0. Retrieved March 30, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsRead.html

DataStax, Inc. (2019d). How are write requests accomplished? — Apache Cassandra 3.0. Retrieved March 30, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsWrite.html

DataStax, Inc. (2019e). How is data read? — Apache Cassandra 3.0. Retrieved March 30, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html

DataStax, Inc. (2019f). How is data written? — Apache Cassandra 3.0. Retrieved March 30, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html

DataStax, Inc. (2019g). nodetool compact — Apache Cassandra 3.0. Retrieved March 30, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCompact.html

DataStax, Inc. (2019h). SELECT — CQL for Apache Cassandra 3.0. Version 3.0. Retrieved April 14, 2019, from https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlSelect.html

DataStax, Inc. (2019i). The cassandra.yaml configuration file. Retrieved March 30, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html

DataStax, Inc. (2019j). USE — CQL for Apache Cassandra 3.0. Version 3.0. Retrieved April 14, 2019, from https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlUse.html

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

DOE/Fermi National Accelerator Laboratory. (2010). The sensitive giant. *US Department of Energy Science News*. https://www.eurekalert.org/features/doe/2004-03/dnal-tsg032604.php.

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

Eich, M. (Ed.). (1988). *SIGMOD Rec. 17*(1).

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

Featherston, D. (2010). Cassandra: Principles and Application. Retrieved March 30, 2019, from http://disi.unitn.it/~montreso/ds/papers/Cassandra.pdf

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

Glukhovsky, M. (2017). Rethinkdb joins the linux foundation. Retrieved April 9, 2019, from https://www.rethinkdb.com/blog/rethinkdb-joins-linux-foundation/

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

Introduction to reql. (2019). "RethinkDB". Retrieved April 16, 2019, from https://www.rethinkdb.com/docs/introduction-to-reql/

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.

Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., & Panigrahy, R. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Stoc* (Vol. 97, pp. 654–663).

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

Kohli, S. (2014). Cassandra at Apple for Massive Scale. Retrieved March 30, 2019, from https://www.youtube.com/watch?v=Bc4ql9TDzyg

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.

Lakshman, A., & Malik, P. (2010). Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

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/

Lynch, J. (2016). Monitoring cassandra at scale. Retrieved March 30, 2019, from https://engineeringblog.yelp.com/2016/06/monitoring-cassandra-at-scale.html

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

Meng, A. (2014). Cassandra Query Language (CQL) vs SQL – Alex Meng – Medium. Retrieved March 30, 2019, from https://medium.com/@alexbmeng/cassandra-query-language-cql-vs-sql-7f6ed7706b4c

Miller, J. (2014). Introduction to cql and data modeling with apache cassandra. Retrieved March 30, 2019, from https://www.slideshare.net/johnny15676/introduction-to-cql-and-data-modeling

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/

Patel, J. (2012). Cassandra at eBay. Retrieved March 30, 2019, from https://www.slideshare.net/jaykumarpatel/cassandra-at-ebay-13920376

Paul, R. (2015). Rethinkdb 2.0: Production ready. Retrieved April 9, 2019, from https://www.rethinkdb.com/blog/2.0-release/

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

Qu, F. (2014). Cassandra Best Practices at ebay inc. Retrieved March 30, 2019, from https://www.slideshare.net/planetcassandra/cassandra-summit-2014-39677149

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/

Reql command: Zip. (2019). "RethinkDB". Retrieved April 12, 2019, from https://www.rethinkdb.com/api/python/zip/

Reql data types. (2019). RethinkDB. Retrieved April 8, 2019, from https://www.rethinkdb.com/docs/data-types/

Rethinkdb. (2019). "RethinkDB". Retrieved April 16, 2019, from https://www.rethinkdb.com

Rethinkdb architecture faq. (2019). RethinkDB. Retrieved April 15, 2019, from https://rethinkdb.com/docs/architecture/

Rethinkdb changefeed docs. (2019). RethinkDB. Retrieved April 15, 2019, from https://www.rethinkdb.com/docs/changefeeds/javascript/

Rethinkdb faq. (2019). RethinkDB. Retrieved April 15, 2019, from https://rethinkdb.com/faq/

Rethinkdb performance report. (2019). RethinkDB. Retrieved April 15, 2019, from https://rethinkdb.com/docs/2-1-5-performance-report/

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

Sicoe, A. D. (2012). A Persistent Back-End for the ATLAS Online Information Service. Retrieved March 30, 2019, from https://cds.cern.ch/record/1432912/files/ATL-DAQ-SLIDE-2012-067.pdf

Singhal, M. (1988). Issues and approaches to design of real-time database systems. *SIGMOD Rec. 17*(1), 19–33. doi:10.1145/44203.44205

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

Team, R. (2012). Rethinkdb is out: An open-source distributed database. Retrieved April 9, 2019, from https://www.rethinkdb.com/blog/rethinkdb-12-release/

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

Using secondary indexes in rethinkdb. (2019). "RethinkDB". Retrieved April 12, 2019, from https://www.rethinkdb.com/docs/secondary-indexes/python/

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/

Williams, B. (2012). Virtual nodes in Cassandra 1.2. Retrieved April 16, 2019, from https://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2

Wingerath, W. (2017). A real-time database survey: The architecture of meteor, rethinkdb, parse & firebase. *Medium*. Retrieved from https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87

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" }
        ]
      }
    ]
  }
]
```

B. RethinkDB List of Drivers[1]

Official Drivers

- JavaScript
- Ruby
- Python
- Java

Community Drivers

- C#
- C++
- Clojure
- Common Lisp
- Dart
- Delphi
- Elixir
- Erlang
- Go
- Haskell

[1] https://www.rethinkdb.com/docs/install-drivers/

- Lua
- Nim
- Perl
- PHP
- R
- Rust
- Swift

Limited Drivers

- Objective C
- Scala