Database Implementation

The NoSQL-Database E-Book

at the Cooperative State University Baden-Württemberg Stuttgart

by

IT14C

11. April 2017

Contents

List of Figures

Listings

Part I.

Introduction

Nowadays as more and more data is created from different heterogeneous applications there's a need for a storage system which supports this kind of unstructured data. As Big Data is unavoidable for most global players this topic is getting more relevant today. One solution for handling unstructured data in context of Big Data is the usage of NoSQL databases. This ebook shall give a brief introduction to several NoSQL databases. Its intention is to give the reader an idea of different database types in terms of specific database examples. This ebook is supposed to be an entry point for further investigations by the reader. While it gives important information it neither provides a comparison of the presented databases nor is supposed to be a comprehensive manual. We will introduce some state of the art NoSQL databases. These databases are sorted by their types and every chapter explains one specific database. Small expert groups of two to four students create these chapters. First the main concepts of every database are explained and sometimes special features are mentioned. Finally, advantages and disadvantages are discussed. In this project not many details are mentioned and it is only a short summary of NoSQL databases to have an overview about current technologies. Small implementation examples are provided but for further information's it is necessary to read in bibliography.

Part II.

Column Oriented DB

1. HBase

1.1. Introduction

1.1.1. Column-oriented DBMS

Column oriented databases store data of a table in columns, instead of storing them in rows. This describes the fact how data is saved on the hard disk.

In normal row-oriented DBMS the data is grouped into a bunch of data. Each of that bunch is saved in a separate file, more concrete it contains a few rows with all columns of this table. (Harizopoulos, Abadi, & Boncz, 2009)

This means that a read access needs to read a lot of unwanted data. For example, if it is necessary to access just first names, all data of every row must be read and then discarded, until the name is found. Aferwards it is necessary to read another first name and so on. (Harizopoulos et al., 2009)

Column-oriented DBMS group store each column in a separate file. So for the same example, it is necessary to access two first names: it only needs to open the 'first names'-file and then search these two names. Only other names have to be discarded, instead of the complete row plus the other names. So no extra time is wasted by just reading unwanted data. (Abadi, 2010)

The storage of the files is the main difference between column- and row-oriented databases. Column-oriented DBMS also use traditional languages like SQL to interact with the data. They also use the same underlying organization like ETL (extract, transform, load - used by Data Warehouses) or other tools for visualization. (Amazon web services, 2017) The most expensive operation in a normal database is seeking. And column-oriented DBMS improve it.

1.1.2. Example

The following two tables show the difference between the column and row oriented structure.

Every row in the following list is a separate file.

Row-oriented databases: One row contains an index, the age, first name, last name and a number. The data set is stored in this order on the file. To access the 3rd name it is necessary to read and discard the complete two rows.

- 01: 10, Smith, Joe, 40000;
- 02: 12, Jones, Mary, 50000;
- 03: 11, Johnson, Cathy, 44000;
- 04: 22, Jones, Bob, 55000;

Column-oriented databases: One row could have an index followed by an entry of the column.

- 10:01, 12:02, 11:03, 22:04;
- Smith:01, Jones:02, Johnson:03, Jones:04;
- Joe:01, Mary:02, Cathy:03, Bob:04;
- 40000:01, 50000:02, 44000:03, 55000:04;

If we are searching the 3rd first name in the table it is just necessary to scan the block for the first names, in this case the second file, instead the block with the complete row, including names, address, numbers, account details, etc.

1.1.3. HBase - Introduction

HBase is a column oriented DBMS, with a few differences to the explanation about general column-oriented DBMS in the introduction above.

HBase is designed to manage a lot of data. More precisely the goal is to store very large tables with billions of rows and millions of columns. It is open-source, distributed, versioned and non-relational. (The Apache Software Foundation - Apache HBase Team, 2017) (Hortonworks Community, 2017)

The main difference to other column oriented databases is that one column is not stored in one block and thus in a file. Instead HBase groups some columns in column families. (Abadi, 2010)

These families are organized by the administrator. The administrator has to group often used data combinations like the column first-name and the column last-name into a column family. Absolutely every combination is possible. But of course the combination should be selected with care so that it makes sense in the project. (Abadi, 2010)

So this part is the most elaborating part because only with good design it is possible to pull out the big advantage of HBase.

1.2. Technology of HBase

In order to understand what the advantages and disadvantages of HBase are, it is important to understand how HBase handles and stores data. This chapter shall give a brief introduction into the basics of HBase's storage architecture. Therefore, we should define some of the words we use in this context.

A table is, like in other databases, an ordered set of rows. Unlike other databases, however, HBase does not require a fixed schema for a table. A row contains **n** columns which are connected by a so called row key. Since HBase does not support primary keys, the row key is used for the primary index. Row **a** of table **t** and row **b** of table **t** do not necessarily need to have the same columns. Row **a** might have a column **x** and a column **y**, while row **b** might have column **z**. At this point, HBase does not require null values for columns for which a row does not provide any data, the columns just do not exist for this row. (George, 2011) All columns belong to so called column families. A column family can hold multiple columns and must be defined at the creation time of a table. The column families and the decision to which family a column should belong are crucial for the performance of HBase. Columns of one column family are stored sequentially in one file. Therefore, it makes sense to store columns that are regulary accessed together (e.g. name and surname or height and weight) in one column family to accelerate the reading operations on those columns. (George, 2011) A column has a name, the qualifier. Together with the row key, the column family name and a timestamp (plus additional flags), it forms the final key that maps to the value of a specific column in a specific row. The timestamp is necessary because HBase supports multiple versions of a single cell, for example to have a complete history of the user's last name. (Bertozzi, 2012) (George, 2011)

The image 1.1 on page 7 illustrates this schema.

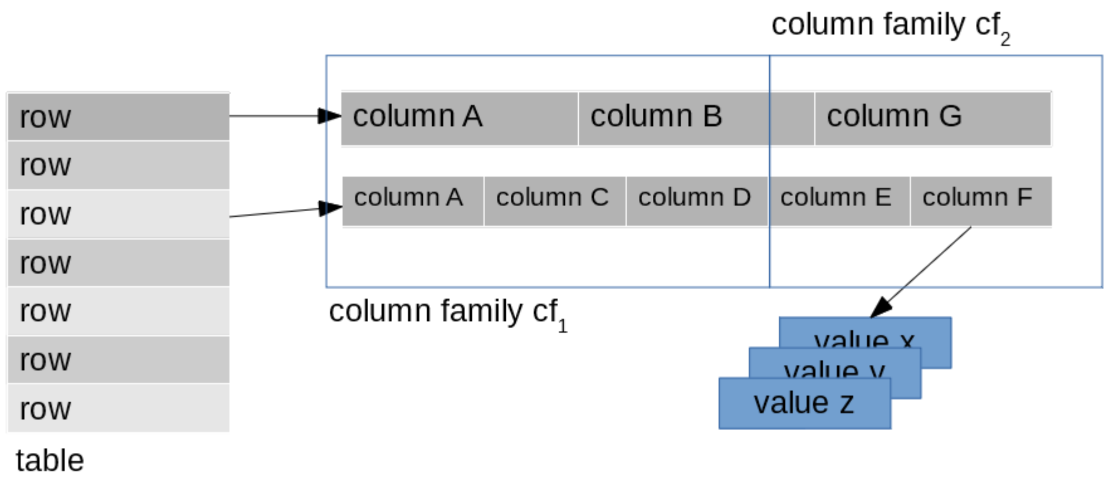


Figure 1.1.: structure of a HBase table

The last paragraph already mentioned the storage of HBase. HBase is a so called column oriented database, therefore it does not store the records row by row but column by column. Since HBase does not have a predefined schema but predefined column families for each table, it stores the values column family by column family. Each column family is stored in a log structured merge tree, a data structure similar to the b+ trees that are used in traditional RDBMs. The log-structured merge tree consists of multiple parts of which each is optimized for the storage medium it is stored on. When data is inserted into HBase, the new entry is inserted into a log file first. After that, the new entry is inserted into an in-memory store. When the in-memory store reaches a certain threshold, it is flushed to the disk as a sorted list of key -> value. This sorted lists are aranged like typical B-trees with an optimization for sequential disk access. During this process a background worker monitors the new files and merges files that contain the same column family together after they reach a certain threshold to avoid jumps on the disk. (George, 2011) (Bertozzi, 2012)

Although HBase's files are usually stored using the Hadoop File System, HBase itself distributes the data as well to allow for better hardware utilization. The data is distributed in so called regions. Regions are ranges of rows that are stored together on one server. This must not be confused with the storage of column families: one file contains the data of one column family, but one column family can be distributed over multiple files. Files that contain the data of the same rows are then stored together in one region. If a region reaches a certain threshold, the region server splits the region into two regions. (George, 2011)

1.3. Advantages and Disadvantages

When evaluating the advantages and disadvantages of HBase it can be revealing to have a closer look at its features. According to (Achari, 2015) HBase's characteristics can be named with the following key words: **sparse**, **distributed**, **multidimensional**, **map** and **sorted**.

The word sparse seems to be a quite negative description for a database, at least in the german translation (dt.: dürftig). It sounds like HBase does not provide all of the important feautures a database should have. But actually the word sparse does not refer to the database itself but to the data stored within it. As described earlier the table schema can be absolutely flexible. The columns don't need to be the same in every row, two rows don't even need to be similar. For that reason there is no need to fill missing values with NULL to keep the schema consistent like it would be nessecary in relational databases. This saves storage space. Additionaly this structure has the advantage that columns can be added or removed at every moment of the database's lifetime. (Achari, 2015)

Furthermore HBase is distributed to several nodes which can be either physically or virtually implemented. When using a physical implementation, the data is stored on several independent (physical) machines, (Wilson, 2008) which are called *Master Server* and *Region Server*. Usually a cluster insists of several Region Servers which store the data (Vohra, 2016) and at least one (up to nine) Master Servers which control the Region Servers within the cluster. (The Apache Software Foundation - Apache HBase Team, 2017) This is HBase's auto failover which makes it reliable and protects against dataloss if one node in the cluster is failing (Wilson, 2008) - on the condition that there has to be more than one Master Server, otherwise there's the risk of a single point of failure if the Master Server fails. (Shriparv, 2014) Auto-sharding explains a bit more in detail how the data is stored on a Region Server. A region is the "basic unit of scalability and load balancing in HBase". And as already explained Chapter 1.2 they simply contain contiguous ranges of rows. When these rows get to big for one Region Server they are split automatically by the system. This is called auto-sharding and permits the dristribution of data to different servers (Region Servers). This allows fast recovery after a server failed, as well as load fine-grained balancing because the regions can be moved between the Region Servers the way it is most suitable (George, 2011)

The terms multidimensional, map and sorted can be explained best in context. Map simply means the structure of key value pairs. Every value saved in the database has one specific key which consists of row key and timestamp among others. And these key value pairs are sorted alphabetically by key. This is important when designing the row key because at this point the administrator can ensure that relating values will be stored together within one region (e.g. two row keys of relating data: dhbw.2017.tinf14c and dhbw.2017.tinf14a). (Wilson, 2008) The conception of the key permits different versions for a value. Due to the timestamp whithin the key it is possible to overwrite an existing value without deleting it. Instead only the latest value is output on a table scan while the other versions are still available (see page 10 Chapter 1.4: Update value). This is called multidimensional and gives the advantage of reproducing updates. (Achari, 2015)

Additionally HBase has the advantage of the Apache Hadoop ecosystem with a lot of features especially for Big Data use cases, such as the MapReduce procedure for analytics. (Shriparv, 2014) The MapReduce procedure was just mentioned as an advantage but it also reveals a disadvantage. HBase does not handle JOINs like it is common in relational databases. Therefore the MapReduce layer provides a remedy. But as well it involves much more effort. A comprehensive knowledge about the Hadoop ecosystem would be necessary to use it effectively. And this fact actually applies to HBase in general. As it is not thought of being used as a standalone configuration the complexity of the Hadoop ecosystem is always there. As already mentioned when talking about sorted keys, it might also be a disadvantage that a row can only be indexed by its key whereas relational databases permit more options or using keys. (Shriparv, 2014)

The HBase book (The Apache Software Foundation - Apache HBase Team, 2017) gives advices when to use or when not to use the database. They definitively point out that it is not suitable for every problem. It is most important to have a huge amount of data. Huge amount in this context means hundreds of millions of rows, *just* a few thousand or even million rows would not be enough to really get advantages from HBase. In this case it is suggested to stay with relational databases. This is for the reason that the cluster would not be made advantage of. Then they stress that HBase is absolutely different from relational databases and most of the common features there are not available in HBase, such as data types, secondary indexes, transactions and advanced query languages. Therefore it is not possible to just migrate a relational database to HBase it is rather a complete redesign. And finally they mention it is important to have enough hardware (minimum about five nodes) which is at least needed for the Hadoop filesystem. Indeed HBase can run standalone on a computer but this case should just be used for development cases. (The Apache Software Foundation - Apache HBase Team, 2017)

Based on the investigation made for this chapter it seems like HBase provides more advantages than disadvantages. This should not be taken as a full fact but rather than a clue. When taking a decision for a NoSQL database it is advisable to compare these directly. This chapter mostly names advantages and disadvantages compared to relational databases. Therefore it might rather be helping to decide wheter a relational database or HBase are more suitable for the use case.

1.4. Example Project

This chapter's intention is to illustrate some of the technical facts discussed earlier. We used a standalone HBase setup on a virtual machine running Linux Ubuntu. For details on the installation process visit www.hbase.apache.org/book.html#quickstart.

The following examples are executed with the HBase Shell, which is included in HBase's binaries. To start the shell execute

```
$ ./bin/hbase shell
```

Listing 1.1: start HBase shell

in the HBase directory.

Create table student

When creating a new table in HBase you need to specifiy the table's name as well as at least one column family. Additionally you can add parameter to specify the table setup.

```
> create 'student', {NAME=>'personal',VERSIONS=>5}, 'school'
```

Listing 1.2: create table

This command creates the table named *student* with two column families *personal* and *school*. The column family personal is configured to store five versions of each column cell. This means that overwritten values are accessible until five overwrites.

Add row

To insert a value you need to provide table name, row key, column family, column qualifier and value.

```
> put 'student', 'Alexandra', 'personal:street', 'Schillerstrasse 31'
> put 'student', 'Alexandra', 'personal:hair', 'blond'
```

Listing 1.3: insert values

The first line adds the value *Schillerstraße 31* to the row *Alexandra* in the table *student*. The column family of this value is *personal* and the column qualifier is *street*.

Scan table

When executing the following command all table entries are listed.

```
> scan 'student'
```

Listing 1.4: scan table

The output will look similar to this:

```
ROW            COLUMN+CELL
Alexandra      column=personal:hair, timestamp=1490775709059, value=
    blond
Alexandra      column=personal:street, timestamp=1490775705439, value=
    Schillerstrasse 31
1 row(s) in 0.0600 seconds
```

Listing 1.5: output of the table scan 1

Update value

The value *blond* should be updated to *pink*.

```
> put 'student', 'Alexandra', 'personal:hair', 'pink'
```

Listing 1.6: update value

Therfor again tablename, rowkey, column family and column qualifier are needed. The output of scan table will look like this:

```
ROW            COLUMN+CELL
Alexandra      column=personal:hair, timestamp=1490776188067, value=pink
Alexandra      column=personal:street, timestamp=1490775705439, value=
    Schillerstrasse 31
1 row(s) in 0.0370 seconds
```

Listing 1.7: output of the table scan 2

Only the most recent value is shown. To get more versions of an updated value a different command is needed.

```
> get 'student', 'Alexandra', {COLUMN=>'personal',VERSIONS=>2}
```

Listing 1.8: get multiple versions

Now two versions of the values stored within column family *personal* are displayed.

```
COLUMN                                              CELL
personal:hair         timestamp=1490776188067, value=pink
personal:hair         timestamp=1490775709059, value=blond
personal:street       timestamp=1490775705439, value=Schillerstrasse 31
3 row(s) in 0.0880 seconds
```

Listing 1.9: output of the table scan 3

Note: This is only possible if the column family is set to more than one version on table initiation.

Types

To demonstate stat HBase does not enforce types, we updated a string value to an integer value.

```
> put 'student', 'Alexandra', 'personal:hair', 3
```

Listing 1.10: put integer into cell

HBase executes this without any (error) message. In fact everything is just stored as byte array. The output looks like this:

```
ROW            COLUMN+CELL
Alexandra      column=personal:hair, timestamp=1490777492998, value=3
Alexandra      column=personal:street, timestamp=1490775705439, value=
    Schillerstrasse 31
1 row(s) in 0.0170 seconds
```

Listing 1.11: output of the table scan 4

Drop table

To delete the table it must be disabled before it can be dropped.

```
> disable 'student'
> drop 'student'
```

Listing 1.12: drop table

1.5. Conclusion

HBase implements new column oriented concepts that Google introduced with BigTable. As a NoSQL database HBase does not need a fixed schema which makes it hard to use when the application is designed for RDBMS. Without the relational schema the key design in HBase is more important than in other databases. This on the one hand provides benefits but on the other hand also increases the effort of designing suitable keys.

As a part of Apache Hadoop HBase supports clustering and distribution on many layers. HBase stores its data distributedly with HDFS while HBase's files are stored over multiple regions and servers. This and the underlying data structures improve the performance when inserting huge amount of data while still obtaining a high speed for analytical operations. Therfore, HBase is suitable for Big Data applications that deal with terabytes of data.

Accordingly HBase is not suitable for relatively small amount of data. The costs of maintaining the Hadoop ecosystem and the additional effort on the key design do not generate substantial value for small applications.

2. Cassandra

2.1. History

Apache Cassandra has been developed by Facebook to solve the so called "inbox search problem" (Finley, 2014, paragraph "The Birth of Cassandra"). Its main use was to query huge amounts of messages efficiently. In 2008 Facebook open-sourced its Cassandra project and replaced it with HBase in 2010. Because of its unique decentralized manner, Cassandra was interesting for a lot of corporations. However, there wasn't a real community around that could provide any support, which made it impossible for companies to use Cassandra in production. A company called DataStax realized the potential of this column-oriented database and started to provide individual enterprise support. This action saved Apache Cassandra from becoming insignificant in the space of NoSQL databases and paved a bright future. Today, Cassandra is the second most popular and the third fastest growing NoSQL database (Finley, 2014).

2.2. Architectural Basics

Apache Cassandra enables users all over the world and across several industries to run their applications with minimal downtime and high scalability. The following chapters outline the architecture of Cassandra and portray the advantages and downsides of Cassandra's architectural approach.

2.2.1. Distributed Architecture

Apache Cassandra is a decentralized database. The database can be distributed across several network nodes. A node is usually a data center with several server racks.
In contrast to most other distributed databases, Cassandra's architecture doesn't need a master server. All nodes running the database are identical. This design has several advantages. First, there is no single point of failure. Cassandra replicates the keyspaces automatically so no data gets lost in case one node stops working. This fault-tolerant approach makes Cassandra well suited for most business critical applications (Edge, 2015) (Mehra, 2015).
Furthermore, all nodes can access the whole database which distributes the payload on the network. This architecture is highly scalable. Additional hardware can be added to the network as one or more additional nodes, whenever more computing power or more storage is needed (Mehra, 2015).

2.2.2. Data Replication

The data stored within a distributed database is usually replicated within the network. This means every data center stores several replicas of the data and these replicas are usually stored on different racks within the data center. Therefore, administrators can define a replication strategy. There are two strategies: The simple strategy and the network topology strategy. In the simple strategy all nodes are treated equally. That means, a previously defined number of replicas are stored on each node and each node stores the same data. In the network topology strategy the number of replicas can be set for each data center individually. The number of replicas shouldn't be greater than the number of racks in the data center (DataStax, Inc., 2016b).
The simple strategy makes it hard to add a new data center to the network, because a new node must initially load all the data plus several replicas. This costs a lot of computing power and puts a high payload on the network. The network topology strategy makes it easier to add new nodes to the network, because an administrator could set a lower

replication number first in order to add the node quickly to the network (DataStax, Inc., 2016b) (Apache Software Foundation, 2016b).

2.2.3. CAP Theorem

NoSQL databases are commonly classified using the CAP theorem. This theorem states that a distributed database can only provide two of the three characteristics consistency, availability and partition tolerance (Trelle, 2011, paragraph "Das CAP-Theorem"). A database is consistent when the same query always returns the same value. Cassandra is only eventually consistent. Especially when time series data is inserted into the database with high velocity, queries might not always return the most recent data. This is a trade off to provide availability and partition tolerance (Edge, 2015). Cassandra is highly available. Since all network nodes are equal, each of them can process user requests and query the entire database. The partition tolerance is also given by Cassandra's distributed architecture. Moreover, the configurable replication increases the partition tolerance. So even if a node stops communication or leaves the network, all the stored data can still be accessed by the other nodes (Parthasarathy, 2013) (Tiwari, 2015).
By default Cassandra is classified as AP. However, it is possible to configure Cassandra to increase consistency. But this comes with the trade off of decreased availability or partition tolerance (Parthasarathy, 2013).

2.2.4. Keyspaces

In Cassandra, a keyspace is an object, that holds together all column families of a design. It resembles the schema concept in relational database management systems. Normally, there is one keyspace per application (Wikipedia, 2016b).

2.2.5. Column Families

One keyspace normally holds many column families, which are comparable with tables in relational databases. These are always tuples that consist of key-value pairs, where the key is mapped to a value that is a set of columns. That means each key-value pair would be one row in a relational database.
There are two types of column families: standard column families and super column families. The standard column family consists only of columns or to be more precise of a unique row key and a number of columns. The super column families are a bit different. Compared to relational databases, a super column family would be something like a view,

a number of tables or also a map of columns. But as super columns are not longer favored and deprecated, it's not meaningful to get deeper in it in the scope of this work (Wikipedia, 2016a).

2.2.6. Advantages

Apache Cassandra can process write operations very fast, because datasets are stored in-memory first and get transformed and permanently saved in read optimized files later. The decentralized manner makes Cassandra fault-tolerant since the databases are replicated across several data centers. It is also easy to add new physical data centers to the network, which makes Cassandra highly scalable. With the configurable replication users can choose whether they want to use less storage or higher fault-tolerance (StackOverflow, 2015).

2.2.7. Disadvantages

The biggest disadvantage of Apache Cassandra is its eventual consistency. As most NoSQL databases, Cassandra doesn't support the ACID property. ACID support would contradict the fast write speed (StackOverflow, 2015).

2.3. Use Cases

In 2014 DataStax analyzed what their customers use Cassandra for. With DataStax being the main supplier for corporate Cassandra support, the outcome of that study can be depicted as representative for corporate usage. However, there are a lot of small projects using Cassandra that are hard to record. Most of the use cases DataStax found fit in one of the following categories (Hasker, 2014).

2.3.1. Product Catalogs

Usually catalogs contain detailed information about all the products a company offers. These products can be physical goods offered by an e-commerce shop as well as digital goods like songs and movies. Cassandra is well shaped for catalogs and playlists, which are also listings of digital goods plus additional information about them, due to the high availability Cassandra provides. This high availability is achieved since the data stored in the database can be accessed by every node within the network. That means the load on the network generated by many user requests gets distributed across the database nodes. Furthermore, Cassandra is highly scalable which is important for companies within e-commerce and similar services (Hasker, 2014) (Oliver, 2014). Popular users of Cassandra for this purpose are Netflix, AOL and Spotify (Hasker, 2014).

2.3.2. Recommendations

In order to personalize the customer experience, many e-commerce platforms analyze their user's behavior and recommend the products they most likely will want to have. But first, these services need to gather a lot of data about their users and store it fast. Cassandra's capability of processing heavy read and write operations supports them (Hasker, 2014) (Oliver, 2014).

2.3.3. Internet of Things

Nowadays, several devices like sensors capture a lot of data and upload it to databases where these data will be analyzed. "[T]he internet of things needs a 'database of things'" (Hasker, 2014). Cassandra can handle such high velocities of time series data, because the input data is first stored in-memory. Later, the data becomes transformed into read optimized files and stored permanently. Moreover, analytics algorithms can run on top of Cassandra even while more data is inserted into the database. This enables fast real-time analytics (Hasker, 2014). Common fields of such analytics are stock markets and connected and self-driving cars (Hasker, 2014) (Oliver, 2014).

2.3.4. Messaging

Initially, Cassandra has been developed to store huge amounts messages and search through them. Facebook distinguished between two scenarios when a user was searching his or her inbox. Either he or she inserts a name into the search bar and wants all the conversations with that person returned or the input is a term that can appear in several conversations with different people. The first is easy to manage since Cassandra manages a key-value store. The key would be the conversation partner and the values are all the messages sent to and received from him or her. The latter scenario describes a column search which all column-oriented databases can handle efficiently (Hasker, 2014) (Oliver, 2014). Furthermore, in messaging availability is way more important than consistency. Users want to see their messages instantly and usually won't recognize a slight delay when receiving messages (Hasker, 2014).

2.4. Setting Up Cassandra

This section covers a high-level instruction on how to install, set up and work with Apache Cassandra.

2.4.1. Installation

The installation of Cassandra is relative easy. First, the Oracle JRE package needs to be added to the system and installed afterwards. After that, the repository source needs to be added to the package manager. Also, to avoid signature warnings, three public keys from the Apache Software Foundation need to be added with the following commands:

```
gpg --keyserver pgp.mit.edu --recv-keys F758CE318D77295D
gpg --export --armor F758CE318D77295D | sudo apt-key add -

gpg --keyserver pgp.mit.edu --recv-keys 2B5C1B00
gpg --export --armor 2B5C1B00 | sudo apt-key add -

gpg --keyserver pgp.mit.edu --recv-keys 0353B12C
gpg --export --armor 0353B12C | sudo apt-key add -
```

sudo apt-get install cassandra finishes the installation and *sudo service cassandra status* shows the installation status. I figured out, that on Ubuntu 16.04 there are some issues while installing, there is some additional software required, that is already pre-installed on Ubuntu 14.04, where it worked fine.

2.4.2. Usage

First, with *sudo nodetool status* it can be checked if the cluster is up and running. If everything is fine there, the connection to cassandra is established by typing cqlsh into the command line. Once connected, all the commands mentioned in the section about CQL work. The usage of Cassandra is easy and self explaining, there is also a documentation page, that is installed with Cassandra and shows several example commands.

2.4.3. CQL

The Cassandra Query Language (CQL) is a language that was developed especially for the use in Cassandra. The common usage is via the CQL shell (cqlsh). With the cqlsh it is possible to create keyspaces and tables, as well as entering some data, update it or

delete it (DataStax, Inc., 2016c). How to do something with the CQL will be explained in the next section.
To get started with using Cassandra, the first is to create a keyspace. This is done by the following command:

```
   CREATE KEYSPACE DHBW
           WITH replication =
           {
            'class': 'SimpleStrategy',
                'replication_factor' : 3
           };
```

With this command, a keyspace called DHBW is created with the replication class SimpleStrategy and a replication factor of 3.
The replication class knows two strategies: SimpleStrategy and NetworkTopologyStrategy. After we created the keyspace, it can be used to switch on the keyspace to work with it. This happens with the command

```
use DHBW;
```

Working on this keyspace a table or a column family can be created in it:

```
CREATE TABLE students (
    mn int,
    firstName text,
    lastName text,
    averageScore double,
    PRIMARY KEY (mn)
);
```

The created table is called students and has four keys, which are mn, firstName, lastName and averageScore. There are different data types to use, in this case integer, text and a double (float) value. Also, mn gets defined as a primary key.
After the table was created, it is possible to enter some data. With the following command two sets of data are entered:

```
INSERT INTO students(mn, firstName, lastName, averageScore)
       VALUES (0, 'max', 'mustermann', 1.0);
INSERT INTO students(mn, firstName, lastName, averageScore)
       VALUES (1, 'john', 'doe', 4.0);
```

Now there are two students with different values.

The thing missing now would be to select some data from the created table. This is, similar to SQL, done with this command:

```

SELECT * FROM students;
```

With that command the two students John Doe and Max Mustermann that have just been entered would be returned with all their data.

To test some more commands, some sample data can be retrieved by installing the Cassandra Dataset Manager, which allows to easily import some sample data, in this case from movielens.

```

use movielens_small;

select * from ratings_by_user;
select * from ratings_by_user where rating=3
        ALLOW FILTERING;
select user_id from ratings_by_user where rating=3
        ALLOW FILTERING;
```

With this code, the keyspace is switched to the movielens keyspace. Then, all ratings are selected by user.
The second command gets all ratings, where the rating is equal to three. The ALLOW FILTERING makes this query possible. Otherwise Cassandra would throw an error saying *Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING..* So when using allow filtering, Cassandra executes the query, no matter if this might take very much time ((DataStax, Inc., 2016a)).
In the third query, demonstrated the strength of Cassandra. In the query, only the user id from ratings by user is selected where the rating is equal to three. In a relational database each line would have to be read to filter for the rating three, then every line with it and get the user id from that line would be returned. However, with Cassandra the only two lines that matter are the user id and the rating, the other values are not checked at all. That's the huge benefit of column oriented databases like Cassandra.

2.5. Lessons learned

2.5.1. Getting Started

The most obvious starting point to get in touch with Cassandra is the Cassandra web page created by the Apache Software Foundation. Unfortunately, this page is declared as work in progress meaning a lot of information is not included yet. DataStax maintains a DataStax Academy platform where they provide some articles on Cassandra as well as webinars and trainings. Not all of that content is for free, but enough to get an overview about what Cassandra is, how it works and how to use it. Furthermore, since a wide community is working with Cassandra, forums like StackOverflow are helpful, too. The world wide web contains a ton of articles and posts about Cassandra. However, they usually depict only a few topics. In order to get a more complete overview of Cassandra, it is helpful to read a book. Authors generally start at a lower level, depending on the addressed audience, and go through a variety of topics step by step. The only downside of books is that it takes a while to write, review and publish them so they can not cover the most recent developments. But in contrast to some pseudonym writing a blog post, the contents of books are more reliable since they are reviewed.

2.5.2. Setup

The installation process of Cassandra didn't take long. There are several tutorials that are quite detailed and step by step, so one can just follow the tutorial and go through the steps. The problem to install it under Ubuntu 16.06 was also hit, but on Ubuntu 14.04, everything worked fine.Most probably, the installation on Ubuntu 16.04 is possible as well. There are several blog posts about these problems, as well as some questions about it in forum threads.
Moreover, there is the DataStax Academy platform, mentioned before. With an account one can also submit a support ticket for DataStax, so this is also a good channel to get help if any specific problem with Cassandra appears.

2.6. Conclusion

The main advantages of Apache Cassandra are its availability, fault-tolerance and scalability. Therefore, it is perfectly suited to run in the back end of business critical applications where consistency and ACID are not required.
To get these advantages, users need several servers in different data centers, which makes Cassandra less interesting to private users. Furthermore, the open-source documentation is poor. Apache's web page for Cassandra is labeled as work in progress. Several important parts of the documentation haven't been written yet (Apache Software Foundation, 2016a).
To sum that up, Cassandra is well suited for corporate use, especially thanks to the DataStax support, and less for private usage.
In the future Cassandra will probably have an important role within the internet of things and real-time analytics. The developer's focus is to suit Cassandra in the way that it runs the database that drives applications but not the analytics itself. It's easy to gather huge amounts of data and store them fast. However, Cassandra is inter-operable with most new technologies like Apache Spark.
All in all, Cassandra is a nice fit for companies that need to manage high amounts of data with a high input velocity. Without a single point of failure, Cassandra's availability is better then most other NoSQL databases.

Part III.

Key Value DB

3. Redis

3.1. Introduction

3.1.1. Key-Value Database

The term “database“ is usually associated with SQL-Databases, tables and strict structure. Every entry in a table has the same attributes which are predefined. But what happens if there is another kind of data? What happens for example, if pictures or videos should be stored instead of numbers and strings? This is the big advantage of key-value databases. Almost any kind of data can be stored - pictures, videos, HTML, JSON, strings, numbers, and many more. Because of this ability, key-value databases have the reputation to be the most flexible NoSQL-databases. There are different key-value databases. The differences are

- the way they save data (changes are firstly saved in-memory or instantly written to the disk)
- the format of values

By the way the databases save data, also persistence and performance of the database are influenced. This, amongst others, will be looked at in section 3.4. What all key-value databases have in common is that they have only one value for one key. But for example one hash with multiple fields is seen as one value.(*Key-Value Database Explained*, n.d.)

3.1.2. Redis

Redis is an open-source key-value database. What is special about Redis is the fact that all data is saved in-memory. This makes Redis the fastest Database. But it makes Redis also very vulnerable. To maintain data persistence, Redis writes changes asynchronously to the disk. On server failure like power loss, all unwritten data is lost and because of the asynchronism not recoverable.(Seeger, 2009) These attributes will be elaborated in the following chapters.

3.2. Technology of Redis

3.2.1. Data Types

Redis has a special way to save the data. In a normal database a string key is connected to a string value, but in Redis it is possible to accessed with a string key to string or other complex data structures. Redis provides several data structures like string, list, set, sorted set, hashes, Bit arrays and HyperLogLogs. (Redis Labs, Inc., 2017a)

Key

To get accessed to a value you need the key of it. Keys in Redis are binary safe. So they have a known length not determined by any special terminating characters. That mean that you can use every binary sequence like “234“, “John“ or a JPEG-File.

String

The String in Redis is the basic kind of value. It is possible to store every kind of data inside a string, because it is also binary safe like the key. The length of the string is up to 512mb. With the basic kind of data the other data types can be build.

```
  > SET testkey examplevalue
(integer) 1
  > GET testkey
examplevalue
```

List

The data type List is just a list of strings. List has in programming languages different meanings and implementation. In this case a list is implemented like a linked list. So every string inside the list has a reference to another string. Because of the implementation of a linked list Redis has a big advantage by adding an element at head or tail. For this operation Redis needs a constant time or this can be expressed as $O(1)$. But on the other hand this brings a bad performance at scanning for an element. This means for accessing an element Redis needs to scan the entire list. Redis Lists are implemented for fast writes, for other operations other data types fit better. (Das, 2015)

Set

Redis Sets are a unordered collection of strings. They look like lists, but they have a different implementation. In Redis Sets are solving problems around the set theory.(Das, 2015)

A Set in Redis can have up to 4 billions of values, each of them is unique. So it's not possible to duplicate values inside a set. The implementation brings a constant $O(1)$ time of adding, removing or checking for existence of an element.(tutorialspoint.com, 2017)

```
  > SSAD testkey examplevalue
(integer) 1
  > SSAD testkey examplevalue2
(integer) 1
  > SSAD testkey examplevalue2
(integer) 0
  > SMEMBERS testkey
 1) "examplevalue"
 2) "examplevalue2"
```

Sorted Set

A Sorted Set in Redis is very much like a Set. So the values inside the Sorted Set are unique but each of them is associated with an floating point value called the score. The elements are sorted by these score from the smallest to the greatest one. While the values are unique the score can be duplicated. If the score is equal the values get sorted lexicographically.(Redis Labs, Inc., 2017a)

Hashes

Redis Hashes are a collection of key-value pairs. They are maps between attributes and values, both are strings. With Hashes it is possible to represent Objects with up to 4 billions attributes.(Redis Labs, Inc., 2017a)

3.2.2. Redis Expire

In Redis it is possible to set a timeout for a key by using the EXPIRE command. When the time to live elapses, the key will automatically be destroyed. The timeout can be cleared by PERSIST, DEL, SET, GETSET and all *STORE commands. Every other command leaves the timeout untouched. Even if you rename a key the new key name has the same time to live. If you put in a negative time to live, the key will be deleted. It is possible to update the time to live of a key by using EXPIRE. (Redis Labs, Inc., 2017b)

3.2.3. Transactions

In Redis it is possible for the user to execute a group of commands in a single step with the commands MULTI, EXEC, DISCARD and WATCH. It guarantees that the transactions are still isolated and atomar:

- Isolated: It is not possible that any other command of another client is executed in between the commands of one transaction.
- Atomar: It is only possible to execute every command of a transaction or none of them are proceeded.

MULTI / EXEC

By using MULTI you can start a transaction. Every following command will be queued. When the user calls EXEC these commands are executed. An array of the replies will be returned in the same order like the commands were entered.

```
 > MULTI
OK
 > INCR foo
QUEUED
 > INCR bar
QUEUED
 > EXEC
1) (integer) 1
2) (integer) 1
```

This example increments the keys foo and bar in one transaction.

MULTI / DISCARD

To abort a transaction you need the command DISCARD. It will flush the queue and exit the transaction.

```
 > SET foo 1
OK
 > MULTI
OK
 > INCR foo
QUEUED
 > DISCARD
OK
 > GET foo
"1"
```

WATCH

In the following example the key mykey is increased without using INCR.

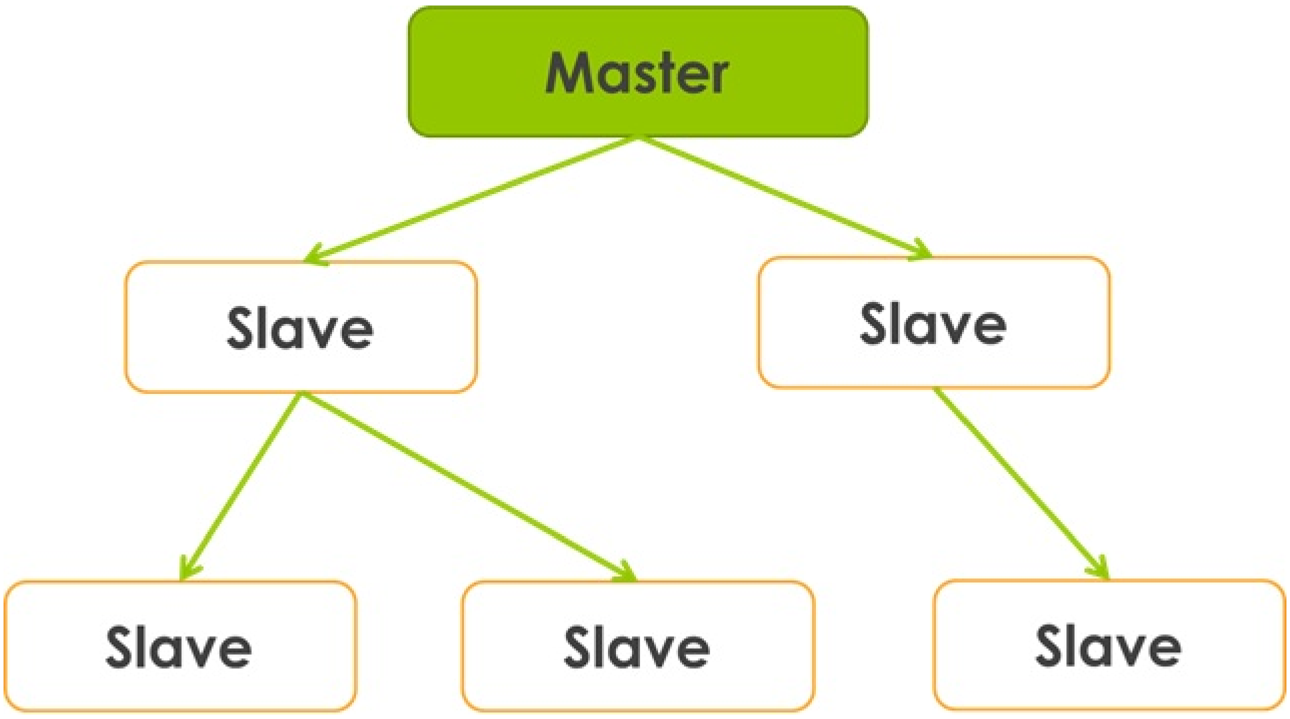
```
val = GET mykey
val = val + 1
SET mykey $val
```

With several clients it is possible to get problems in this example. If two clients want to increase the same key at the same time there will be the wrong result. For example mykey has the value 1. Then both clients read this value and add 1. They both set the value to 2. But the result should be 3 because the key should be increased twice.
To solve this problem Redis has the command WATCH. WATCHED keys are monitored to check changes against them. If one watched key is changed by another client the transaction aborts, EXEC returns null and the user can repeat the operation. UNWATCH, EXEC and DISCARD flush all watched keys.

```
WATCH mykey
val = GET mykey
val = val + 1
MULTI
SET mykey $val
EXEC
```

In this example the tranaction will abort if any other client changes the value of mykey between the WATCH and EXEC command. (Redis Labs, Inc., 2017f)

3.2.4. Replication



(TH Köln, Campus Gummersbach, 2011)

Redis uses a Master-N-Slave-replication. Here are some important facts:

- It is possible to have as many slaves as you want in row or in series.
- The slaves synchronize with the master.
- Redis uses asynchronous replication.
- Replication can be used for scalability (slaves can be used for very slow read operations) or just for data redundancy.
- If the master falls out, one slave can become new master.

(Redis Labs, Inc., 2017d)

3.2.5. Security

Redis is not designed for maximum security. Any trusted client in an trusted environment can have access to the database. That‘s why the administrator should be cautiously while the implementation, because it‘s maybe not the best idea to connect the database directly to the internet or an untrusted environment. But Redis has thought about their lack of security and has implemented some ways to make it a bit safer against attacks. The first thing is to rename or disable operations. The second thing is to enable the authentication layer and another thing is called the protected mode.

Rename Commands

The main idea behind rename commands is that the normal commands are disabled. To work with the database the user needs to know the renamed commands. If the old command is passed trough the database it replies with an error message. All commands can be renamed in the configuration file redis.conf with the directive rename-command. For example the command get is renamed into „specialorder “with rename-command GET SPECIALORDER. With this operation it is also possible to disable commands. This happens also with the rename command inside the redis.conf. The directive rename-order gets as second argument an empty string, rename-command GET SPECIALORDER. Now it is not possible to get accessed to this command. The listing shows how Redis replies to queries. (Jochen Schnelle, 2012)

```
  > GET testkey
(error) ERR unknown command 'GET'
  > SPECIALORDER testkey
"5"
  > FLUSHDB
(error) ERR unknown command 'GET'
```

Authentication Layer

Redis has no access control implemented but it is possible to enable a tiny authentication layer in the redis.conf file. After this layer is enabled Redis will refuse by any query of an unauthenticated client. A client can authenticate itself by sending the AUTH command an the password. The password is set by the administrator in the redis.conf in plain text.(Redis Labs, Inc., 2017i) By choosing a password it should be clear that redis is a very fast database so it is possible to brutforce with about 100 000 attacks per second to break it. (Jochen Schnelle, 2012)

Protected Mode

The protected mode should help users to protect the database from external networks. It automatically enters the special mode if and only if:

- The default configuration is set

- No AUTH password is configured

When Redis is in protected mode it only replies to the loopback interface. Any other query gets an error message. However if the protected mode is needed the administrator can just disable the mode.(Redis Labs, Inc., 2017i)

3.3. Use Cases

In the following you will learn something about four use cases which you can realize with Redis. There are many more but these examples give a deeper look into Redis and explain why choosing it.

3.3.1. Caching

The first use case is called Caching. Caching is important for responsive applications. The advantages of Caching with Redis are easy to say:

First of all it reduces the database traffic if you have fewer database accesses. Without caching you would need to access the database everytime you need to use the data from the database. (Redis Labs, Inc., 2017e)

Caching with Redis is in a way different to Memcache, which is more efficient but a less better choice. Because of the data structures of Redis you gain a lot of power, that it is more efficient.

With allowing key names and values to be as large as 512MB each, its possible to have intelligent caching and manipulation of cached data. Memcache for example limits it to 250 bytes.(Haber, 2016)

If Redis is used as a cache, it can happen, that if you you add new data to the server and the memory limit is reached, the server automatically removes some old data.

To handle this correctly, Redis has got six different eviction policies.

The first one is Noeviction. It returns errors if then memory limit is reached and the client tries to execute commands so that more memory would be used than is reachable.

The second policy is named allkeys-Iru. It tries to remove less recently used (LRU) keys before adding new data. With this policy old data can't be kept, but new data has always space to be added.

The third policy is comparable to allkeys-lru. But the big difference is that it only removes less recently used keys with an expire set. It is called volatile-lru.

The name of allkeys-random explains the next policy in general. If its used, random keys will be removed to make space for new added data.

The fifth policy “volatile-random“ removes random keys with an expire set only if space for new data is needed.

The last policy is called volatile-ttl. It evict keys with an expire set and try to evict keys with a shorter time to live first.(Redis Labs, Inc., 2017g)

3.3.2. Leaderboards

Leaderboards are generally used in many multiplayer games. No matter if it's a browser game, a phone application or an AAA game for PC, Playstation or XBox. For a leaderboard it is important to have live updates and if there are many players, it should be fast to get all items.

Redis offers these advantages and that's why it is used for leaderboards.

The disadvantage behind Redis it's only possible to get query on keys, not on the exactly field. If you want to search for a specific name in the leaderboard you always have to find the key and then you can go further to the name. That's the same with points and other things written in a leaderboard.(Socialpoint, 2013)

3.3.3. Queues

Another use case with Redis are the queues. To explain it there will be shown an example in the following.

An application has got some Producers, to handle tasks. A producer put the tasks on the Redis message queue. This queue is connected with consumers, who can take a task and work on it. While this process is going on, the consumer removes the task from the message queue and puts it onto the Redis processing queue.

It is a kind of a backup queue. There are all tasks who are in process. If there are some issues like network problems or consumer crashes this queue can handle and recover the tasks.

A task can have a timestamp. If the task is currently on the processing queue and it is there for a too long time it can retransferred to the message queue and another consumer can work on it.

This example shows that Redis always uses two queues to handle possible and known issues.(Sabo, 2016)

3.3.4. Publish / Subscribing

The last shown use case is Publish and Subscribe. Redis is perfect if you want to publish a message and you don't know how many and which users will read the message. And on the other hand the users maybe don't know anything about the author / publisher.

As a subscriber it is possible to subscribe and unsubscribe the message.(Redis Labs, Inc., 2017c)

3.3.5. Companies Using Redis

Twitter uses Redis for scaling. Every timeline in Twitter is an index of tweets indexed by an id. In Redis you can access the data by the id not by the name or other values. Twitter is in a way a Publish and Subscribe system. A tweet is a small message but will be send to many timelines and they are large. Scroll down on this timeline will load another tweet. Redis is not used for on disk features, but for key-value stores and for Ads service. It is used in Twitter since 2010.(Scalability, 2014)

Github should be running fast, for this it uses Redis for a persistent key and value store for the routing information. For the normal data it uses MySQL database.(Github, 2009)

Snapchat has a feature where people can post pictures and videos for 24hours to show their friends what they were doing. It's called Snapchat Stories. For caching these stories, Snapchat uses Redis.(Redis Labs, Inc., 2017h)

StackOverflow uses Redis as the distributed caching layer. But the data will be compressed before sending to Redis. Before using Redis, StackOverflow didn't use a separate caching tier.(Scalability, 2011)

Flickr uses Redis as a secondary index for MySQL. It is possible to add and remove a contact, to change a relationship between contacts and to upload and remove user photos. Flickr is connected with Yahoo, so all the user details will be shown on Yahoo as well.(Cohen, 2013)

Many other companies like Airbnb, GitHub, Instagram, Trello, Uber and many more are using it for real time updates and caching.(Techstacks, 2017)

3.4. CAP Theorem

The CAP theorem or Brewer's theorem is a model to categorize distributed systems. CAP stands for:

- Consistency
- Availability
- Partition Tolerance

Consistency means that a read request always returns the most recent write or an error. Availability stands for a guaranteed return of a previous write, even though not necessarily the most recent one. But it never returns an error or a timeout. Partition Tolerance is the part which has to be fulfilled in any case. It means that if the network connection between nodes does not work, the other parts of CAP (consistency and availability) are still fulfilled.(Messinger, 2013)
Because of this databases are divided into CP and AP databases. They cannot be CAP databases since consistency and availability contradict each other in case of a node failure. A read transaction after such a failure will either result in an error or in the return of a former write.(Greiner, 2014)

3.4.1. Redis' Positioning inside CAP

To position Redis in the concept of CAP is rather difficult, because Redis originally was not designed to be a distributed system. As a result Redis is considered to be CP, i.e. consistent and partition tolerant. Because data is written to the disc asynchronously every once in a while, data is very likely to get lost on server failure. This is why Redis is not an AP database.

On the other hand, consistency is not guaranteed if Redis is implemented multi-node. It can be lost if a read operation on a slave is performed. Then the response can contain expired data and overwrite changes on the master which are complete but not yet replicated to the slaves.(Kingsbury, 2013)

The CAP theorem only applies to distributed systems. But as Redis is usually distributed single-node and not designed to be run multi-node, it should not be placed inside the CAP.(Sheehy, 2013)

3.5. Conclusion

Key-Value Databases can save any kind of data. So, no matter if the data is a string or a picture, you can store it. Redis is one of them and has special datatypes. With these datatypes, the user can store any kind of complex data among lists or hashes. The in-memory database Redis is implemented in C and open-source. Redis is one of the fastest databases because all data is saved in-memory. Another advantage of Redis is the opportunity of replication. The Database uses a Master-Slave technology so it is possible to replicate asynchronously for the purpose of scalability or data redundancy. A big disadvantage of Redis is security. Because of the asynchronous storage Redis is not safe in the point of server failure. Another problem in this topic is the fact how Redis has implemented security. Redis is not designed for security, the user should make the whole system secure.

To recap briefly Redis can fit in diverse scenarios and is very flexible. But it also has big disadvantages, so the user should check thoroughly if Redis is the best solution.

4. Riak

Riak is available in two versions. There is Riak TS for time series data and Riak KV. The chapter about Riak concentrates on Riak KV (further "Riak").

4.1. Introduction into Riak

The following chapter provides an introduction into Riak and its main features.

4.1.1. General Information

Riak is a distributed key-value NoSQL database designed for high availability use cases. As long as the client can reach one Riak node the data is available. The reason is that data is saved across multiple servers. How the clustering works will be shown in the next chapter. (Basho, 06.04.2017)
Riak is available for different operating systems, e.g. Ubuntu, CentOS or Mac OS X but not for Windows. The installation is straight-forward because you just have to download a package from the official website and install the package. (Basho, 06.04.2017)
As data is saved across multiple servers even hardware or network failures can be handled by Riak. If you need more space for your data new servers can be added easily. By adding new servers the scalability is nearly linear which is very impressive. (Basho, 06.04.2017)
Data is saved in buckets. A bucket in Riak can be compared with a table in a SQL-database. (Basho, 06.04.2017)
Now if you have a look on the CAP-Theorem one can say that Riak definitely concentrates on "A" and "P" - Availability and Partition Tolerance. If your system needs high availability and you can not accept downtime Riak is probably the best solution. Another feature of Riak is its latency: since the CRUD-operations do not involve complex joins the requests are serviced promptly. (Basho, 06.04.2017)
On the other hand if your system needs a high consistency of the data Riak is not the right choice. (Basho, 06.04.2017)

4.1.2. Riak Clustering

The high availability of Riak can only be achieved by the Riak clustering. The official website of Riak recommends that there should be at least five nodes in one cluster. A node is a server in production environment - during the development of the software there can be more than one node on a server. (Basho, 06.04.2017)
All nodes have the same responsibility, this means there is no kind of master-node which has special tasks. (Basho, 06.04.2017)

The clustering is visible in the logo of Riak as well. There is one node and three lines to other nodes which symbolize the replication of data:

Figure 4.1.: Riak Logo (Basho, 06.04.2017)

Automatic re-distribution of data

When new servers are added or when machines are removed Riak automatically re-distributes the data with no downtime. Data is spread in the cluster until each node owns the same amount of data. This is why developers do not need to care about where the data is saved. Riak uses consistent hashing to distribute data evenly across the nodes in a cluster. Consistent hashing limits the reshuffling of keys when a hash-table structure is rebalanced. (Basho, 06.04.2017)

Intelligent Replication

Even if nodes fail the user should be able to read and write data. The replication scheme ensures the availability by setting a replication variable, that specifies the number of nodes on which a value will be replicated. The default number is three which means that each object is replicated three times. If Riak can access one node where the object is replicated it is available for the client. (Basho, 06.04.2017)
The following picture describes the replication:

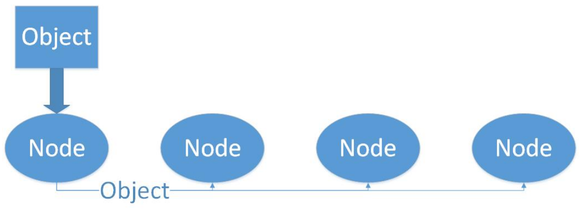
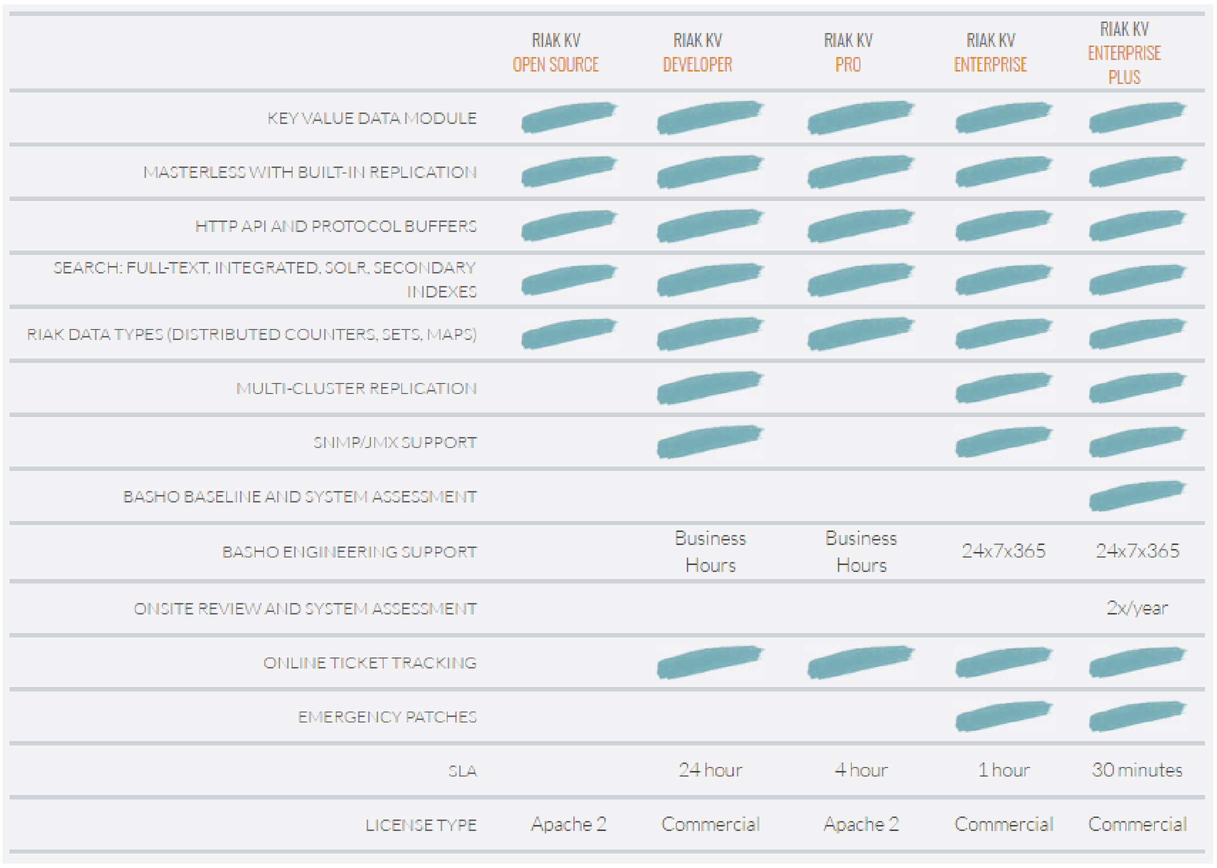


Figure 4.2.: Riak Clustering

4.2. Open Source vs. Enterprise

Basho provides five different versions of the Riak KV database letting the customers always find the perfect configuration for their purpose. The variants are named *Open Source*, *Developer*, *Pro*, *Enterprise* and *Enterprise Plus*. Every configuration has a different composition of features. Figure 4.3 shows a short overview of the versions and the related features.



	RIAK KV OPEN SOURCE	RIAK KV DEVELOPER	RIAK KV PRO	RIAK KV ENTERPRISE	RIAK KV ENTERPRISE PLUS
KEY VALUE DATA MODULE	✓	✓	✓	✓	✓
MASTERLESS WITH BUILT-IN REPLICATION	✓	✓	✓	✓	✓
HTTP API AND PROTOCOL BUFFERS	✓	✓	✓	✓	✓
SEARCH: FULL-TEXT, INTEGRATED, SOLR, SECONDARY INDEXES	✓	✓	✓	✓	✓
RIAK DATA TYPES (DISTRIBUTED COUNTERS, SETS, MAPS)	✓	✓	✓	✓	✓
MULTI-CLUSTER REPLICATION		✓		✓	✓
SNMP/JMX SUPPORT		✓		✓	✓
BASHO BASELINE AND SYSTEM ASSESSMENT					✓
BASHO ENGINEERING SUPPORT		Business Hours	Business Hours	24x7x365	24x7x365
ONSITE REVIEW AND SYSTEM ASSESSMENT					2x/year
ONLINE TICKET TRACKING		✓	✓	✓	✓
EMERGENCY PATCHES				✓	✓
SLA		24 hour	4 hour	1 hour	30 minutes
LICENSE TYPE	Apache 2	Commercial	Apache 2	Commercial	Commercial

Figure 4.3.: Overview of different versions of Riak KV (Basho, 01.04.2017)

In the following, every feature is individually viewed and explained in detail.

Key Value Data Module

Of course every configuration uses the key value data module developed by Basho.

Masterless with Built-In Replication

All configurations are masterless with an integrated replication. This means that data is replicated automatically on multiple nodes so that the application remains available for both read and write operations. There is no single master and no single point of failure. This is the way the database achieves high availability. (Basho, 01.04.2017)

HTTP API and Protocol Buffers

Every version works with a simple HTTP API and Protocol Buffers. Protocol Buffers is a method of serializing structured data and is especially useful for storing data. It is developed by Google. (Google Developers, 06.04.2017)

Search

All implementations of the Riak KV support an integrated fulltext search and Apache Solr. Apache Solr is a popular open source enterprise search platform built on Apache Lucene. This means with Apache Solr the user can search the whole database at once. (The Apache Software Foundation, 06.04.2017)

Riak Data Types

Certainly all versions use the Riak data types which are *Flags*, *Registers*, *Counters*, *Sets* and *Maps*. Flags are similar the same as Boolean, except the values are called *enable* and *disable*. Registers are essentially named binaries like Strings. Flags and Registers are no bucket-level Riak data types. They cannot be used on their own and have to be embedded in Maps. Counters keep track of increments or decrements. Sets are collections of unique binary values. Maps enable the creation of complex, custom data types because all other data types could be embedded. (Basho, 06.04.2017)

Multi-Cluster Replication

This feature is only available for the commercial versions of Riak. The data clusters are replicated automatically in several data centers of the customer. If one cluster fails another one provides the necessary data. (Basho, 01.04.2017)

SNMP / JMX Support

SNMP means *Simple Network Management Protocol* and is a protocol for collecting and organizing information about managed devices on networks. (L8 ManeValidus, 06.04.2017) JMX stands for *Java Management Extensions* and is a Java technology that provides tools for managing and monitoring applications, devices and system objects. (Bryan ssm, 06.04.2017) Customers that use a commercial configuration of Riak are able to implement these extensions and monitor their database.

Basho Baseline and System Assessment

This feature is obtainable just for customers of the Enterprise Plus package. An engineer of the Basho team reviews the whole configuration of the database via remote access before Riak is deployed. (Basho, 01.04.2017)

Basho Engineering Support

The Basho Engineering Support is a simple support hotline. Since the Basho team only provides payed support, this offer is not available for the open source variant. The user has to have an account to contact the support team. (Basho, 01.04.2017)

Onsite Review and System Assessment

That system assessment is nearly the same as the previous one. The only difference is that the assessment is not done via remote access, but a team of engineers come to the customers location and review the system onsite. (Basho, 01.04.2017)

Online Ticket Tracking

The Online Ticket Tracking is a functionality of the account which is needed if you do not use the open source version. In combination with the ability to contact the support team, the user can see the current state of his support ticket. (Basho, 01.04.2017)

Emergency Patches

If the customers of the Enterprise versions face a problem with their database, the Basho team provides a patch as soon as possible. (Basho, 01.04.2017)

Service Level Agreements

The Service Level Agreements are between 24 hours and 30 minutes response time after a problem report was sent by the customer. Afterwards the engineering team provides a solution for the problem as soon as possible. (Basho, 01.04.2017)

License Type

The last point of this comparison is the license type. Riak KV Open Source and Riak KV Pro are available under the open source license Apache 2. The other three configurations use a commercial license. (Basho, 01.04.2017)

4.3. Advantages

Since Riak is a distributed database there are different advantages making Riak special. Riak focuses on high availability, easy scalability and data safety. This is achieved through the distribution of the database across several nodes. (Basho, 01.04.2017) The main advantages of the distributed approach are:

- Installation
- Scalability
- Availability
- Interaction
- Error management

In the following subsections this concepts will be described in more detail.

4.3.1. Installation

The installation of Riak is easy and straight forward. Riak is available for various Linux distributions and Mac OS X. There is an Dabian package for Ubuntu, delivered through bashos web page (HTTP://docs.Bashocom/riak/kv/2.2.2/downloads/). This package can be easily installed with Ubuntus package manager. After the setup you are ready to go. With the command *"riak start"* a Riak cluster with one node and the standard settings is started. (Basho, 01.04.17b)

4.3.2. Scalability

Riak uses a distributed cluster approach built up of several nodes which makes it highly scalable. If there is a need for higher performance or stability there could be easily added several nodes to the cluster. This can be done even when the database is running. Before a node can be added to a cluster it needs to be started with the *"riak start"* command. After the node is running it can be added to a cluster with the *"riak cluster join"* command. (Basho, 01.04.17a)

4.3.3. Availability

A special feature of Riak is its high availability Riak uses a masterless system resulting in no single point of failure. If all nodes fail except one this last node will take all of the responsibilities of the other nodes. The cluster gets very slow, but it stays available and responsive.

4.3.4. Interaction

Another unique selling point of Riak is its native HTTP 1.1 API. This API is designed as RESTful Web service. Create, read, update and delete (CRUD) actions can be performed over the corresponding HTTP methods. This makes Riak a very flexible and handy database. Besides that Riak guarantees client support for common programming languages like Java, Ruby, Python, C#, Node.js, PHP, Erlang and Go.

4.3.5. Error management

If multiple clients can write concurrently and potentially to the same key it is very likely that errors could happen. Therefore Riak has to use an error management. There is an logical approach called vector clock which abstracts the states of a data set on an analogous clock to track the history of updates to a value. If the data is corrupted through conflict writes, it can be restored through this mechanism.

4.4. Disadvantages

The distributed approach of Riak a very beneficial and useful as the previous section shows. But besides this advantages there are some downsides as well. Several nodes and the masterless concept lead to various problems like:

- Inconsistency
- Vector-clocks
- No rollbacks

In the next subsections we will dig a little bit deeper in those problems. (FH Köln, 01.04.17)

4.4.1. Inconsistency

The main problem of highly-available, clustered systems like Riak is the validity of data sets. Due to the fact that there are no ACID transactions data sets can get inconsistent. There is no mechanism guaranteeing the transfer of a consistent state into another. This results in conflicting responses and anomalies which have to be handled.

4.4.2. Vector-clocks

Although the error management concept of Riak is very reasoned the vector-clock system is leading to some difficulties. It is possible that Riak creates different values for an object on various nodes. These values are called siblings. This could lead to inconsistency and conflicts.

4.4.3. No rollbacks

Despite Riaks sophisticated error management system there is no way to completely restore a dataset. This is due to a missing rollback and commit mechanism. This means if you have done any change to the data it is irreversible.

4.5. Use Cases

As you may have noticed, Riak is a quite special database solution. There are several use cases Riak is perfectly suited for. The first one described in here is session data. This means the usage of Riak for storing users and sessions into a Riak database. This data is usually used to save data about the applications connection with the user. Therefore it is very important to have this data highly available. Another common use case for Riak are chat and messenger applications. Traditional rational databases are not suited for the heterogeneous data messaging applications bring along. Therefore key value databases like Riak are more efficient way to save those data. Furthermore the highly-available, low-latency data architecture of Riak provides a good base for messaging apps which are always available. The next point is business continuity. This means applications which should not have a downtime of only several minutes. With Riak you can scale and maintain such applications even if the database is running. This is why Riak fits perfectly well to this use case. The last point described in this section is the usage of Riak for saving content and documents. This documents are highly unstructured because there are a huge amount of different data like pdf files, log files, emails, chat history, books, articles and videos. With Riak all of this data can be saved in a proper manner. There is no overload as in a rational database. (Basho, 01.04.17c)

To conclude with, Riak is used by several companies in the gaming, retail, telecommunications and transportation area as high available and scalable database. Well known companies which use Riak are Uber, Rovio, Best Buy, Xing and Symantec. (Basho, 01.04.17a)

4.6. Conclusion

All things considered one can say that the feature "CRUD"-Operations via REST is very useful but in newer versions **C**ross **O**rigin **R**esource **S**haring should be available since you can not send HTTP requests directly from an front-end to the database.
Furthermore Riak is a **distributed**, **scalable** and **fault-tolerant** NoSQL database. Use cases are mostly applications where high availability of data is the most important point. Another use case are applications with fast growing data as you can add new servers/nodes to your cluster and the data is replicated automatically. (Basho, 06.04.2017)
As already described in the chapter "Use Cases" Riak is especially useful for session data, documents, chat applications and business continuity as all of the use cases need a high availability of the data. (Basho, 06.04.2017)
You should not use Riak if you expect the database to be always consistent since consistency is not possible. This is due to the CAP-Theorem where Riak concentrates on **A**vailability and **P**artition Tolerance.

Part IV.

Documentbased DB

5. Introduction to document based databases

A possibility to realise NoSQL databases are document based databases. The structure of document based databases is like the structure of key-value databases. The big difference are the values which can be reached of the keys. At document based databases the value is a document. The document are collections of data and they exist in different structures. For example the data could be stored in XML or as a JSON object. But not every document is stored as a real document because a document is only a collection of data in the context of document based databases.
This formats have different possibilities to store the data but it is possible to store more then only one value in one document. An other part about document based databases is the structure of the documents. Every document in a database can have another structure.

So it is possible to have a document which looks like the following example:

```
{"id":1,"age":25,"street":"examplestreet"}
```

Listing 5.1: Example

And another document which looks like the following example:

```
{"id":2,"user":"abc","country":"Germany"}
```

Listing 5.2: Example of an other document

But both can be stored in the same database also if they have not the same structure. The possibility of using JSON or XML as format gives this database a advantage when using it with some web applications. At web applications data are often stored in this formats and so it is a good possibility to save the data the same way in the database (FH Koeln, 2013; Cocomore, 2017; Brück, 2015).

6. CouchDB

6.1. Introduction

This ebook was created in the context of a database Implementation lecture at the Cooperative University of Baden-Württemberg. This ebook should explain the base functionality of the document-oriented NoSQL database CouchDB. CouchDB's initial release was in year 2005. Nine years later CouchDB was released in stable version 2.0 (Apache Software Foundation, n.d.). CouchDB is developed in the programming language "Erlang" which was designed by Joe Armstrong, Robert Virding and Mike Williams (Erlang - Wikipedia, 22.03.2017).
CouchDB is licensed with the Apache License 2.0. The initial idea of CouchDB was to developed a database which is easy to manage in the most common functions. So this was the reason why the developer creates a HTTP-based REST API (Anderson et al., 2010) document storage model with a powerful query engine."(Anderson et al., 2010) Based on this fact it is not possible to create SQL queries for data retrieval (Scheliga, 2010). CouchDB stores the data in JSON-Format and the query language is Javascript (Marcel Wolf, 2016).

6.2. Document Structure

Documents are CouchDB's central data structure (Tutorialspoint, n.d.; Anderson, Jan Lehnardt, Noah Slater, 2010). Based on the fact that the CouchDB stores the content of the database not in tables another way is required. The data is stored in a specific form of documents.

6.2.1. Document Characteristics

A CouchDB server hosts named databases, which store documents (Apache, 2014). As explained above, the documents are the place where the database content is stored. To access the data a way to retrieve the information from the documents is required. That is the reason why each document in CouchDB has a unique identifier ID (Brown, 2012; Tutorialspoint, n.d.). The user can choose a specific ID that should be in the form of a string and based on the fact that we want to avoid collisions with twice used ID generally, the so-called Universally Unique Identifier (UUID) is used. In this way, the creation of duplicate IDs is prevented (Tutorialspoint, n.d.).
In addition CouchDB provides a RESTful HTTP API for database documents that can use these IDs to read, change and update documents (Apache, 2014). In the official apache docs, documents are described as the "primary unit of data in CouchDB" and they state out they can "consist of any number of fields and attachments" (Apache, 2014). Furthermore do they include metadata values of varying types (Apache, 2014). For example, text, number, boolean or lists (Apache, 2014).

6.2.2. Design Documents

Design documents are a special type of CouchDB document and beside the core document storage it is probably the most important component of the CouchDB database (Anderson, Jan Lehnardt, Noah Slater, 2010; Brown, 2012). These documents contain the application code or in other words, they contain the database logic about the document (Anderson, Jan Lehnardt, Noah Slater, 2010; Brown, 2012).
"Think of the design document as the glue that turns each of your documents into the format or structure that you need for your application." (Brown, 2012) This quote shows the importance of design documents. To understand the structure of design documents it is necessary to understand what design documents provide.
The following paragraphs explain the purpose of:

- Views
- Shows
- Lists
- Validations
- Update handlers

- Filters

Views

Functions that create lists of documents data are called views. Therefore, they take and process the information inside documents to create lists, that are even search-able. The process to create views is incredibly simple and very powerful because in fact, the user does not need to know the document ID (Brown, 2012). In other words, views are an easy way to organise and group documents in a way that help to understand the meaning of the documents content (Anderson, Jan Lehnardt, Noah Slater, 2010). The purpose of the view is also described as the collection of documents (Brown, 2012).

Shows

To convert a document to another format, a show function is needed. Usually, the most useful format is HTML, although the user can select any format, including JSON or XML if these formats suit their application better. One reason to use a format changing function is that in that way it is possible to simplify the document's JSON content to a reduced format (Brown, 2012).

Lists

A list is to a view as a show is to a single document. The list function is used to format the view as an HTML table, or a page of information or as XML of your document collection. In this way, a list transforms the entire content of one view into another format (Brown, 2012). CouchDB list functions allow you to create the output of views in any format (Anderson, Jan Lehnardt, Noah Slater, 2010). Here's an example (Anderson, Jan Lehnardt, Noah Slater, 2010) design document that contains one list function:

```
{
  "_id" : "_design/foo",
  "_rev" : "1-67at7bg",
  "lists" : {
    "zoom" : "function() { return 'zoom!' }",
  }
}
```

Listing 6.1: Example List Function (Anderson, Jan Lehnardt, Noah Slater, 2010)

Document Validation

To maintain consistency in the CouchDB we need so-called validation functions. "Often in document-based software, the client application edits and manipulates the data, saving it back (Anderson, Jan Lehnardt, Noah Slater, 2010)." Right after the saving of a new document into CouchDB the validation function is called. This function can either check or even reformat the incoming document to meet your requirements and standards for different documents (Brown, 2012). With Validation functions, the user does not have to worry about data causing errors in the DB (Anderson, Jan Lehnardt, Noah Slater, 2010).

Update Handlers

If the user wants to execute an action on a specific document after updating it he or she have to use update handlers. Unlike document validation are these handlers explicitly called. As an example can these handlers be used to increment values in a document or add and update timestamps (Brown, 2012).

Filters

The example of storing information about a CD and DVD collection explains the usage of filters very well. If the user stores information about a CD and DVD collection in a single database, he or she may wants to exchange only the CD records with another database (Brown, 2012). In some use cases, it might be useful to filter the content of the database. These use cases could be, for example, exchanging information between CouchDB databases or using replication (Brown, 2012). If filter functions are called, it probes the list of supplied documents from the replication and then either returns the document or null (Brown, 2012).

6.2.3. JSON Document Format

CouchDB uses JavaScript Object Notation (JSON) for data storage (Anderson et al., 2010). The author Tim Juravich describe JSON as the “strange markup of the document” (Juravich, 2012). “JSON is a lightweight data-interchange format based on JavaScript syntax and is extremely portable (Juravich, 2012).”

```
{
    "language": "javascript",
    "views": {
        "all": {
            "map": "function(doc) { emit(doc.title, doc) }",
        },
        "by\_title": {
            "map": "function(doc) { if (doc.title != null) emit(doc.
                title, doc) }",
        },
        "by\_keyword": {
            "map": "function(doc) { for(i=0;i<doc.keywords.lenghth();i
                ++) { emit(doc.keywords[i], doc); } }",
        },
    },
    "shows": {
        "recipe": "function(doc, req) { return '<h1>' + doc.title + '</
            h1>' }"
}
```

Listing 6.2: Example JSON Document based on (Brown, 2012)

The example in figure 1.2 provides a simple design document that defines three views and one single show (Brown, 2012). The syntax of JSON should be pretty self-explanatory. Curly braces wrap objects. Every Object is key/value lists and the keys are strings, which are wrapped in double quotes. A value can be a string, an integer, an object, or an array (Anderson, Jan Lehnardt, Noah Slater, 2010). Keys and values are separated by a colon and multiple keys and values by comma (Anderson, Jan Lehnardt, Noah Slater, 2010). In other words, the general structure is based around key/value pairs and lists of things (Anderson et al., 2010).

6.3. REST http API

To control and get access to a CouchDB instance it is possible to use a REST API. This API is the first selection of managing the database but it is also possible to use Fauxton which is an administration interface for CouchDB. This chapter will introduce the REST API and will give an overview about the main functionalities. First of all some basics will be explained after that the sections of the API will be introduced. The different sections would be: The server part, the database part, the document part and the replication part.

6.3.1. Basics

The REST API is used over standard http or https methods. The CouchDB API doesn't support all possible HTTP requests methods for example the "PATCH" method isn't supported. Possible methods are: GET, HEAD, POST, PUT, DELETE and COPY. For example the GET method is used to get a specific item of the database and the PUT method allows to create a new object in the database. The root URI to send a request would be `http://www.domain.tld:portnumber/` at the default configuration. But also many other URIs with the root URI at the beginning are existing for every single operation (*CouchDBRestRFCPatch*, n.d.; Apache Software Foundation, 2017a).
To get or send the needed data a standard data structure has to be defined. The CouchDB REST API uses the JavaScript Object Notation (JSON) to structure the data. The motivation for JSON is "because it is the simplest and easiest solution for working with data within a web browser, as JSON structures can be evaluated and used as JavaScript objects within the web browser environment. JSON also integrates with the server-side JavaScript used within CouchDB.“ (Apache Software Foundation, 2017b)
For this reason it is also recommend to mostly set the header of the http request at "accept" to "application/json" instead something special is needed. One additional thing to know is the fact that the header "Content-Type" should be "text/plain" even if the information is saved at the JSON format. If the json from the user was incorrect for example if it was not formatted correctly the response of the API would be the HTTP Status Code 500. In addition to this Status Code the API also uses other standard HTTP Status Codes like 200, 201, 400, 404 or others. But not all codes are supported but the meaning of all used codes is documented(Apache Software Foundation, 2017a).

6.3.2. The Server Part

The server part of the REST API starts at the root URI `http://www.domain.tld:portnumber/`. For example a get request to this URI gives a response with information about the status of the database server. Generally the server part has different URIs which can be used to configure the server or to get specific information. For example it is possible to call `http://www.domain.tld:portnumber/_all_dbs` which returns all created databases at the server. It would be look like this json string: [“testdb","testdb1","testdb2"]. If no database exists it would give an empty JSON array: []. Finally the server part concludes all necessary information about the server and also gives some options which are helpful for the administration part of the database server (Apache Software Foundation, 2017c).

6.3.3. The Database Part

The database part of the REST API concludes all operations which can be used for one database. It starts with creating a database and goes over to add entries and configure the security options of a database. An example to create a database would be following PUT request at the URI: `http://localhost:5984/<databasename>/_security`.
Another example would be to configure the security settings of the database “testdb”: First the current settings would be requested over GET `http://localhost:5984/testdb/_security` with the response which can be seen following:

```
{
    {
        "admins":{"names":["admin"],
        "roles":["adminstrators"]},

        "members":{"names":["user"],
        "roles":["tester"]},
        "ok":true
    }
}
```

Listing 6.3: Example security response

After that a PUT request to the same URI with the following body is send:

```
{
    {
        "admins":{"names":["admin","admin2"],
        "roles":["adminstrators"]},
        "members":{"names":["user"],
        "roles":["tester"]},
        "ok":true
    }
}
```

Listing 6.4: Put request body

The request returns only a short answer from the REST API:

"ok":true

A second GET request to the same URI returns the string which can be seen following:

```
{
    {
        "admins":{"names":["admin","admin2"],
        "roles":["adminstrators"]},
        "members":{"names":["user"],
        "roles":["tester"]},
        "ok":true
    }
}
```

Listing 6.5: Get response

All URIs which belongs to the database part of the API are available under `http://localhost:5984/<databasename>/<command>`. So only the right database name and the right command has to be appended to the root URI (Apache Software Foundation, 2017d).

6.3.4. The Document Part

The document structure is a central element of the database and it has also an own part in the API. This part manages the CRUD (create, read, update, delete) operations to the documents which belongs to one database. The design of the REST API gives the possibility to manage more then one database so it is necessary to have the possibility to select the database first and then select the document.
In the chapter The Database Part the database part is presented with the `http://`

localhost:5984/<databasename> path. This path is going to be expanded to http://localhost:5984/<databasename>/<documentid>. This basic path could be used for different options which can be used to the document. This options are selected about the HTTP request methods. For example PUT is used to create a new document with information in JSON string which is stored in the request body. Other available methods are: HEAD, GET, DELETE and COPY.
All necessary options to the documents are realized with this one URI without the part of the attachments. CouchDB has the possibility to add attachments to a document. The management of this attachments over the REST API is realised on the same way like the management of the documents. The only difference is the missing HTTP COPY method and a different path. For the attachments the path http://localhost:5984/<databasename>/<documentid>/<attachmentname> is used. About this path all operations can be executed (Apache Software Foundation, 2017e).

6.3.5. The Replication Part

CouchDB also supports the replication between two databases. This option is managed by some settings which can be managed by the URI http://localhost:5984/_replicate. With this URI is only one single replication triggered it is not like a background job which is executed every X minutes. The interesting part of this URI is the fact that is not a correct RESTful API method. The RESTful architecture has one principle which means that every request gives a representation of an object. The call for the replication is only the call of a procedure which is located on the server or on another remote location (Apache Software Foundation, 2017f).

6.3.6. Conclusion

Finally the REST API of CouchDB gives the possibility to do all operations which are possible about an graphic user interface also about the REST API and the HTTP protocol. But also it should be noticed that not all methods are complete RESTful and they are mixed with methods which comply with the REST architecture. The documentation argue with the fact that they want to use the optimal technique for all events and so they decide to mix both ways together to give the user a better experience.

6.4. Security

Storing sensitive data today requires a high standard of IT security. If databases store personal data or business secrets it is necessary to protect these data against loss, destruction and theft. For this reason the database architects should be informed about the actual security mechanism, authentication mechanisms and about existing security gaps. This chapter describe the base authentication mechanisms, security issues and the administration of users.

6.4.1. User Authentication

After the first installation of CouchDB every user disposes automatically over privileges of an administrator. Every user can create database or change system properties. This very big security gap is also known as "Admin Party". Since version 0.11 it is possible to differ between three user roles
(Apache Software Foundation, 2013b):

1. Server administrator
   The user role of a server administrator disposes of all privileges. They have reading access and writing access to all databases stored on the CouchDB server and can change all server settings (Anderson et al., 2010).

2. Database administrator

   The database administrator disposes of reading access and writing access of one specific database. This user role have the privileges to create, edit, delete documents but "they can not create and neither delete [other] databases." (Apache Software Foundation, 2013b) Additional they can create new database members and database admins for these specific database.

3. Database members
   The database members have reading access to all documents but they have only writing access to 'normal documents'. Especially they can not change design documents.

Creating server administrator
There are two technical capabilities to create a new server administrator. The first possibility is to change the *local.ini* file
(Apache Software Foundation, 2013a):

```
[admins]
username = password
```

Listing 6.6: Example Create new Server Administrator with local.in File (Apache Software Foundation, 2013a)

Currently every admin has the possibility to read these plain-text password. This situation is a very big security gab. Only after restart CouchDB the password will be encoded with a hash. This hash will be create within three steps (Anderson et al., 2010):

1. creates a new 128-bit [Universally Unique Identifier] (random numbers with low collision probability) = **salt** (Anderson et al., 2010)
2. "Creates a sha1 hash of the concatenation of the bytes of the plain-text password and the salt (sha1(password + textbfsalt))"
(Anderson et al., 2010)
3. "Prefixes the result with -hashed- and appends textbfsalt"
(Anderson et al., 2010)

After restart CouchDB the *local.ini* looks like:

```
[admins]
username = -hashed-207b1b4f8434dc60429672c0c2ba3aae61568d6c,96406178
    a0718239acb72cb4e8f2e66e
```

Listing 6.7: Example Create new Server Administrator with local.ini File (Apache Software Foundation, 2013a)

But it also possible to create a server administrator with the provided API (for further information please see chapter 1.4.3 "The Database Part"):
(Anderson et al., 2010)

```
curl -X PUT \$HOST/\_config/admins/username -d '"password"'
```

Listing 6.8: Example Create new Server Administrator with REST API (Anderson et al., 2010)

Until now it is only possible to create a new databases after you entered the administrator password.

Creating database administrator and database members

To create a database administrator or a simple database member it is necessary to change the security object of the selected database. These object is located under */db_name/ _security*

(Apache Software Foundation, 2013b). The data in these object is in JSON format (Apache Software Foundation, 2013b):

```
{
  "admins" : {
     "names" : ["ines", "tobi"],
     "roles" : ["boss"]
   },
   "members" : {
     "names" : ["leon"],
     "roles" : ["producer", "consumer"]
   }
}
```

Listing 6.9: Example Create new Database Members and Database Administrator (Apache Software Foundation, 2013b)

If a database have no database users every user can change regular documents! CouchDB stores data of the single users in an user authentication database named (_\users). Every user have a separate document. In this document it is possible to save the properties of every user for example id, name or user roles.The property "type" can only substituted with the key "user". An example user definition in such a document is shown in the following graphic: (Apache Software Foundation, 2013b):

```
{
 "\_id":"org.couchdb.user:ines",
 "type":"user",
 "name":"ines",
 "roles":["guest"],
 "password\_sha":"fe95df1ca59a9b567bdca5cbaf8412abd6e06121",
 "salt":"4e170ffeb6f34daecfd814dfb4001a73"
}
```

Listing 6.10: Example User Definition (Apache Software Foundation, 2013b)

6.4.2. Cookie Authentication

The standard user authentication with user name and password shows a increasingly security gap. Therefore, CouchDB offers the possibility of user authentication with Cookies. By the use of Cookies users can be authenticate without filling user-related data in permanent opened password dialogues. CouchDB offers the possibility to use Proxy or implement particular authentication modules to sync or authenticate user data with existing LDAP systems or Active Directory (Anderson et al., 2010).

6.4.3. Validation Function

To be able to guarantee the consistency of the data in a database system, it is helpful to provide rules directives. Afterwards every document must fulfill these requirements. For example, it is possible to define directives which ensures that the field "name" may not bee empty. If this field will be empty an error message will be shown. (Scheliga, 2010).

```
{
    \_id: "\_design/designdoc",
    validate\_doc\_update: "function(newDoc, oldDoc, userCtx) {
    if(newDoc.name === undefined) {
    throw {forbidden: 'Document must have a name.'};
    }}"
}
```

Listing 6.11: Example Validation Function (Scheliga, 2010)

The deposited validation directives launch automatically if a new document created or an existing document changed. Every validation directives can include an arbitrarily number of these rules. However every design document can only deposit one validation function (Scheliga, 2010).

6.5. Conclusion

In conclusion, this book explains the background to understand the concept of the document-based database CouchDB. The provided information give a deep inside on the document structure, the REST http API and security aspects the user gets in contact by using CouchDB. In addition to the provides overview, there are some advantages and some disadvantages of using CouchDB that are notable.

On the one hand, big advantages of CouchDB are the scalability and fast read and write operations. More features that can be described as advantages are for example the synchronisation, offline usage and the support for mobile devices. CouchDB has the ability to synchronise between multiple databases and in addition the offline functionality provide the function that data on the device can be manipulated and synced later, when the device get a connection to the main database (Morony Josh, 2017).

Another important advantage is the way the document storage is implemented. Based on the fact that everything is stored as documents and there is no pre-defined schema the way how the datasets are designed is up to the user. This creates a lot of flexibility for data storage, but it also has its downsides and it can be difficult to learn because there is no *100%* correct way to store the data (Morony Josh, 2017). Furthermore, CouchDB provide a simple REST API where all items have a unique URI that gets exposed via HTTP. In addition, the build-in administration interface Futon that is accessible via Web provide a easy way to interact with the database and is with the REST API feature one of the most important advantages.

On the other hand, some drawbacks are worth mentionable. CouchDB guarantees eventual consistency to be able to provide both availability and partition tolerance. In detail, this implies two write and read operations in the same time frame will cause the problem, that the update will not necessarily be see-able (Emin Gun Sirer, 2012). It is likely that some use-cases do not work with this issue. For example, selling concert tickets, would not work with this model (Emin Gun Sirer, 2012).
Remarkable is that only a few big projects are published which use CouchDB. For example, the platform readwrite published that "the Compact Muon Solenoid Experiment (CMS) at CERN (The European Organization for Nuclear Research) will deploy the NoSQL database CouchDB into production this summer"(KLINT FINLEY, 2010). In addition, CouchDB is also used by CloudWork(Bruno Pedro, 2013).
Resting upon the pros and cons it is noticeable that every use-case for the database has to be conceptualised to proof either CouchDB is the right solution or not. Thereby, this ebook has the purpose to contribute strongly to the overall course ebook about NoSQL databases.

7. MongoDB

This chapter guides you through the features of MongoDB. Whereby we go through what is MongoDB in general, Data model and CRUD Principles. Thereafter we check through the Use-Cases of MongoDB and performance and limitations. Additionally, we get a short Insight how MongoDB is used in a Big Data context. After all, we come up with a conclusion.

7.1. What is MongoDB ?

MongoDB is a part of NoSQL family and belongs to the document-oriented databases. Therefore, it doesn't have the concepts of tables, rows and columns. Instead, MongoDB is built on an architecture of collections and documents. Documents contain sets of key-value pairs like JSON and are the basic unit of data in MongoDB. A collection includes a set of documents and offers the same functionality as relational database tables(Banker, 2016)

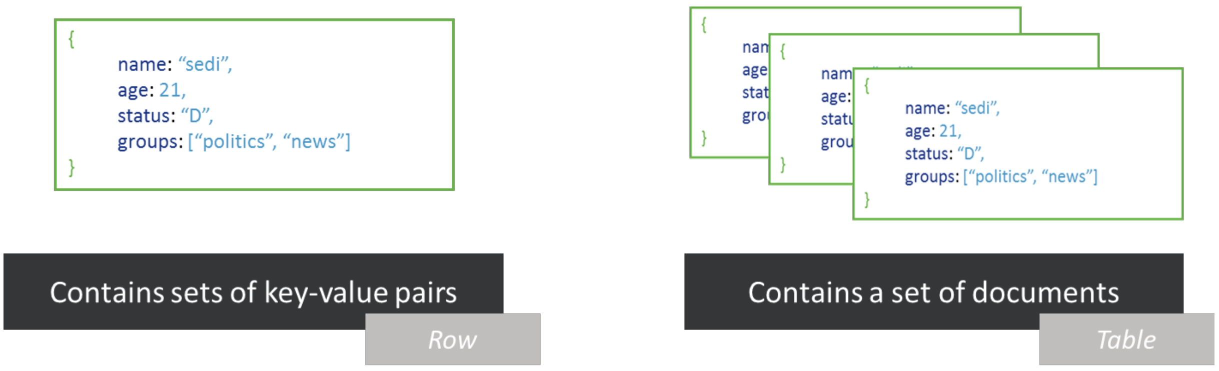

{
name: "sedi",
age: 21,
status: "D",
groups: ["politics", "news"]
}
Contains sets of key-value pairs
Row
{
name: "sedi",
age: 21,
status: "D",
groups: ["politics", "news"]
}
Contains a set of documents
Table

Figure 7.1.: Documents and collections

MongoDB stores data as Binary JSON documents also known as BSON. The documents can have different schemas, which means that the schema can change dynamically as the application evolves. Automatic sharding enables data in a collection to be distributed across multiple systems for horizontal scalability as data volumes increase(Edward & Sabharwal, 2015). Additional, MongoDB doesn't only support Key-Value operations.

It allows complex queries, aggregations and secondary indexes that unlock the value in structured, semi-structured and unstructured data. One of MongoDB's major features is the support for many types of queries like text search, range queries, geospatial queries over to MapReduce queries(MongoDB Inc., 2013b).
MongoDB was created by Dwight Merriman and Eliot Horowitz, who had encountered development and scalability issues with traditional relational database approaches while building Web applications at DoubleClick, an Internet advertising company that is now owned by Google Inc. The database was released to open source in 2009 and is available under the terms of the Free Software Foundation's GNU AGPL Version 3.0 commercial license. However, the database is one of the most popular NoSQL databases and is used by several company like Bosch, Facebook, Expedia and so on(Plugge, Hows, Membrey, & Hawkins, 2015).

7.2. Use Cases: What is MongoDB for?

This section will give you an insight between the features and potential of MongoDB and some problems that is suited to solve.
Beginning with online and mobile Apps. Nowadays Companies want their business on their smartphones or access it from everywhere through the web. In comparison to RDMBS MongoDB addresses the upcoming challenges of these plans. Furthermore, MongoDB promises to make it easier than other alternatives. Requirements for going mobile or online are hard to manage. For example, different types of device like smartphone or wearables are creating new types of unstructured and semi-structured data. Another reason is the number of devices and users. Meanwhile, Response times must keep and provide the same User experience. So, Scalability has now a high priority. MongoDB tries to decrease the degree of difficulty for these Requirements. That is why MongoDB offers a flexible data model and rich query functionality. Therefore, MongoDB can manage any kind of data, no matter how dynamically the data changes. In addition to that, MongoDB's development concentrates on scalability and can handle a lot of Users and data sets(MongoDB Inc., 2013b).
Another Example for a suited use case is a catalog. Mention that almost anybody knows some requirements a Catalog must meet. Deleting, creating and changing items or their features or their attributes – only to mention few - are a standard set of Requirement of a catalog. Behind the scenes, we see a lot of challenges for a RDMS. There will be a lot of changes in the data, like new data and new metadata to your catalogs. We already talked about the untrusted and semi-structured in the first Use case. Again, we got the same problems with a RDBMS. How does MongoDB make it easier for developers? First, it is how the data is structured in MongoDB. With MongoDB's JSON document model

makes it easy to store different assets with different attributes in a single place. It also makes it simple to represent complex, hierarchical relationships. Schemas in MongoDB are self-describing. You can add new products and features and evolve the schema instantly, without taking the database down or impacting performance. Lastly an expressive query language, indexing, including text search and geospatial, and analytics provide flexible access to the data, no matter how the application, business or developer needs to find it(MongoDB Inc., 2013b).
All in all, these are only few examples for use cases and for what MongoDB is. In general MongoDB claims to be suited for high flexible data schemas to provide the ability for data changes, structed, unstructured and semi structured data. Additionally, MongoDB eco system let developer spend less time for the design of models, entities, relationships and tables, and more time on the application(MongoDB Inc., 2013b).

7.3. Datamodel

Before designing a database, it is crucial to analyse the data and the requirements of the application. In contrast to relational databases there is no strongly recommended way to structure your data, like for example a normalized data model to avoid inconsistencies on updates. However, there are well established patterns which help developers to create their data model (Banker, 2016). Later in this chapter, there will bw some examples for common design patterns. But at first, the following paragraphs will concentrate on the data concepts of MongoDB in detail.

7.3.1. Databases and Collections

As mentioned in the introduction, MongoDB is structured in databases, collections and documents. MongoDB provides a mongo shell to communicate with the database. The code snippets provided in this chaptered can be performed on the mongo shell (Inc., 2016).

A Database can hold collections of documents. The creation of a database occurs automatically when the first document is pushed to a collection of this database. Thus, there is no command to create an empty database (Inc., 2016). The following commands show how simply it is to select and create a new database by inserting its first document.

```
use newDatabase
db.newCollection.insertOne( { value: 2 } )
```

Listing 7.1: Create Database

This example also shows the process of creating a new collection. As the database, the collection is created as soon as the first document got inserted. Certainly, there is also a function to explicitly create a new collection with an option to pass parameters affecting the collections behaviour. In the example in Listing 7.2, a collection with limited number of documents is created. All options can be found in the MongoDB documentation (Inc., 2016).

```
db.createCollection( 'newCollection', { max: 1000 } )
```

Listing 7.2: Create Collection

7.3.2. Documents

MongoDB stores documents in the BSON format. This format supports multiple datatypes. Many of them are common in the information technology, such as Double, String or Boolean. A full list would go beyond the scope of this book, but can be found in the BSON specification. In general, there are all types needed for object oriented programming (bsonspec.org, 2017). The chapter Queries and CRUD operations will treat the way BSON types can be used to query through documents. But before that, the concept of documents will be explained.

7.3.3. Embedded Documents and Referencing

The below example in Listing 7.3 shows how a document is structured. The field name _id is reserved to be used as primary key. It has some special characteristics like being unique in the collection and immutable. To make sure this field is unique, it is recommended to use a unique ObjectId. If there is no value submitted with the document, an ObjectId is generated automatically (Inc., 2016).

Other fields can be added as needed. For example, a field containing the date of birth of a person. Or an object containing the first and last name and another object with contact details. This concept of containing objects inside a document is called embedded sub-documents and has its own strength and weaknesses (Inc., 2016). It may be resulting in a performance growth if the application always needs the document with its whole embedded data. But one common pattern for MongoDB says to consider whether embedded data or referencing is more suitable for the given situation (Banker, 2016). By referencing, the embedded data is stored in a separate document with a reference to its related document. This concept is the foundation for creating normalised data models. How this can be

```
var newDocument = {
  _id: <ObjectId>,
  dateOfBirth: new Date('Jan 01, 1990'),
  name: {
        last:  'Doe',
        first: 'John'
    },
  contact: {
    phone: '12345678',
    email: 'john@doe.com'
  }
}
```

Listing 7.3: Embedded Documents

implemented for the document with John Doe can be seen in the code snippet in Listing 8.9.

```
var newPerson = {
  _id: <ObjectId1>,
  dateOfBirth: new Date('Jan 01, 1990'),
}

var newName = {
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
    last: 'Doe',
    first: 'John'
}

var newContact = {
    _id: <ObjectId3>,
    user_id: <ObjectId1>,
    phone: '12345678',
  email: 'john@doe.com'

}
```

Listing 7.4: Referencing Documents

When deciding whether referencing is reasonable, the atomicity of write operations also influences this decision. In MongoDB atomicity is only guaranteed on document level. As no single write operation can affect more documents, a normalised data model cannot be updated with one atomic operation. At this point denormalised data models have benefits over normalised ones. However, another design pattern for MongoDB is to overthink your overall database selection if you need atomic, consistent, isolated and durable operations, also known as ACID principles (Banker, 2016).

7.3.4. Validation

MongoDB comes with a way to validate documents by itself. There are different ways to handle documents which do not match its validation criteria. By default, invalid documents are rejected with an error. A collection, which would check if the new document contains a date of birth, would look like the following in Listing 7.5. Furthermore, it is also possible the check the fields for a certain data type (Inc., 2016).

```
db.createCollection( 'users', {
  validator: {
      $or: [ {
          dateOfBirth: { $exists: true }
      }
        ]
    }
}
```

Listing 7.5: Validation for Collections

7.4. Queries and CRUD operations

MongoDB has several queries and operations to manage its data. This chapter gives a short overview of the most important operations.

7.4.1. Create (Insert)

Before searching for or working with data, the database needs information that can be worked with. How documents can be created was introduced while treating how to create a new database. This command inserts one document into a collection and can be found again in the Listing 7.6. But there are also commands to insert an array of documents at once (Membrey, Hows, & Plugge, 2014).

```
db.newCollection.insertOne( {name: 'John', age: 30} )
db.newCollection.insertMany( [
    {name: 'Max', age: 20},
    {name: 'Marie', age: 25}
] )
```

Listing 7.6: Create (Insert)

7.4.2. Read (Find)

To find data stored in a MongoDB, operations to either find one or multiple documents can be used. This example in Listing 7.7 shows both operations. The difference is that the second operations stops after the first result and it is generally advised if only one result is excepted. Of course, the results can also be filtered by field values as shown in line three, or sorted by values as shown in line four (Membrey et al., 2014).

```
1 db.newCollection.find()
2 db.newCollection.findOne()
3 db.newCollection.find( { 'name.last': 'Doe' } )
4 db.newCollection.find().sort( { 'name.first': 1 } )
```

Listing 7.7: Read (Find)

7.4.3. Update

MongoDB also provides a function to update documents with three input parameters: the search criteria, the new object and options. It is important to understand that any fields that are not part of the new object are also removed from the old object, as it were completely rewritten. The option upsert implies to add any fields that do not exist yet. It is also possible to update multiple options that match the search criteria with one command (Banker, 2016). An example for a basic update operation can be found in Listing 7.8.

```
db.newCollection.update(
    { name: 'John' },
    { age: 21, name: 'John', lastname: 'Doe'  },
    { upsert: true } )
```

Listing 7.8: Update

7.4.4. Delete

The last operation of CRUD examines the deletion of documents, collections or whole databases. The remove operation, as shown in line one of the code snippet in Listing 7.9, removes all documents which match the criteria. The _id field can help to be sure to only delete the right document. It can lead to conflicts if a documents is deleted without considering its references, because references from other documents will not change automatically (Membrey et al., 2014). Line 2 shows how to delete a collection and line 3 shows how to delete an entire database.

```
db.newCollection.remove( { age: 20 } )
db.newCollection.drop()
db.dropDatabase()
```

Listing 7.9: Delete

7.5. Architecture

7.5.1. Core Processes

The MongoDB server package comes up with three main processes:

- mongod
- mongo
- mongos

Mongod is the core database server. Once started, the *mongod* listens by default on port 27017 for requests. It also manages the data format and performs all background operations. For administration the *mondod* provides an HTTP interface, witch can be reached at the localhost on port 28017 (1000 higher than the default port). The data directory *mongod* connects to is by default C:\data\db (or /data/db). This directory definitely has to exist and the default ports have to be free, otherwise the process fails to start (Edward & Sabharwal, 2015).
The *mongo* process is an interactive MongoDB shell. It gives the user the possibility to communicate with a running *mongod* process via a JavaScript like query language. If running on the same host, it automatically connects to the *mongod* process and a preinstalled test database (Edward & Sabharwal, 2015).
The *mongos* process is working like a routing service and is the basis for MongoDBs sharding ability that will be described later (7.5.3). It holds the information about where the requested data is located and forwards the request from an application server to the right destination (Edward & Sabharwal, 2015).
By running a *mongod* process you already have a standalone deployment of MonogDB that can be accessed by a client. But in case of failure there is no redundancy or recovery, that prevents data loss, so its not recommended to use this in a production environment. To avoid this, replication is used to guard against hardware failure or database corruption. It also gives the possibility to perform normally high-impact maintenance with little or no impact (Plugge et al., 2015).

7.5.2. Replication

„MongoDB supports the replication of a database's contents to another server in real time or near real time“(Plugge et al., 2015, p. 285). For that MongoDB provides two different replication methods. The traditional *Master/Slave Replication* and *Replication Sets.*

Master/Slave Replication

In a Master/Slave setup one *mongod* instance acts as a master, the others declared as slaves. All write and read operations are requested to the master and the slaves replicate the data of the master, but can't be requested by a client. If a failure occurs that forces the master to go down, the hole system can't be reached anymore. The data, till the last replication to the slaves, is saved, but can't be accessed until the master comes up again. In MongoDB the master holds a capped collection called *oplog.* The *oplog* is an ordered history of all logical writes, that are executed within a defined time period. The operations stored in the *oplog* in an idempotent way, so they can be performed multiple times without changing the result (Edward & Sabharwal, 2015). That's useful when a slave runs into failure while executing the operations onto its data. In that case the replication process can be simply restarted. If the slaves syncing process last to long or the slave was down for a longer time, the oplog data could be deleted before the slave was able to synchronize (Edward & Sabharwal, 2015). In that case the slave has to start a resync process. To avoid such a situation, the oplog length should be chosen in consideration of the slaves performance.
The configuration of MongoDB with a *Master/Slave Replication* is only recommended for more than 50 nodes (Edward & Sabharwal, 2015). At that point the next described replication method, the *Replica Set*, is reaching its limits, because the communication overhead becomes to big.

Replica sets

In contrast to the *Master/Slave Replication*, in a *Replica Set* no fixed master is defined. Instead the nodes are declared as primary or secondary. Each node in a *Replica Set* can become primary, but there is only one primary at a time, the others are secondaries. All write operations going through the primary, but read operations can also be performed by secondaries (Edward & Sabharwal, 2015). The replication process works just like it does in a *Master/Salve Replaction*, but if a primary goes down a new primary is elected out of the secondaries.

Communication

All nodes in a *Replica Set* communicate with each other. As life sign they are sending a heartbeat signal to each node and getting back status replies of each node. Those replies contain information about the node, such as is he primary or secondary and what type of node he is. Each node can be assigned a certain number of votes and a priority.
This results in a various types of nodes:

- **normal secondaries:** hold a copy of the primaries data, accept read requests and are primary candidates
- **priority-0-members:** secondaries that will never become primary
- **hidden-members:** priority-0-members that can't serve read request, because they are hidden for the client
- **delayed-members:** have a delay in synchronization to prevent human failure
- **arbiters:** don't hold data, they just solve ties in a election process
- **non-voting-members:** normal secondaries, but they can't vote

If the primary recognizes that the heartbeat of a secondary has stopped, he has to check if he still can reach the majority of the set. If it can't he demotes itself to secondary and starts a election process. Also the election process is started if a secondary recognizes the primary is down. All voting nodes now collect the for the election required information form the primary candidates. The election of a primary depends on various parameters. Important is, that the elected node has the most recent data of all nodes. The candidate with the most votes is promoted to primary. When the old primary comes up again, he will be a new secondary (Edward & Sabharwal, 2015; Plugge et al., 2015).

Write Concerns and Read Preferences

MongoDB provides two important configurations to regulate consistency and availability. With *write concerns* it's possible to configure a minimum number of secondaries, that have to replicate the data, before the client gets the success response. Figure 7.2 shows how a write process would run, by configuring a minimum of one secondaries.

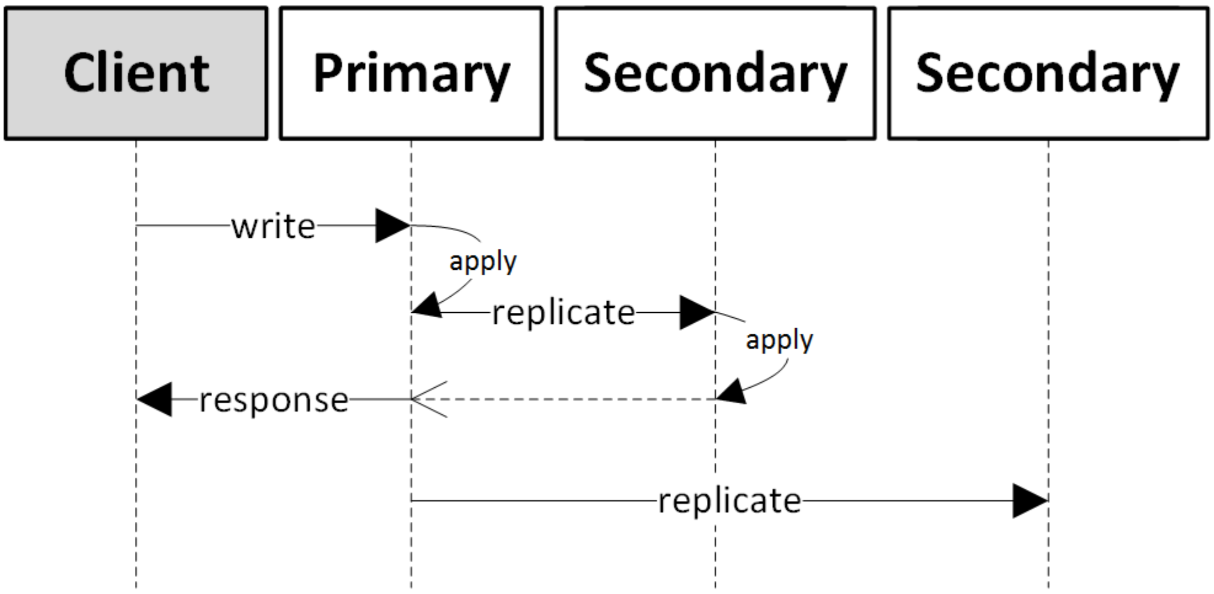


Figure 7.2.: Write process with write concerns

Read preferences allow the administrator to route read operations. They determine from which node a client is allowed to read. MongoDB supports five read preferences:

- **primary:** all read operations are requested to the primary node
- **primaryPreferred:** read operations are requested to secondaries, if the primary is unavailable
- **secondary:** read operations are requested to secondaries
- **secondaryPreferred:** read operation are just requested to primary, if no secondary is available
- **nearest:** read operations are requested to the node with the lowest network latency

(Inc., 2016)

7.5.3. Sharding

If the amount of data exceeds the capacity of a single database server, partitioning is needed to distribute the data on multiple servers. For MongoDB this ability is even more important, because it uses memory mapped file I/O to access its underlining data storage (Plugge et al., 2015). MongoDB uses a horizontal partitioning mechanism called *sharding*. The data collection gets divided and distributed onto multiple servers called shards. Every shard is an independent database managed by multiple *mongod* processes. All the shards are combined to one logical database. The partitioning and routing are managed by the earlier mentioned *mongos* (7.5.1) process. All write and read requests of an application are send to a *mongos* process, that holds the information where the requested data is stored and forwards the requests to the responsible *mongod* processes. The data is distributed based on a configured *shard key* and chunk size. The metadata of a sharded cluster is stored on special config servers, where the *mongos* processes can obtain the routing information (Edward & Sabharwal, 2015; Plugge et al., 2015).

7.5.4. Summary

Figure 7.3 describes one possible deployment architecture, that contains all in this section mentioned artifacts. Clients can connect to a *mongos* process, running on an application server. This process forwards the requests, based on the information stored on the *config servers*, to the right shard. A shard is an replica set, containing several *mongod* processes. The *shard key* in this example is the year. So Shard-1 contains all data from 1999 til 2009 and Shard-2 contains the data from 2009 til now.

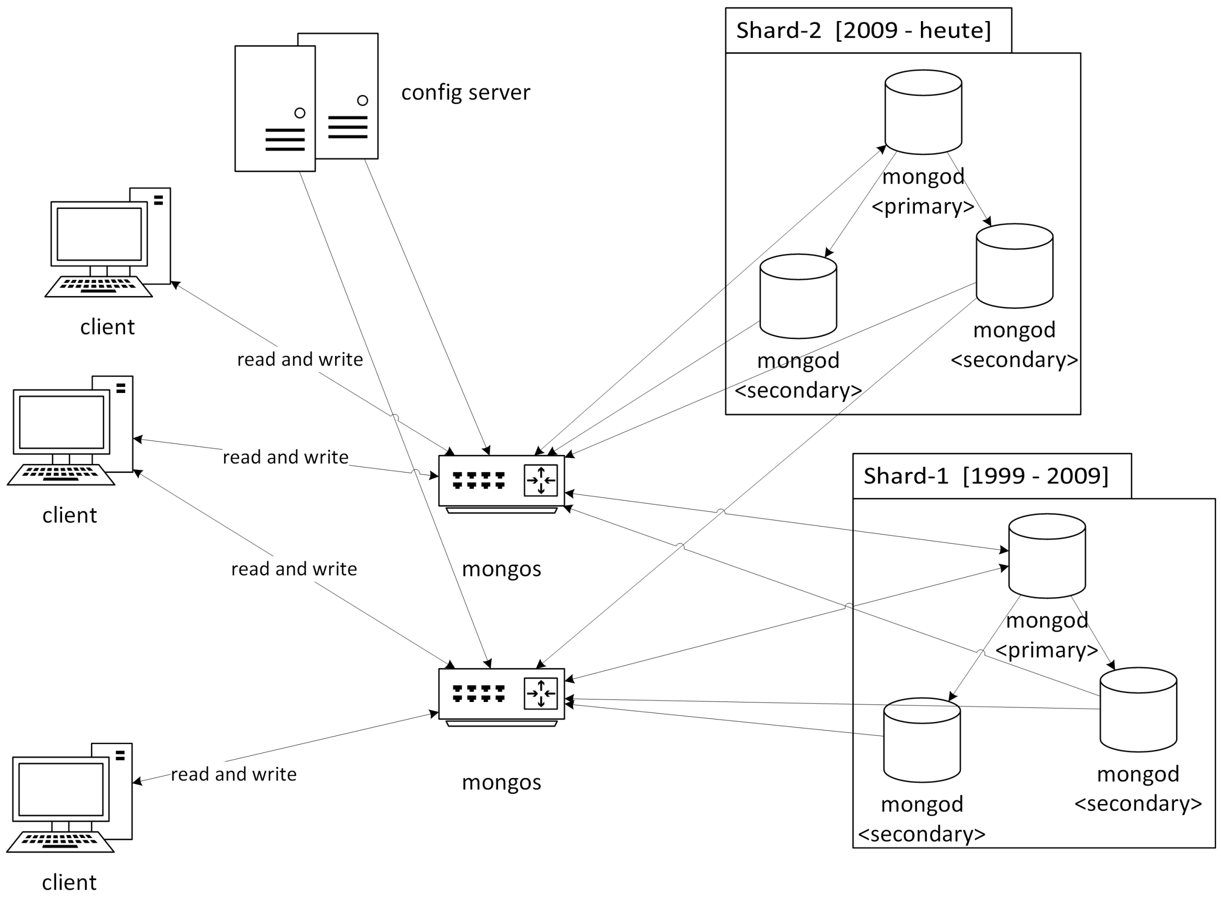


Figure 7.3.: MongoDB Architecture Example

7.6. Performance

7.6.1. Performance measurement introduction

Before a performance measurement or comparison between database systems can be made, it is necessary to define the performance indicators. Depending on the use-case of the measured databases the results can be completely different. For example a real-time system is a lot more dependent on access times, latency and fast updates for concurrent users. A scientific database has to be fast at processing complex queries with joints and preprocessing routines like aggregation (O'Neil, 1997). Of course the benchmark should be implemented to test the performance of a database system in a way that reflects your usage in the future.

For database systems there is a council called the TPC . This organization tests different database systems on physical and virtual machines and scores them by performance. The performance indicators can be seen on their website and there are different benchmarks depending on the use-case of the system (Tpc.org, n.d.).

Another standardized benchmark for database implementations is the Wisconsin Benchmark. This was one of the first standardized benchmarks and was made for relational databases. The test contains of inserts, selections, joints and projections. For further details on this benchmark, see the paper (DeWitt, 1991).

This paper will summarize MonogDB performance compared to other popular NoSQL and SQL databases.

7.6.2. Influencing factors to database performance

All the results provided by this paper are dependent on the underlying hardware used. Depending on the host system for the database application, the performance can vary a lot (Lee, 2009). Examples for big performance factors are available memory, processor speed and the used storage.

For evaluation which database system should be used, it is important to know on what kind of hardware the production system will run. This is due to the fact, that different database systems were developed to meet certain performance goals on different hardware. Therefore some databases scale well with more memory and memory bandwidth since they try to cache a lot of the data in system memory. But there are also databases which try to be very lightweight on memory for low end systems or massive I/O operations (Boncz, 1999).

Nearly all databases are reliant on the used storage. This storage is the only way to keep the data, even when the system is turned off. Even In-memory databases use the persistent storage to save their current data (Wang, 2001). As transactions ultimately have to be written to the storage, this becomes the biggest bottleneck. With traditional HDDs having a high latency when accessing random data, which happens frequently on a database system, the introduction of SSDs eliminated those problems. Benchmarks done on traditional HDD storage can be translated most of the time to SSDs since all databases will perform better but should stay at the same relative performance.

7.6.3. NOSQL compared to SQL databases

It is often one of the first question, when talking about database performance. SQL or NOSQL, which one is faster? Comparing those two types of databases generally, is not really useful. The way how data is stored is completely different and the use-case defines if SQL or NOSQL fits better.

A good example for the difference of these two types is Twitter. They use a NOSQL database for all the tweets. Although Twitter is using MySQL heavily. The reason for this is easy to explain. A tweet contains some sort of content like text or images and a lot of additional information like hashtags, user and topics. Storing that data in a relational, normalized way, it would take a lot of tables and joins to represent a tweet . All hashtags would be referenced over multiple tables. These joins take time and since Twitter is a real time platform, latency should be low (Weil, 2017).

In contrast the NOSQL implementation is way easier. The complete Tweet is stored as a document. The hashtags and links are stored in the same document in a key value manner. Now when you want to read 100 Tweets, in a NOSQL context, this can be done by one simple query. In a relational model, complex joins for each Tweet would be necessary. In such a scenario performance is obviously on the NOSQL side, but only because the use-case is well suited for non-relational models.

7.6.4. MongoDB performance

This paper will be limited to benchmarks of low level functions for databases. These functions are: reads, writes and deletions. Performance is measured between MongoDB and MariaDB. MariaDB is a SQL database and as previously mentioned, general comparisons between NOSQL and SQL are not useful. In this case two implementations of SQL and

NOSQL are compared which is relevant when the use-case can be implemented by both designs without drawbacks. In such a scenario, raw performance is a valid factor. In addition to the benchmark implemented by the author of this section, another one is used for validity and a broader overview. The referenced paper contains additional databases.

The following performance test were performed, using NodeJS. For MongoDB connectivity the official MongoDB driver was used. The same applies for the MariaDB driver . The driver selection can cause huge performance differences. There are several MariaDB drivers for NodeJS and of course other programming languages. Therefore comparisons of database performance should be done, using the same programming language (Mscdex, n.d.). The inserted data contains just an id represented by an integer.

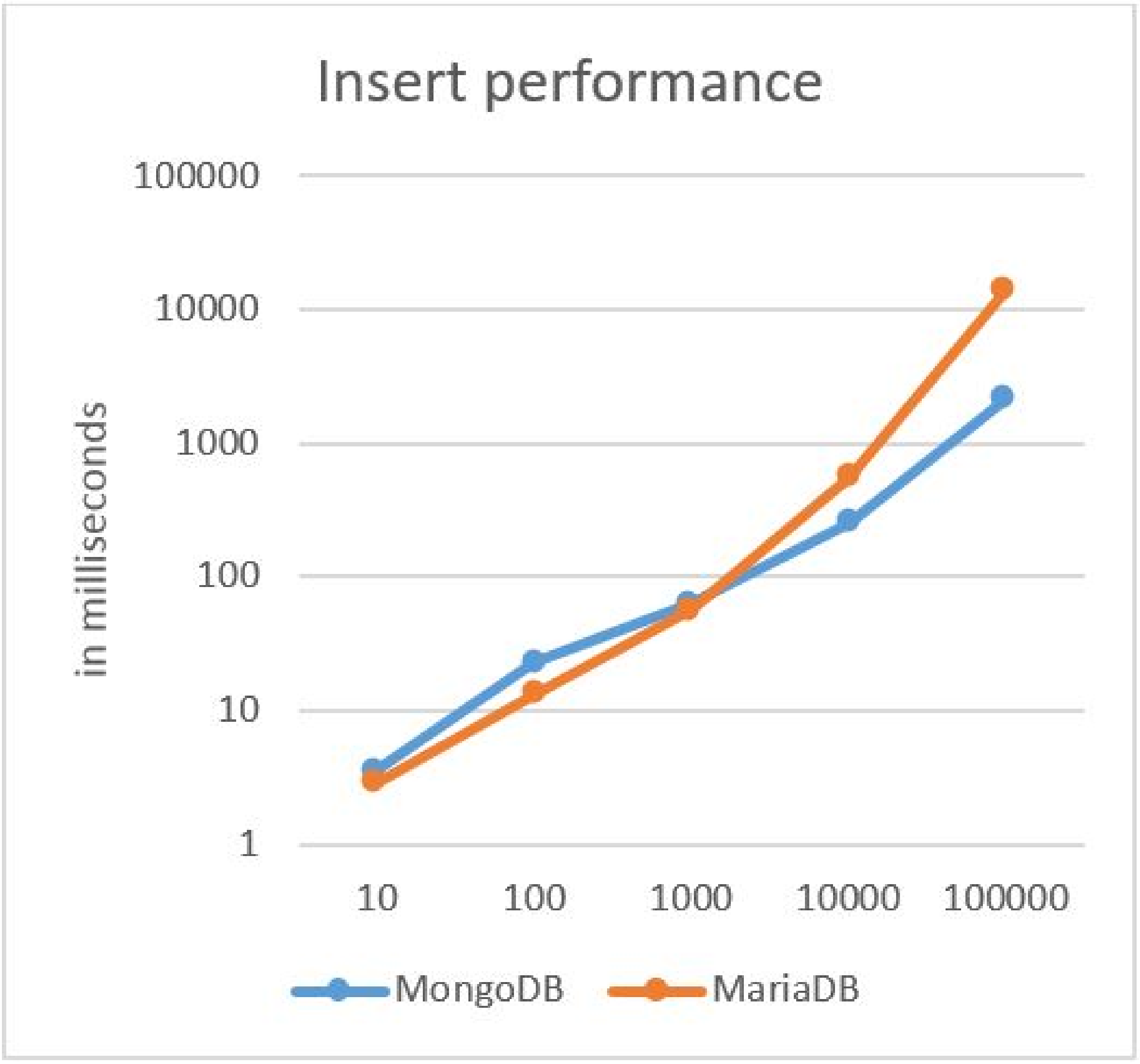


Figure 7.4.: Insert MongoDB vs MariaDB

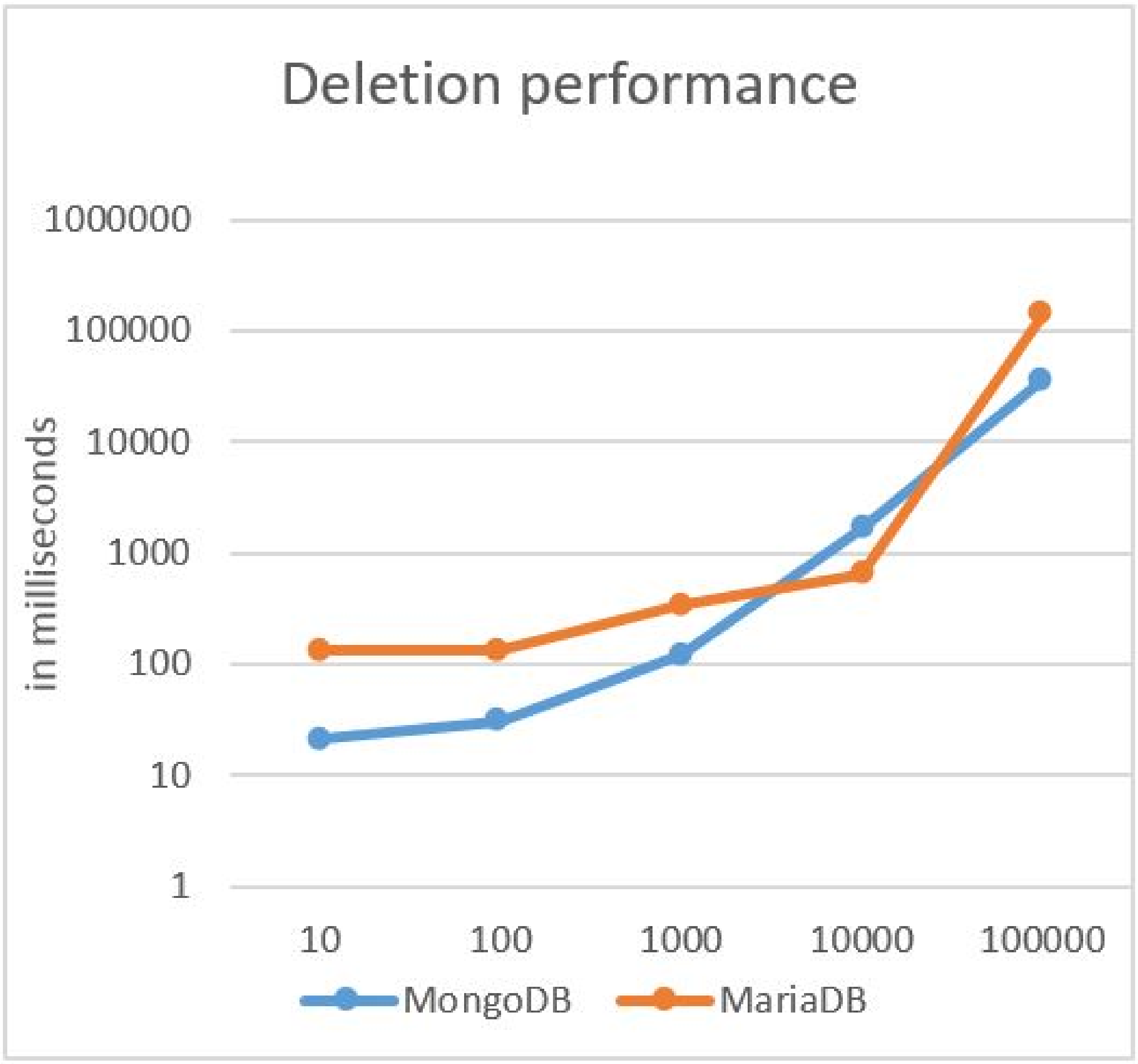
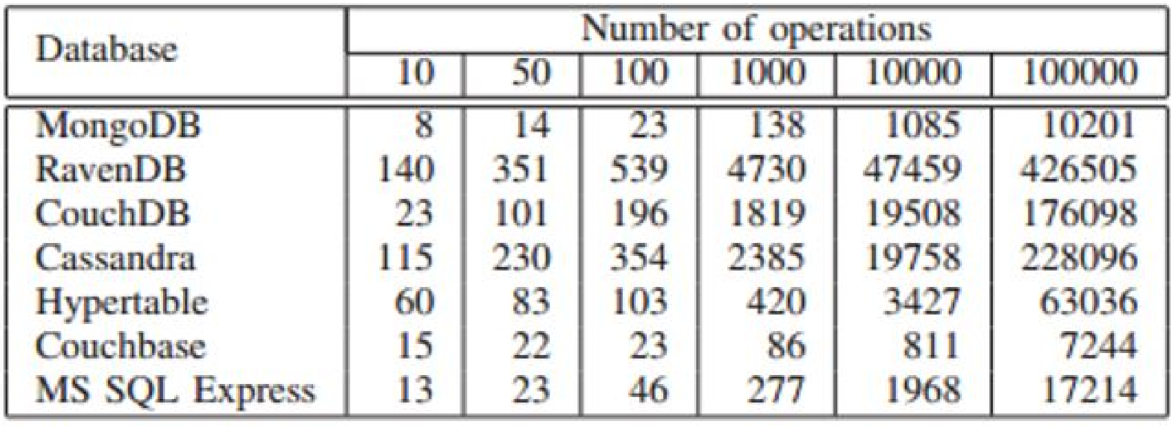


Figure 7.5.: Deletion performance

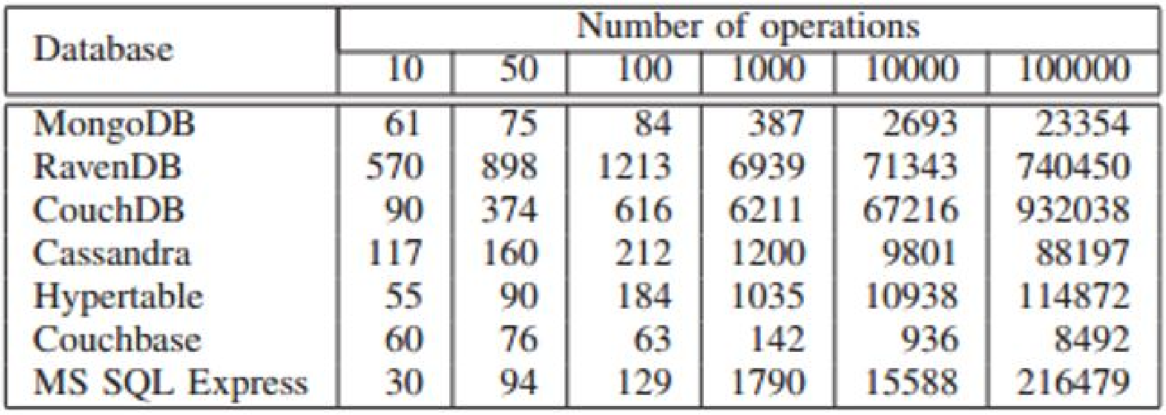
The numbers show an interesting result. In the insert test MongoDB performs worse than MariaDB just slightly up to the point of 1000 inserts. After that point MariaDB falls behind. With increasing number of inserts the performance leap of MongoDB increases a lot. A reason for this could be some kind of overhead for inserts on MongoDB, further investigation is needed to evaluate the results and the cause. A similar behavior can be seen on deletion test.

The following performance results are provided by the paper (Li & IEEE, 2013)



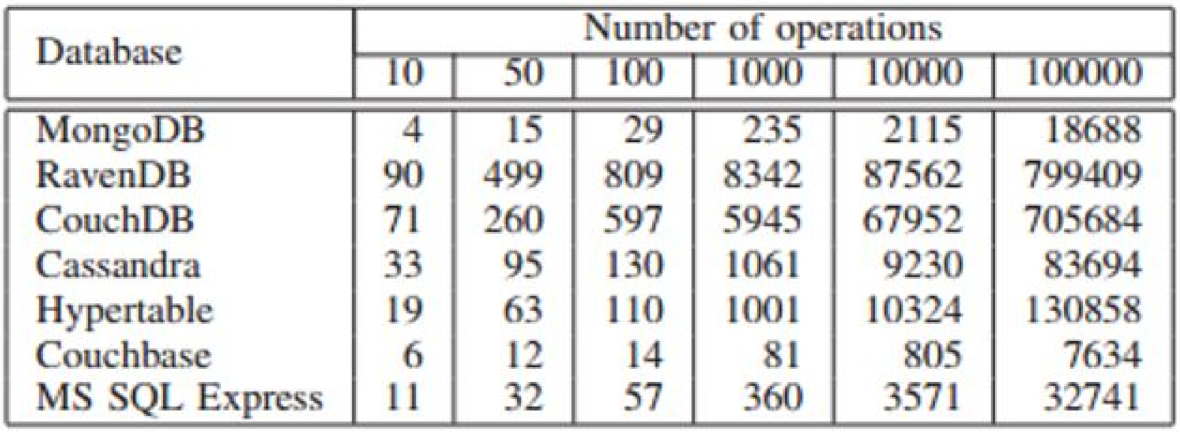
Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	8	14	23	138	1085	10201
RavenDB	140	351	539	4730	47459	426505
CouchDB	23	101	196	1819	19508	176098
Cassandra	115	230	354	2385	19758	228096
Hypertable	60	83	103	420	3427	63036
Couchbase	15	22	23	86	811	7244
MS SQL Express	13	23	46	277	1968	17214

Figure 7.6.: Time for read operations in milliseconds



Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	61	75	84	387	2693	23354
RavenDB	570	898	1213	6939	71343	740450
CouchDB	90	374	616	6211	67216	932038
Cassandra	117	160	212	1200	9801	88197
Hypertable	55	90	184	1035	10938	114872
Couchbase	60	76	63	142	936	8492
MS SQL Express	30	94	129	1790	15588	216479

Figure 7.7.: Time for write operation in milliseconds



Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	4	15	29	235	2115	18688
RavenDB	90	499	809	8342	87562	799409
CouchDB	71	260	597	5945	67952	705684
Cassandra	33	95	130	1061	9230	83694
Hypertable	19	63	110	1001	10324	130858
Couchbase	6	12	14	81	805	7634
MS SQL Express	11	32	57	360	3571	32741

Figure 7.8.: Time for delete operation in milliseconds

Interestingly the benchmarks from the paper show similar results on write performance when comparing the SQL implementation and MongoDB. After 100 writes MongoDB performs better. When comparing to other NoSQL databases, MongoDB is always in the upper half of the field. So it can be said, that this database should be suited well for demanding applications and performance shouldn't be a major problem.

These statements only apply for single instance usage. Since MongoDB uses a single master architecture for multiple instances, throughput will be less than database systems that trade consistency for performance. Using sharded servers can increase performance for horizontal scaling. With this addition, MonngoDB is also very usable for scientific applications with high amounts of I/O and data (Dede, 2013).

7.7. MongoDB and Big Data

In terms of Mobile Applications, IoT, Industry 4.0 and cloud-computing, data is vast, unstructured, sometimes unwieldy and complicated. In this context, Big Data is identified by its velocity, variety and volume. Therefore, requirement and expectations has changed how to store, process and analyze data. It has led to the development of NoSQL databases such as MongoDB(MongoDB Inc., 2013a).
However, in the era of Big Data, there a 2 kind of database solutions for facing Big Data. We distinguish operational Big Data Systems and analytical Big Data solutions. Features of Operational Big Data Systems provides real-time, interactive, dynamic workloads that ingest and store data. MongoDB belongs to this category and is a popular technology for operational Big Data applications(MongoDB Inc., 2013a).
On the other hand, Analytical Big Data technologies are useful for retrospective, sophisticated analytics of your data. A most-known example of an Analytical Big Data technology is Apache Hadoop. Hadoop is designed for storing and processing large sets of data on a distributed environment based on commodity servers and storage. It is an open-source Apache project, which consists of a distributed file system called HDFS (Hadoop Distributed File System) and a data processing and execution model called MapReduce(Wadkar, Siddalingaiah, & Venner, 2014).
Choosing between operational and analytical Big Data solution isn't the right way of thinking about facing this Decision. Many organizations are harnessing the power of Hadoop and MongoDB together to create complete big data applications. At the one hand, MongoDB powers the online, real time operational application, serving business processes and end-users, exposing analytics models created by Hadoop to operational processes. At the other hand, Hadoop consumes data from MongoDB, blending it with data from other sources to generate sophisticated analytics and machine learning models. Results are loaded back to MongoDB to serve smarter and contextually-aware operational processes – i.e., delivering more relevant offers, faster identification of fraud, better prediction of failure rates from manufacturing processes(MongoDB Inc., 2013a).

In the following, you see a Figure which shows a Design pattern how to combine these two technologies to be ready for a Big Data environment:

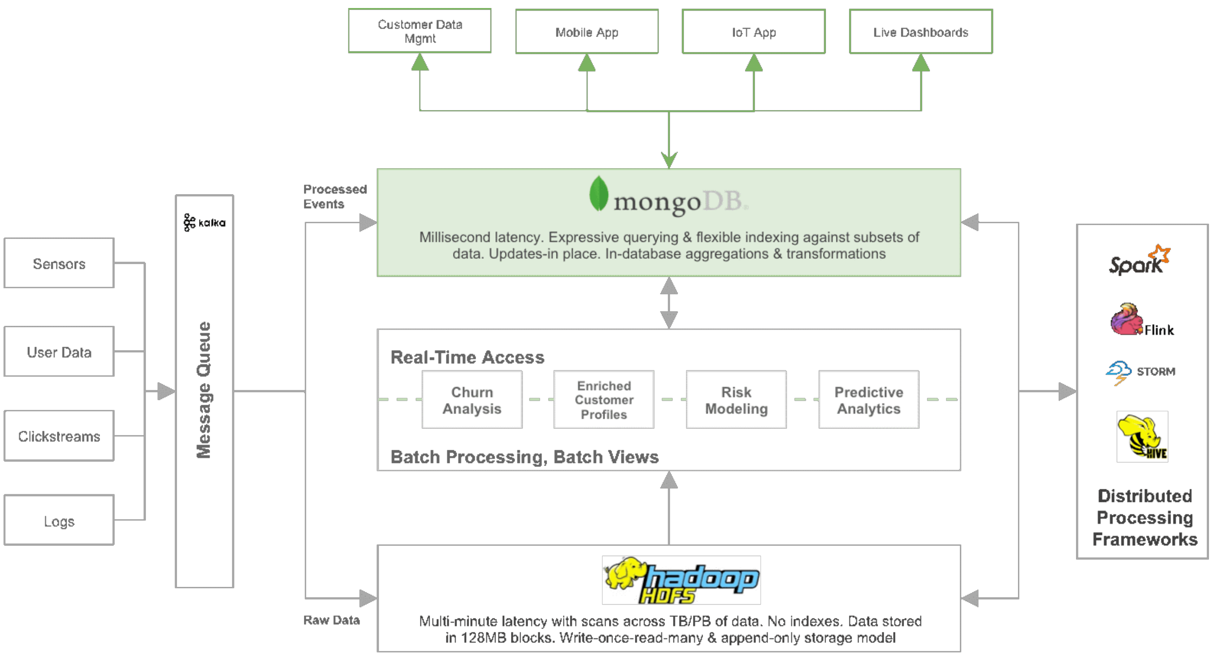

Customer Data Mgmt
Mobile App
IoT App
Live Dashboards
Processed Events
mongoDB
Millisecond latency. Expressive querying & flexible indexing against subsets of data. Updates-in place. In-database aggregations & transformations
Sensors
User Data
Clickstreams
Logs
kafka
Message Queue
Real-Time Access
Churn Analysis
Enriched Customer Profiles
Risk Modeling
Predictive Analytics
Batch Processing, Batch Views
Raw Data
hadoop HDFS
Multi-minute latency with scans across TB/PB of data. No indexes. Data stored in 128MB blocks. Write-once-read-many & append-only storage model
Spark
Flink
STORM
HIVE
Distributed Processing Frameworks

Figure 7.9.: Design pattern for integrating MongoDB with Data Lake (MongoDB, 2016)

7.8. Conclusion

All in all, MongoDB is a powerful and popular documented-oriented database. It comes with a lot of features, which meets the modern-day requirements and challenges for business applications, platforms and the web. The examples about the MongoDB data model, queries and CRUD operations showed how easy it is for developers to setup and work with this database. MongoDB is also very flexible in terms of extending the structure of documents. However, it is strongly recommended to consider the usage of common patterns to create a strong and future-proof data model. The usage of patterns will also help developers to understand a given database structure and work together on larger projects. Even in a big data context MongoDB is a good choice for big data applications. That is why you will find a lot of companies and startups using MongoDB in their development.

7.8.1. MongoDB in CAP-Theorem

In a single server or Master/Slave configuration MongoDB prioritize consistency over availability. That might be the reason why in most literature MongoDB is positioned

at CP-side of the CAP-Theorem. But in *Replica Sets* it is possible to trade some of the consistency for a higher availability, by configuring *read preferences* (7.5.2). By allow reading from secondaries, there is no way to ensure the client is reading consistent data. „This behavior is characterized as eventual consistency, witch means that although the secondary's state is not consistent with the primary node state, it will become consistent over time“(Edward & Sabharwal, 2015, p. 108). With *write concerns* (7.5.2) it is possible to obviate inconsistent reads happen to often, by ensure that a minimum number of secondaries is consistent.
Figure 7.10 describes how the three values change depending on the configuration. The partition tolerance is always fulfilled, because MongoDBs architecture is designed that way. By allow reading from secondaries availability will be increased at the expense of consistency and some consistency can be recovered using *write concerns*. But availability is always limited to reading, so it can never be fulfilled for all aspects of an application.

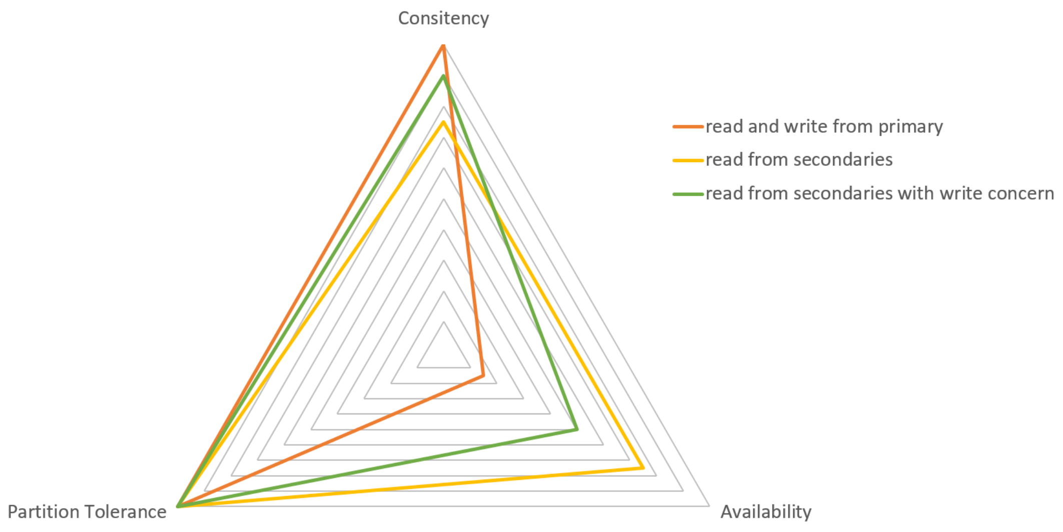


Figure 7.10.: Write process with write concerns

8. RethinkDB

8.1. RethinkDB

This chapter briefly introduces the major topics around RethinkDB, a real time open source shema-less document-based database. It is widely used by startups and Fortune 500 firms such as NASA, GM and Distractify. Founded in 2009 RethinkDB saw major funding by Y Combinator, had their first release 2012 and went to version 2.0 in 2015. October 2016 the company behind RethinkDB closed down due to the lack of business success but the software got then purchased in February 2017 by a Linux Foundation daughter: The Cloud Native Computing Foundation. It has then been re-licenced under Apache Licence 2.0, going away from their initial copyleft-like licence (CNCF, 2017; Kincaid, 2009).

8.2. How RethinkDB works

8.2.1. Data storage in Rethinkdb

The data structure

RethinkDB stores data as a B-tree-structure. A B-tree is a data structure which is balancing itself and allows filtering and sorting within logarithmic time. This B-tree is saved on a file system instead of the RAM in large structures (Rouse, 2005). This file system is called BTRFS (B-Tree File System), This enables the copy-on-write scheme provided by BTRFS thus making it possible to repair saved data from the copied clone. Other benefits to this filesystem are a garbage compactor to reduce older copies, low cpu-overhead, optimization for solid-state drives, data consistency (more detail about consistency in the ACID and CAP chapter 8.2.3 , b-tree aware caching and multi version control8.2.4. B-Tree aware caching is a way to give RethinkDB the capability of using far more data than available RAM.
RethinkDB does not include a hardware data consistency, this has to be maintained by

the used file system itself, which RethinkDB supports the most commonly available (The Linux Foundation, 2017c)

partioning and multi-datacenter support

Data in RethinkDB can be saved on multiple servers. This is done by replicating databases and providing each of them with a specific tag such as 'de_east' or 'westeros'. A table can have an non-specific amount of replicas on each server. On servers data can be partitioned into shards and furthermore tagged like replicas. The partition is done by a range of specific sharding algorithms and uses the primary key of each table. This means, that a shard key and a primary key is identical. For example if a data set has a primary key containing only letters and is ordered alphabetical, the sharding algorithm will likely split the data around the key 'm'. Thus the new two partitions containing every data with a key from 'a' to 'm' and from 'n' to 'z'. Evidently this algorithm always tries to part at the best pivot to have new partitions in equal size.(The Linux Foundation, 2017f, "How does multidata-center support work")

8.2.2. The atomicity model

Atomicity means, that either the complete stack of operations will be executed or non at all, there is no middle ground (Rouse, 2006). Operations within single json documents are guaranteed to be atomic, queries accessing different keys are not as they may be inconsistent in read and write operations (jepsen.io, 2016). The atomicity differs itself from other NoSQL databases by the way, that every set of operations, every chained query is atomic by the restriction named above. Plus, another limit is, that upon executing a non deterministic operation RethinkDB will nolonger be able to ensure their atomicity. In this case, RethinkDB will automatically throw an error by default. This behaviour may be shut by setting the according flag (The Linux Foundation, 2017d, "How does atomicity work").

8.2.3. ACID and CAP theorems in RethinkDB

RethinkDB's architecture is based, as mentioned in the section above 8.2.2, on the atomicity model. This model is part of the ACID paradigm for databases. ACID is a acronym for atomicity, consistency, isolation, and durability and describes the key desired parameters during a transaction to and off of a Database. ACID is norminized under ISO/IEC 10026-1:1992 Section 4 (Rouse, 2006). RethinkDB has also support for every other ACID paradigm except full isolation and absolute data consistency and therefore might not the best choice if full ACID-Support is needed. But RethinkDB provides a basic consistency as specified within the CAP-theorem. What this theorem is, has been elaborated within the Architectural Basics of Cassandra 2.2.3. The basic consistency within RethinkDB has been ensured by the fact, that every shard has a single replication and read and write actions are performed on this replication but not on the shard itself. Data remains immediately consistent and conflict-free. The Database also provides availability needed by the CAP-theorem as data is also accessible both up-to-date and out-of-date. Out-of-date queries are executed on a snapshot and without trying to get the most current data set. This means, that these queries are faster and guarantee availability but may not return nor access the most current data. The up-to-date queries are assuring that they return the latest data consistent and artifact-free. As before mentioned data is not absolutely consistent. This tradeoff roots within the partitioning. RethinkDB assures data to be consistent on the same network but cannot do the same for network partitions if data has to be up-to-date.(The Linux Foundation, 2017a).

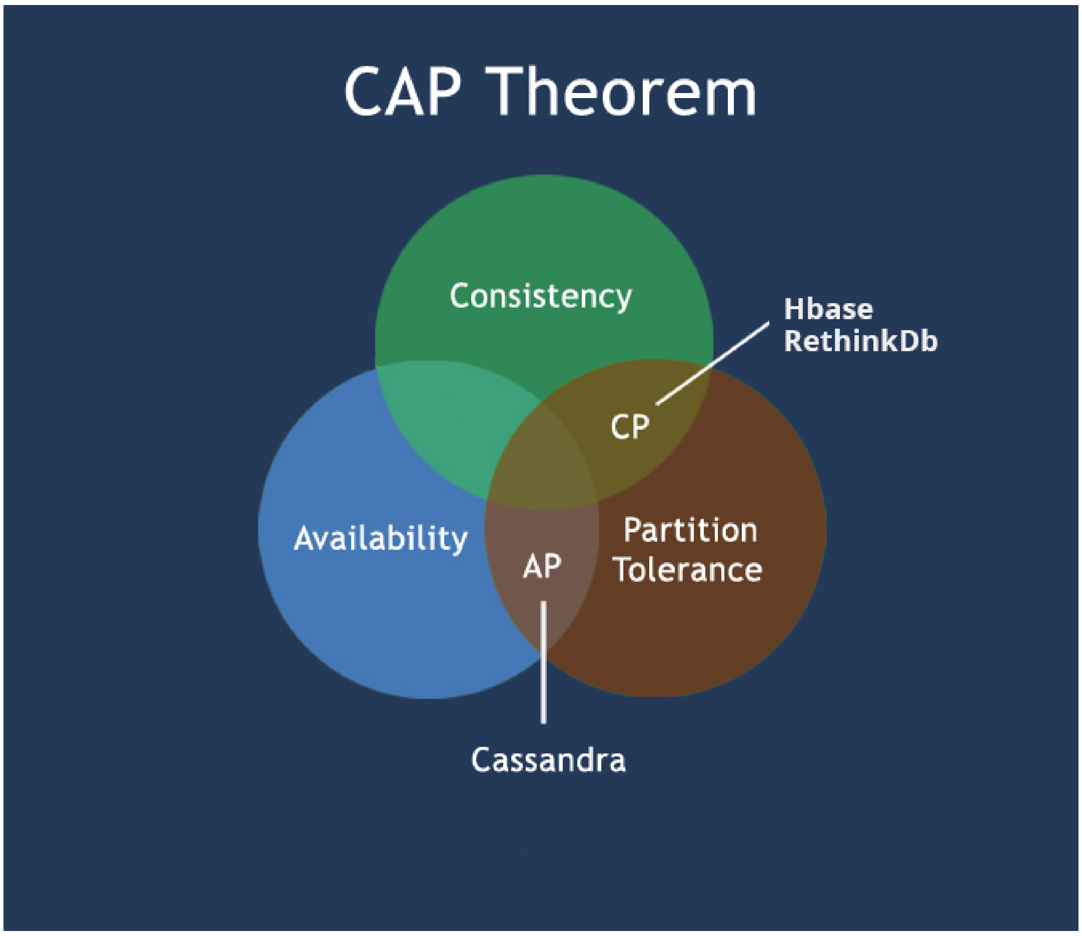


Figure 8.1.: Position of RethinkDB in the CAP-Theorem (*CAP Theorem*, n.d.)

8.2.4. multiversion concurrency control

RethinkDB has built-in support for multi version concurrency control. This means, that every writing operation is by default only made after a snapshot, simply a read operation, has been taken off of that tree. This has following benefits:

- Easy roll backing to previous data
- Lock-Free read and write operations
- Enables non-blocking queries making real-time hour long possible

(The Linux Foundation, 2017d, "How are concurrent queries handled? ")

8.3. The query language ReQL

RethinkDB has its own query language called ReQL. A query contains the table, an action and the order to run on a specific database connection. For Example the query to get an id (1) from a table (test) looks like this:

```
r.table("test").get(1).run(connection)
```

Listing 8.1: get document from Database (javascript driver)

One idiomatic aspect of ReQL is already evident in this example. ReQL is a chain of commands. Multiple queries can be written as a one-liner and are thus executed as one without disturbance. If there are dependencies on another query one should use this chain-technique to make sure it is executed in the right order. This works due to the fact, that the query is only parsed and executed on the server while being build on the client. On top of that, RethinkDB is lazy executing the queries. It immediately stops upon satisfaction. Furthermore are those queries functional and allows adding lambda functions as parameter. RethinkDB has build-in support for example for javascript code through the V8-Engine, map-reduce, table-joins and math (The Linux Foundation, n.d.-a).

```
r.db("test").tableCreate("test", options).run(connection)
```

Listing 8.2: create the table 'test'

Every table created gets a primary key for indexing. By default it is id but this can be changed by providing an option:

```
{primaryKey: 'name'}
```

Listing 8.3: options for create table

By setting this option as in the example, the table now is indexing every row under „name“. TableCreate has, as most of the many different options available, accessible in their documentation (The Linux Foundation, n.d.-d).

8.3.1. query executing

RethinkDB has a special way of executing a query. One key point is, that the query is not executed on a client but on the server itself. By receiving the query, the server creates a list of instructions consisting of internal logical operations. RethinkDB now tries to make this list as efficient as possible by executing most basic operations first and more time consuming, such as manipulating data last. Each operation set is called Node and their complexity ranges from single document queries to deep complex subquery commands. After executing, the server returns his result as a datastream not only to the client itself but also to every other relevant server. This has the benefit, that RethinkDB does not really care on which server the query is executed and therefor can parallize this process. (The Linux Foundation, 2017d, "How does RethinkDB execute queries?")

8.4. RethinkDB perfomance analysis

RethinkDB had has a major performance issue since its start but kept on improving on this aspect over the years. For example in a comparison in 2014, MongoDB has been 3 times faster than RethinkDB at executing queries over a large patent data set(juristat.com, 11.04.2017). A newer comparison of 2015 turns this around but only for writing data to the database. Reading, RethinkDB has still been half as fast as MongoDB has been in that benchmark(Thangarajan, 2015). Within an official benchmark test, the RethinkDB Team got this result:

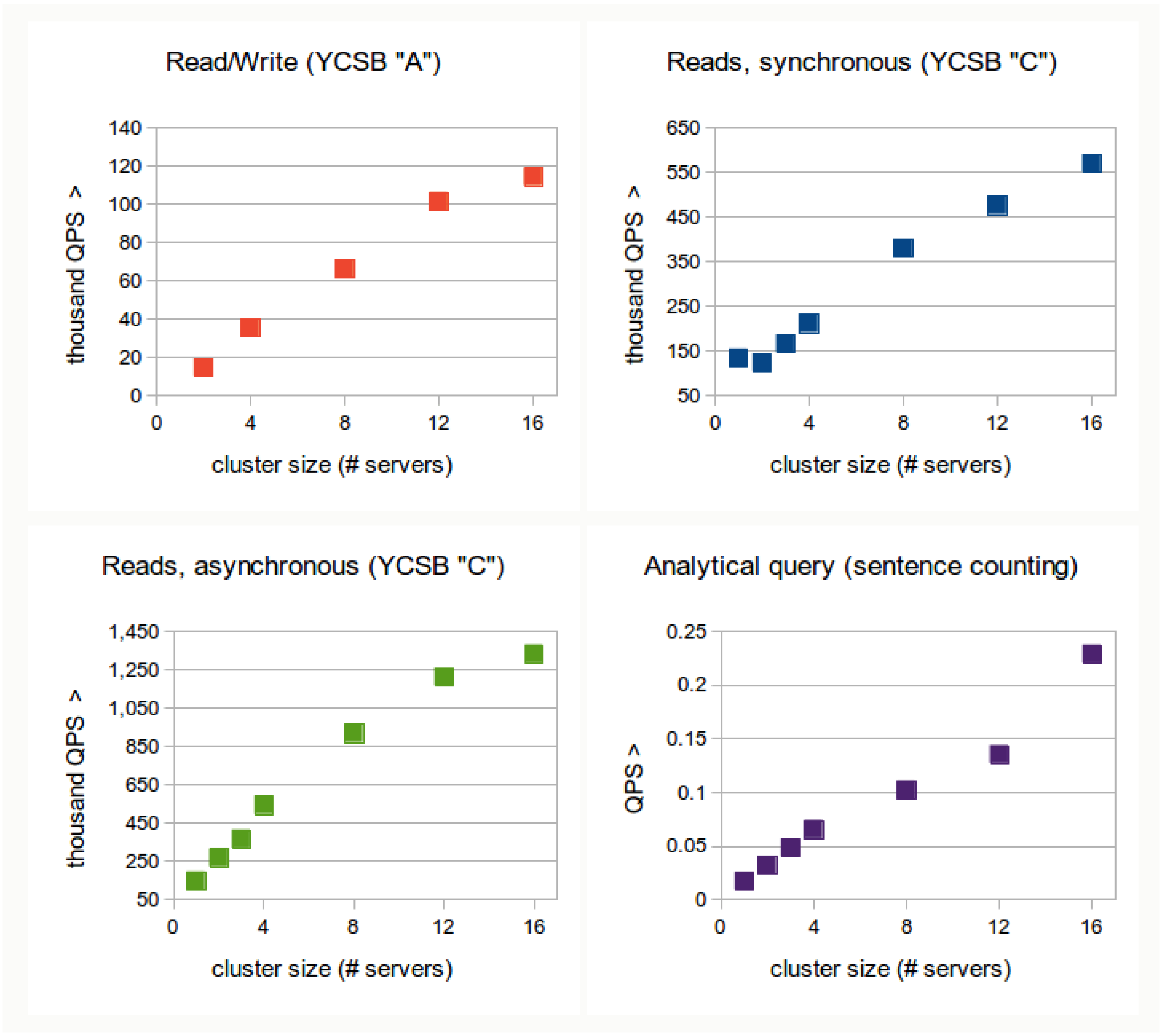


Figure 8.2.: Performance Report of RethinkDB (The Linux Foundation, n.d.-c)

They states, that in their working environment they were able to perform around 16 000 queries in a second (QPS). For this benchmark, RethinkDB used "analytic workloads in a simplistic but very common fashion" (The Linux Foundation, n.d.-c).

8.5. Limits

As every existing technique RethinkDB, too, has restictions. These can be hard or soft limitations depending on their context. (The Linux Foundation, n.d.-b)

It is not possible to create more than 64 Shards in RethinkDB. The number of databases are not limited by anything than diskspace and RAM as is the number of tables. The recommended size limit for a single Document is 16MB and should be respected due to potential perforance issues. Also do Arrays in a RethinkDB Server have a limited length.

This length can be configured per query and by default is at 100.000 elements. Primary keys are limited to 127 characters. But secondary keys aren't hard limited by those 127 characters, if they exceeds their limit, RethinkDB will use linear search for the following characters.

JSON queries are hard limited as well. They can only be 64 MB in size or smaller.

The ordering in RethinkDB is byte-wise. The functions orderBy and between uses the byte representaiton of the carater. This has the result that RethinkDB dose not normalize identical characters with multiple codepoints. For example the character "é" has the UTF-8 representation `\u0065\u0301` and `\u00e9`, the will be grouped separately.

The usage of the cli option `-direct-io` is restricted to supporting file systems only. Typically encrypted and compressed file systems will not support this option.

Numbers are double precision IEEE 754 floating point. Integers are stored precisely from -2^{53} to 2^{53}, outside that range they will be rounded.

8.6. Data Types

RetinkDB has all the basic Data types. Number, string, boolean, object, array and the `null` value. Additionaly RethinkDB stores specific data types like tables, streams, selections, binary objects, time objects, geometry data types, and grouped data. (The Linux Foundation, 2017e)

Numbers can bee Integer or Floatingpoint values. They will be stored with 64-bit percision. `NaN` or Infinit can not be stored.

String are stored with UTF-8 encoding.

Booleans can be `true` or `false`

Null is a special value. It is not the number zero or an empty string. It symbolizes the absence of any other value.

Objects are key-value pairs. Any valid JSON object is a valid RethinkDB object.

```
{
  "key"  : "valueString",
  "key2" : false,
}
```

Listing 8.4: example Object

The keys can only be strings, but the values can be any data type. A RethinkDB Document is a object.

Arrays are lists of values. Arrays can be empty. It is not enforced to use only on type of values in an Array. But it is highly recommended.

```
[
  "valueString",
  false,
]
[]
```

Listing 8.5: example Arrays

The values of an Array can be any data type. If you use arrays for many values, more than 100.000 you should notice that arrays are fully loaded into Server RAM before being send to the client.

Specific Data types

Databases are returned after a `db` function call.

Tables are returned after a `getAll` function call. Their behavior is similar to selections.

Selections are the result of `filter` ot `get` function calls. Selections comes in three variants `Selection<Object>`, `Selection<Array>` and `Selection<Stream>`.

Streams are like arrays, lists. Streams are not writable and are lazy, they give you the next dataset if the current has been processed. With Streams you can read tremendous long lists of Data, without holding everything in Memory.

8.7. Changefeeds

Changefeeds are the main part of RethinkDB's real-time functionality. Almost all queries can be used as a changefeed. With a Changefeed you recive any changes on your query. (The Linux Foundation, 2017b)

```
r.table('project').changes().run(conn, (err, cursor) => {
  cursor.each((err, row) => {
    if(err) throw err;
    processRow(row);
  })
});
```

Listing 8.6: example usage

The `changes()` method returns a cursor. The `each()` method of that cursor object iterates over each change. If the underlaying data of the query changes the callback of `each()` will be called. For example a new project will be added `id:3, name: "awesome Project", priority:` into the project table. The `each()` callback will be called with the changes of the table.

```
{
  old_val: null,
  new_val: {id:3, name: 'awesome Project', priority:1}
}
```

Listing 8.7: example usage

If the callback for `each` returns false, the cursor will stop iterating.

single document changefeeds

The single document changefeed only notifies the subscriber if the document changes.

```
r.table('project').get(3).changes().run(conn, callback);
```

Listing 8.8: example usage

This would listen for changes of the project with id 3.

Filtering Changefeeds

The method `changes()` integrates with the ReQL language. `changes()` can be used with mostly any outher command.

The chaning of ReQL methods can be as complex as you wish. For Example changefeeds can be filter:

```
r.table('project').changes().filter(
    r.row('new_val')('priority').lt(r.row('old_val')('priority'))
)('new_val').run(conn, callback)
```

Listing 8.9: example usage

This will be notified on every change, there the priority is lower as the previous. Chaining methods with changefeeds has some limitations.

handling latency

With some latency on the client side, it is possible that there will be mutliple writes into a table, before the clinet gets all changefeed events. By default a changefeed subscriber will only get one change object. If this is not the desired procedure the option `squash: false` should be used on the `changes()` method. With this the change object will be send for every change.

Changefeeds do not have a delivery guarantee.

8.8. Conclusion

RethinkDB is a highly potential database with many use cases and some valuable customers. Being consolidated under the Linux Foundation, this potential might be more furthered and current problems, like the performance issues reduced. In this future, the lack of documented community drivers will hopefully evaporate and more and more third libaries will be added to RethinkDB. Even though the namespacing, with just the *r*, might be strange to use at first but due to the query chainability, it's many functions and the changefeed make up a lot for the uncertainty resulted in shut down and revival. RethinkDB's easiness in usability and installation-wise, it's simple web interface and the well-elaborated documentation are reason enough to give it a shot.

To synthesize it, RethinkDB is worth considering for web & app projects alike especially if real time support is needed.

Part V.

Graphbased DB

9. neo4j

9.1. Introduction to Graph Databases

Nowadays a huge number of different databases systems exist. You always must select the best suiting one for your specific use case. Data is often ordered in relations between objects, like different cities and their connecting streets. If there is the possibility to structure your data in the database the same way as the data is structured in real life, you may get a speed and usability advantage out of that. The best solution to picture these structures is to use the graph. That is why graph databases have been developed, which now will be presented to you.

9.1.1. Graph Databases

Graph databases use graph structure to order data in the database. There is more than one solution for using graphs. The main aspects of the first are nodes and edges, which are essential for the database structure (Robinson, 2013, pp. 1-4). The connection between the nodes are the relations of the data. In the example picture below (Rouse, 2016, para. 1), you can see how graph databases work with the nodes and their relations. For example you can see the married couple Julie and Bob.

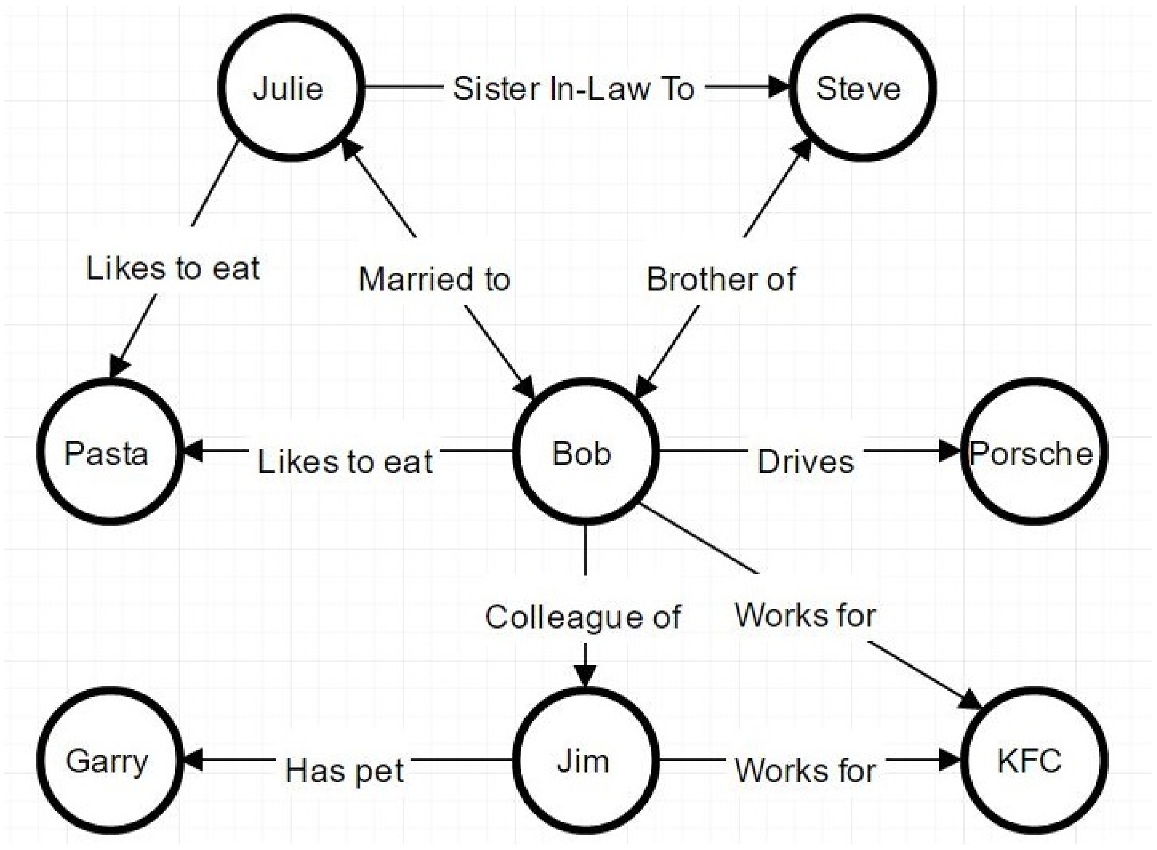


Figure 9.1.: ExampleGraph

You can also build property graphs, which you can use for routing. For example, you can find the shortest way from one city to another. The special thing about property graphs is, that attributes can be added to the nodes and edges. So you can store further information in your graph (Rodriguez, 2010, p.3).

The second one is used for the semantic web and concentrates on triples. Semantic web stands for machine processable web. In the semantic web the W3C consortium defined a framework how to work with the triples (W3C, 2014, "Overview", para. 1). It is called Resource Description Framework and its structure is like following:

- First part: subject, where the relation starts.
- The second: predicate, the relation
- The third: object, where the predicate points to.

Despite you can see the same node and edge structure, it is different to the property graphs. In Resource Description Framework, which is the most common used framework, you can only build connections between the nodes, but you can neither set properties to the nodes nor to the edges.

9.1.2. Neo4j

Neo4j - developed by Neo Technology - is one of the first and up to now the most popular graph database implementation. Neo4j is more than five times popular than the second graph database - OrientDB (Solid IT GmbH, 2017, para. 1).
Its first Version was released in 2010 after three years of development. It is called as transactional, disk based database, which follows the ACID principle (Neo Technology, Inc., 2017h, "Neo4j Internals", para. 4). The implementation is in Java and can be used in two different license models. The Community and the Enterprise Edition.
The Community Edition is free, but only running on a single node. You can use all the features of Neo4j without high availability through clustering and hot backup. These additional modules are coming with the Enterprise Edition only. There are some further categorizations for the Enterprise version like the test licenses Evaluation, the Educational and the Neo4j Loves Open Source licenses (Neo Technology, Inc., 2017g, "About Neo4j Licenses", para. 1).
Neo4j uses Cypher as query language. With indexing and the labels of the graph it helps to accelerate the queries, which is one of the main advantage of using graph databases and especially Neo4j.

Now you get a deeper insight into Neo4j and its data structure.

9.2. Data Structures

In opposite to the bulk of NoSQL databases Neo4j as a graph database is focused on the relationships between data. Therefore the data structure is based on two components: nodes and relationships. Both of them can contain a variety of informations. Let's start with the nodes. Each node consists of a JSON object which defines its properties, so the nodes can be compared to documents in MongoDB for example. In the definition of this node properties you can use anything, that's provided by the JSON standard. In a library for example we could have books with properties about their title, page number or publication date. With this properties we get the ability to store data and in connection with the later introduced cypher query language we can search for nodes by them. But currently searches would be executed over the whole graph and any node can contain different properties. (Neo Technology, Inc., 2017c, "Nodes", para. 3) (Gupta, 2015, p. 80) (M. Hunger, 2013, slide 20-21)

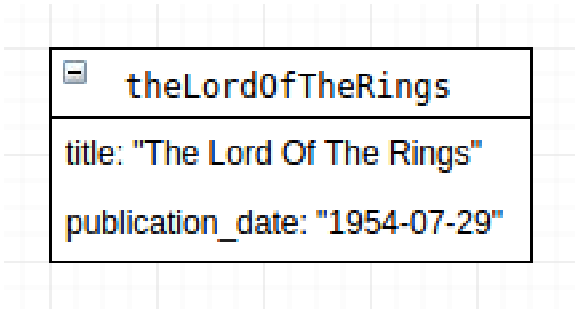

theLordOfTheRings
title: "The Lord Of The Rings"
publication_date: "1954-07-29"

Figure 9.2.: Single Book Node

To structure the data, every node can be labeled. This labels can be interpreted as a typing. Looking back at our library example we could create nodes labeled as “book”, “author” or “borrower”. By labeling the node we organize them in sets, like the MongoDB collections or SQL tables. Beside the better structure we achieve a more efficient searching, because the amount of data to search in can be reduced right at the beginning, by the definition of the node set (label) to search in. At this point we have nodes with properties organized in sets by their labels, but no relationships. (Neo Technology, Inc., 2017c, "Labels", para. 2) (M. Hunger, 2013, slide 26-27)

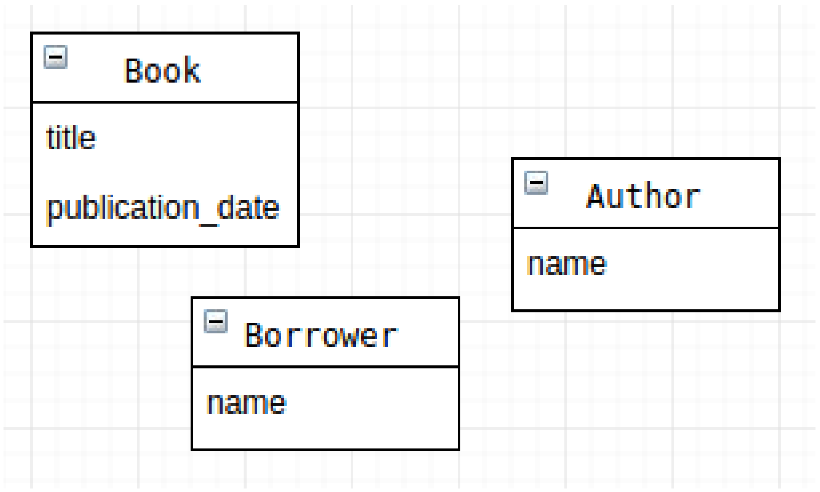


Figure 9.3.: Labeled Node Sets

The relationships in Neo4j are quite similar to the nodes. They are also able to take properties in form of JSON objects and has labels to define their type or in this case better to say their function. In the library example a relationship between book and borrower could be labeled as “borrowed” and contains properties like “borrowDate”, “returnDate” or maybe “rating”. Here we should also have a look at a good structure, so it would be better to extract the rating and put it in another relationship like “read”. The main and obvious difference are the two connected nodes. So every relationship needs exactly two nodes connected to it. But they do not have to be different, so it"'s possible to point a relationship on its own origin. Naturally the node labels are irrelevant here, so relationships can be defined inside and between node sets. (?, ?, "Relationships", para. 1) (Gupta, 2015, p. 81-82) (M. Hunger, 2013, slide 22-25)

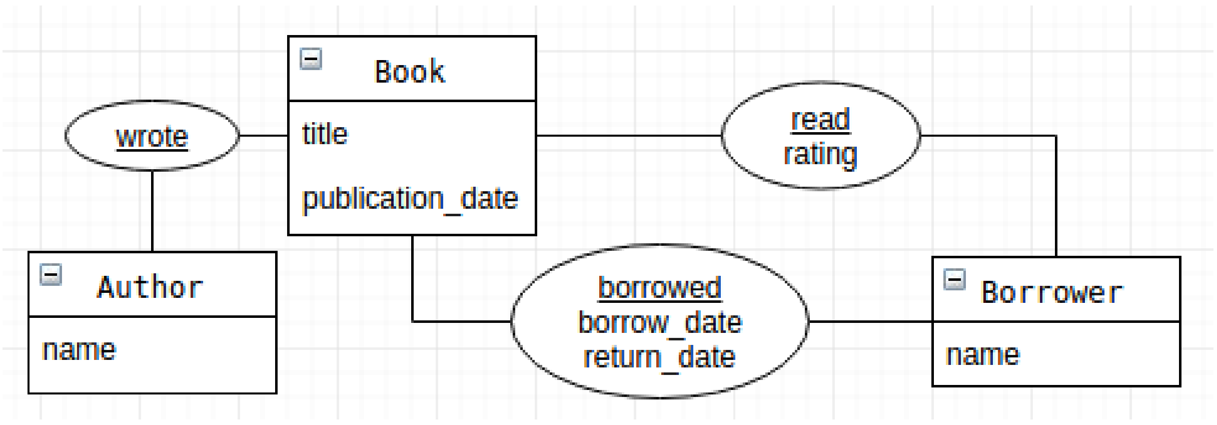


Figure 9.4.: Related Nodes

After looking at the elements of the Neo4j data structure, we will now have a look at how we create this structure in our database instance. At first it"'s necessary to create some nodes. Therefore we use the CREATE command followed by round brackets to define the node itself. The first element of the node definition is an optional variable name at first to save the reference to the created node, if needed. After this a colon and label name is required to define the node set. The label doesn"'t need to be defined before using it here. At last we enter the JSON object with our node properties and close the definition brackets. (Neo Technology, Inc., 2017e, "Create a Record for Yourself", para. 1) (Gupta, 2015, p. 80)

```
CREATE ( variable:LABEL {} )
CREATE ( theLordOfTheRings:Book { title: 'The Lord Of The Rings',
    publication_date: '1954-07-29' } )
CREATE ( jrrTolkien:Author { name: 'J. R. R. Tolkien' } )
CREATE ( johnSmith:Borrower { name: 'John Smith' } )
```

Listing 9.1: Create Database

Secondly we can create the relationships. Thanks to the ascii-art syntax it"'s easy to understand the create statements. It starts again with a CREATE and is followed by round brackets. In this case this brackets can contain a node definition again, but also a simple variable name to reference an existing node. After the brackets a hyphen connects the first node to the relationship definition part surrounded by square brackets. This definition is structured like the node definition: optional variable name, label, JSON object. An ascii arrow ("->") connects the closing square bracket with the second node, this relation will point to. In the style of the first node we have here round brackets with a variable or a whole node definition. (Neo Technology, Inc., 2017e, "Create a Record for Yourself", para. 1) (Gupta, 2015, p. 81-82)

```
CREATE (sourceNode)-[ variable:LABEL {} ]->(targetNode)
CREATE (jrrTolkien)-[ :WROTE ]->(theLordOfTheRings)
CREATE (johnSmith)-[ :BORROWED { borrowDate: '2017-03-15', returnDate: '
    2017-04-14' } ]->(theLordOfTheRings)
```

Listing 9.2: Create Relationships

Below you can have a look at the created graph of the simple example.

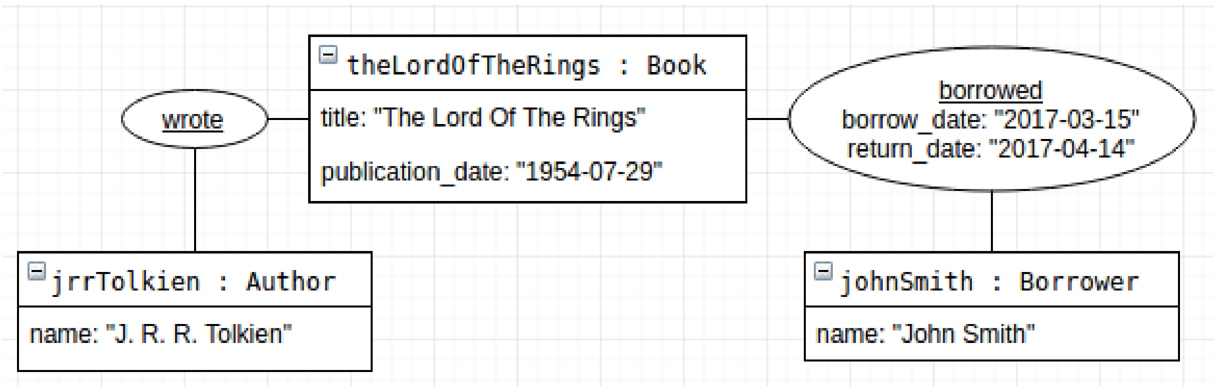


Figure 9.5.: Sample Graph

9.3. Cypher Language

Cypher is Neo4j's open graph query language (Neo Technology, Inc., 2017a, "Cypher Query Language", para. 1). It was newly created to match the data-structures of Neo4j and to fulfill the special needs of Graph-Databases. In addition it's based on SQL to allow an easy entry point for developers, which already had to work with SQL. (Neo Technology, Inc., 2017e, "About Cypher", para. 1) Cypher's syntax provides a familiar way to match patterns of nodes and relationships in the graph. Cypher is also a relatively simple but still very powerful language. (Mahler, 2014, "Fazit", para. 1) Very complicated database queries can easily be expressed through Cypher. This allows users to focus on their domain instead of getting lost in database access because it allows the user to state what he wants to select, insert, update or delete from his graph data without requiring him to describe exactly how to do it.

Example Cypher contains a variety of clauses. Among the most common are: MATCH and WHERE. (Neo Technology, Inc., 2017b, "A few words about Cypher", para. 3) These functions are slightly different than in SQL. MATCH is used for describing the structure of the pattern searched for, primarily based on relationships. WHERE is used to add additional constraints to patterns. For example, the below query will return all movies starting with "T", and return its cast as a collection:

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)
WHERE movie.title STARTS WITH "T"
RETURN movie.title AS title, collect(actor.name) AS cast
ORDER BY title ASC LIMIT 10;
```

Listing 9.3: Cypher Example

ASCII-Art and Nodes Cypher uses ASCII-Art to represent patterns. It surrounds nodes with parentheses which look like circles, e.g. **(node)**. (Neo Technology, Inc., 2017e, "Nodes", para. 1)

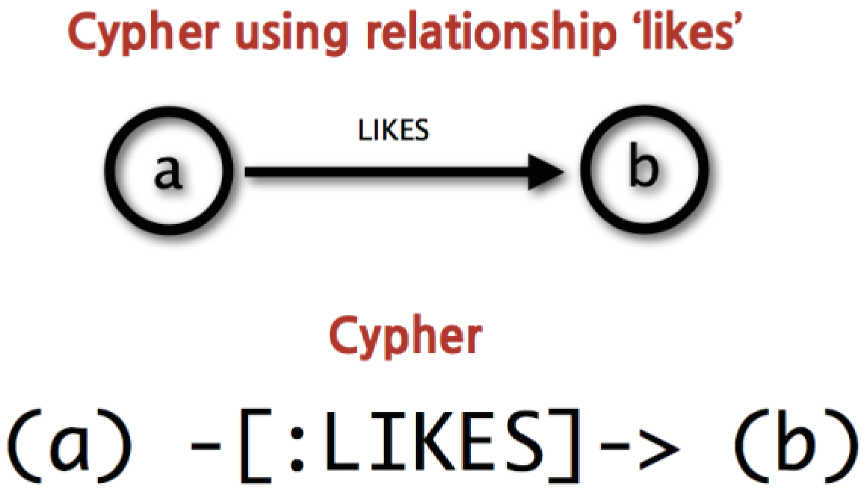


Figure 9.6.: node ascii art

To reference a node at a later time, it can be stored in a variable like (p) for person or (t) for thing. In real-world queries, there would probably be longer, more expressive variable names like (person) or (thing). If the node is not relevant to the question, the parenthesis can also be empty ().

Relationships To fully utilize the power of graph databases complex patterns between the nodes can be expressed. Relationships are basically an arrow −> between two nodes. (Neo Technology, Inc., 2017e, "Relationships", para. 1) Additional information can be placed in square brackets inside of the arrow.

This can be

- relationship-types like **-[:KNOWS|:LIKE]->**
- a variable name **-[rel:KNOWS]->** before the colon
- additional properties **-[since:2010]->**
- structural information for paths of variable length **-[:KNOWS ..4]->**

(Neo Technology, Inc., 2017e) ("Relationships", para. 3)

```
CREATE (you:Person {name:"You"})
RETURN you
```

Listing 9.4: Create a Record

CREATE creates nodes with labels and properties.

Figure 9.7.: graph you

```
MATCH (you:Person {name:"You"})
FOREACH (name in ["Tobias","Kai","Manuel"] |
CREATE (you)-[:FRIEND]->(:Person {name:name}))
```

Listing 9.5: Create Relations

FOREACH allows the execution of update operations for each element of a list.

```
MATCH (you {name:"You"})-[:FRIEND]->(yourFriends)
RETURN you, yourFriends
```

Listing 9.6: Show Relations

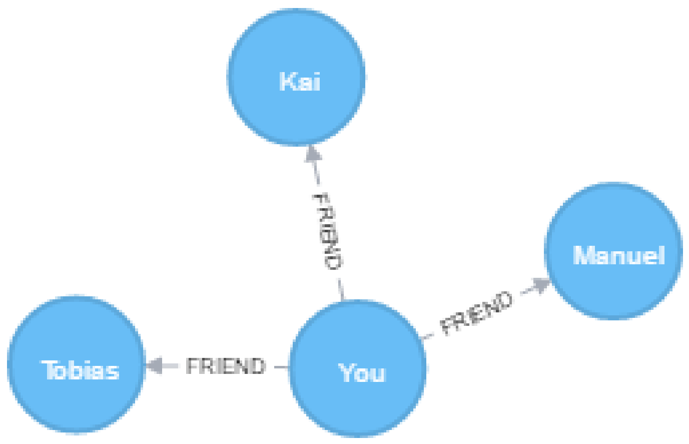


Figure 9.8.: graph friends

9.4. Comparison with other Database Systems

The following chapters will compare graph databases, especially Neo4j with other database systems such as relational databases and NoSQL (not only SQL) databases. In addition to plain comparisons there will also be some examples how graph databases can be used with other systems to maximmize the advantages of each system.

9.4.1. Comparison with Relational Databases

This chapter will give a basic overview of the similiarities and differences between graph databases and relational databases. More specifically this chapter will focus on Neo4j and SQL. In Neo4j relationships are first-class citizens. In SQL these relationships can only be created by using foreign keys and therefore Neo4j eliminates foreign keys. Each node contains a list of relationship-records. These relationship-records are organized by type and direction and can hold additional attributes. When you would normally run a JOIN-operation these records are used. This is the biggest advantage graph databases have over relational databases: The costs of expensive search and match operations are eliminated. This leads to much higher performance levels than those of relational databases. In addition the data models of graph databases are simpler and more expressive as seen in the below images (B. R. . L. W. Hunger M., 2016, pp. 9-10).

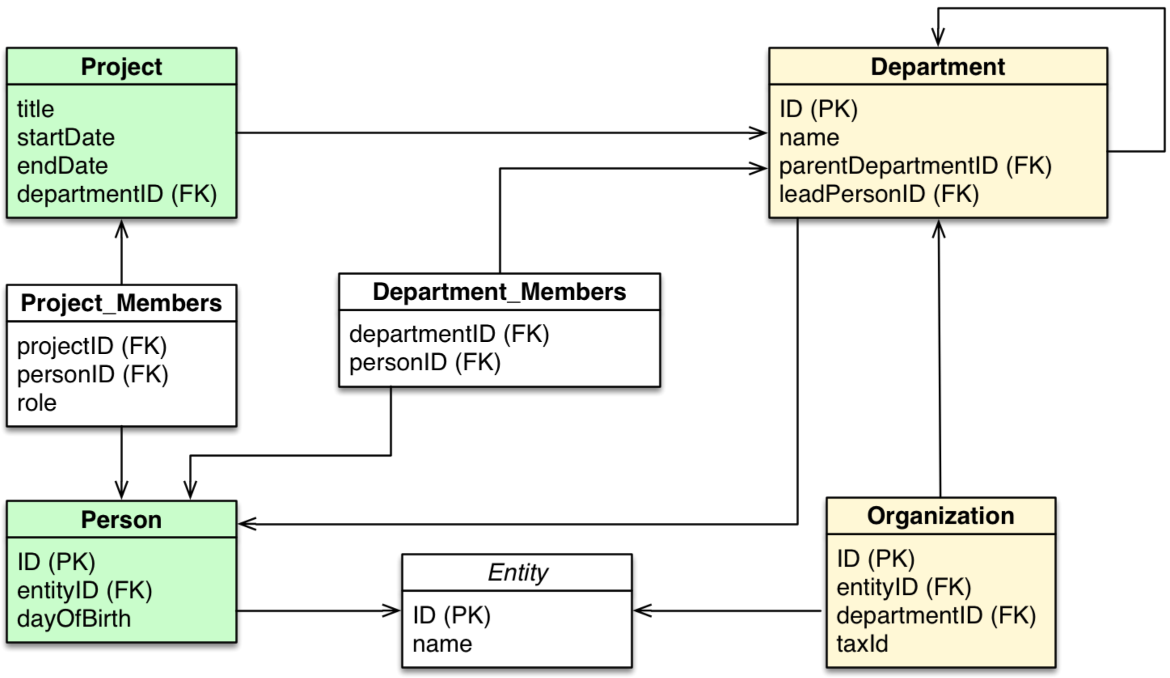


Figure 9.9.: SQL data model

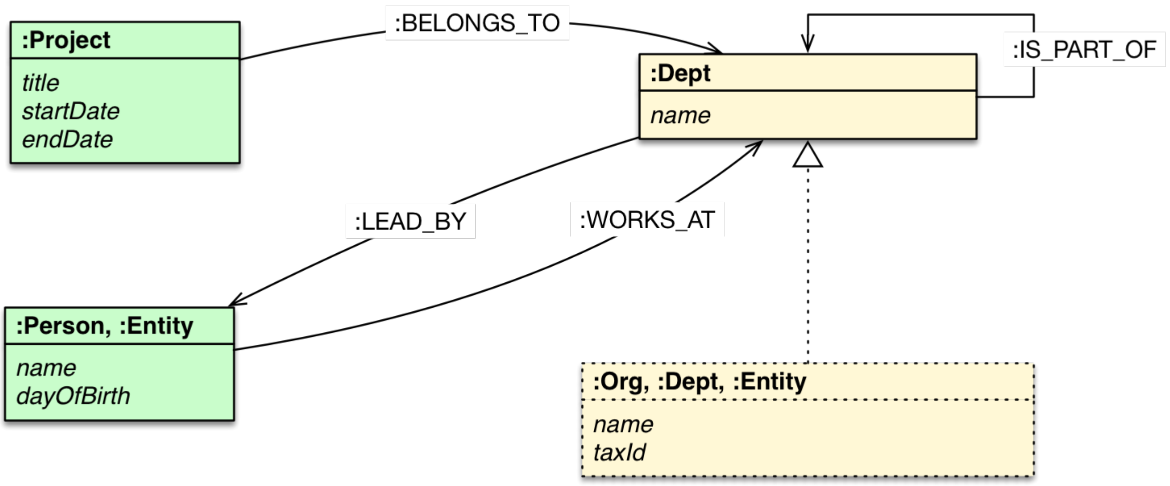


Figure 9.10.: Neo4j data model

Like SQL Neo4j also supports the transactional concepts (ACID). That means that data is never lost after it has been commited to the database. The query language is pretty similiar, but cypher, the query language of Neo4j, is more expressive. Following is a short comparison of the same transaction in SQL and Cypher. This example also demonstrates the strength of Cypher by eliminating two JOIN-operations. (Neo Technology, Inc., 2017d, "Working with Neo4j", para. 1)

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
```

```
WHERE d.name = "IT Department"
RETURN p.name
```

Listing 9.7: Cypher Statement

```
SELECT name FROM Person
LEFT JOIN Person_Department
ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"
```

Listing 9.8: SQL Statement

9.4.2. Comparison with NoSQL Databases

Since relationships are very important in graph databases, it's quite difficult to compare them with NoSQL databases, since they lack relations. Following statement by Webber and Robinson (2015) explains this scenario in a good way:

> Most NoSQL databases store sets of disconnected aggregates. This makes it difficult to use them for connected data and graphs. One well-known strategy for adding relationships to such stores is to embed an aggregate"'s identifier inside the field belonging to another aggregate — effectively introducing foreign keys. But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

(?, ?, p. 15).

9.4.3. Integration with other Database Systems

This section will describe how to use Neo4j together with other database systems in a very basic way. It will not go in-depth and there will be no code examples to keep it as simple as possible. To get the advantages of each database system, data needs to be stored in each database with its own data models. This is called polyglot programming: using multiple different languages, here multiple different database systems. There are existing tools for different database systems which can be used as some kind of connector to another system. The connectors let the other system subscribe to update events, so the data can be inserted in one database system and then added in the other system. The developers of MongoDB for example have created a tool called "mongo-connector" where other applications can listen for update events. This enables a one-way synchronization with Neo4j. Of course all the data model transformations have to be made manually, but once set up the full potential of both databases can be used. (Neo Technology, Inc., 2017f, "Goals", para. 1)

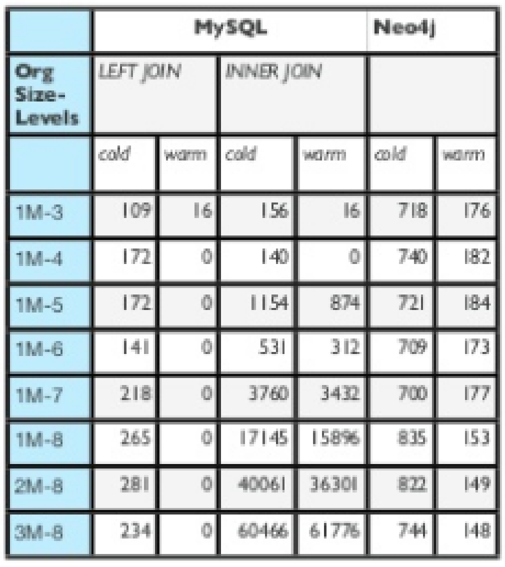
9.5. Conclusion

In the end you can say that Graph databases, and therefore also Neo4j, don't need joins to achieve relationships, since they are already first-class citizens. This has a huge impact on performance, since join-operations are very expensive. As Neo4j holds its data as JSON it has a similiar structure to document based databases and could be used together. That way you can get the advantages of both systems. The only disadvantage is the need to store the whole data twice, which requires more storage.

In general Neo4j fullfills the CAP-theorem in points of **c**onsistency and **a**vailablity and doesn't provide **p**artition tolerance. Because Neo4j is relationship-oriented, trying to achieve partition tolerance can cause many side effects and complex queries would decrease the performance considerable.

Neo4j has many use cases. For example it is used by companies such as Walmart for real-time recommendations, Ebay for logistics, LinkedIn as a representation for their Social Network and TomTom for geo-routing. An interessting fact at this point was published by the Neo4j staff. They said Neo4j is not used by Facebook, which "they should" change (Neo4j Staff, 2011, para. 5).

Neo4j is mainly used for relational analytics and is hence not efficient unless you have many joins. The following chart will enforce this statement.



	MySQL				Neo4j	
Org Size-Levels	*LEFT JOIN*		*INNER JOIN*			
	cold	warm	*cold*	warm	*cold*	warm
1M-3	109	16	156	16	718	176
1M-4	172	0	140	0	740	182
1M-5	172	0	1154	874	721	184
1M-6	141	0	531	312	709	173
1M-7	218	0	3760	3432	700	177
1M-8	265	0	17145	15896	835	153
2M-8	281	0	40061	36301	822	149
3M-8	234	0	60466	61776	744	148

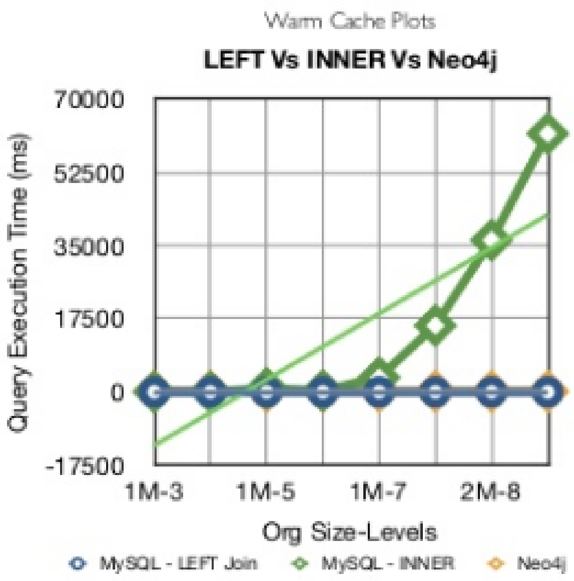


Figure 9.11.: neo4j sql comparison

(Dalal, 2014, slide 19).

As shown in the graph, Neo4j's full potential unleashes when many join-operations are executed as the query execution time stays constantly low, whereas the execution time for inner joins in MySQL rises exponentially. Another performance aspect is, that Neo4j in its second version can automatically index often used Nodes and gets an additional speed advantage at run time.

Overall Neo4j is quite unique, but has high potential in the sector of relational databases. It is useful to perform complex relational data queries and can be used for many different business cases. It is still the most popular graph database and some weaknesses can be handled by using Neo4j with another NoSQL database together.

Part VI.

Conclusion

This work contains a comparison of several popular NoSQL databases. The easiest way to choose a database solution on a given use case is to compare its needs to the CAP theorem. But not every database solution fits exactly into the CAP theorem. Each of them has its own capabilities, advantages and drawbacks.

The strenghts of column oriented databases clearly are on the fast reading and writing operations, when only accessing one single column, because not every line has to be read completely, but only the specified column. Also, no null values have to be inserted, which saves a lot of space. Aditionally, column oriented databases are easily scalable and distributed and good for saving massive amounts of data. The weakness is about transactions, those are handled significantly slower then e.g. on relational databases.

Key Value is an approach perfectly suited for uses cases where own keys are generated. The main difference to other NoSQL databases is that data is only accesible over the saved key. This aggravates the querieng of data appart from the key. An avantage of Key Value databases is the super flexible data model. Data sets could be saved without any type specification to a key.

Document-based nosql databases provides multiple advantages. With a single query, they are able to access huge amout of data fast. Especially compared to SQL it offers easier implementation because it stores data schema-less and enables flexibility. Over and above, document-based databases can be customized to meet certain requirements for the cap therorem like partitioning and sharding even over multiple servers through clustering. Besides to the advantages, document-based databases do have drawbacks such as the slow attribution aggregation or slow updates. In addition, all documents are stored unsorted and are completly loaded into the ram. Data is stored denormalized which causes high memory usage.

Graph databases like Neo4j take the opportunity to manage tightly related data, such as social networks or real-time recommendations. The similarity in the data structure to document based databases keeps the freedom in schema and the possibility to integrate and sync Neo4j with them to achieve the advantages of both types. On the downside the deep connected data disables the ability to provide partition tolerance. In addition the complexity of the data structure prevents high performance while read/write operations of non-related documents, but satisfies in querying relationships.

This Paper provides an insight into popular NoSQL databases and is a good start for beginners to learn the basic concepts and principles of NoSQL databases.

References

(n.d.).

Abadi, D. (2010). *Distinguishing two major types of column-stores.* Retrieved 2017-03-21, from http://dbmsmusings.blogspot.de/2010/03/distinguishing-two-major-types-of_29.html/

Achari, S. (2015). *Hadoop essentials.* Packt Publishing. Retrieved from https://www.safaribooksonline.com/

Amazon web services. (2017). *Apache hbase in amazon emr.* Retrieved 2017-03-21, from https://aws.amazon.com/de/emr/details/hbase/

Anderson, J. C., Lehnardt, J., & Slater, N. (2010). *Couchdb: The definitive guide.* O'Reilly Media.

Anderson, Jan Lehnardt, Noah Slater. (2010). *CouchDB: The Definitive Guide.* O'Reilly Media. Retrieved 2017-03-10, from http://guide.couchdb.org/draft/documents.html

Apache. (2014). *1.1. Technical Overview — Apache CouchDB 1.7.0 Documentation.* Retrieved 2017-03-10, from http://docs.couchdb.org/en/2.0.0/intro/overview.html

Apache Software Foundation. (n.d.). *15.1.1. version 2.0.0.* Retrieved from http://docs.couchdb.org/en/stable/whatsnew/2.0.html

Apache Software Foundation. (2013a). *Couchdb wiki: Setting_up_an_admin_account.* Retrieved from https://wiki.apache.org/couchdb/Setting_up_an_Admin_account

Apache Software Foundation. (2013b). *Security features overview: Authorization.* Retrieved from https://wiki.apache.org/couchdb/Security_Features_Overview

Apache Software Foundation. (2016a). *Documentation.* retrieved on 01.04.2017 from http://cassandra.apache.org/doc/latest/.

Apache Software Foundation. (2016b). *Dynamo.* retrieved on 01.04.2017 from http://cassandra.apache.org/doc/latest/architecture/dynamo.html.

Apache Software Foundation. (2017a). *10.1. api basics — apache couchdb 2.0 documentation.* Retrieved 13.03.2017, from http://docs.couchdb.org/en/2.0.0/api/basics.html

Apache Software Foundation. (2017b). *10.1. api basics — apache couchdb 2.0 documentation.* Retrieved 13.03.2017, from http://docs.couchdb.org/en/2.0.0/api/basics.html#json-basics

Apache Software Foundation. (2017c). *10.2.1. / — apache couchdb 2.0 documentation.* Retrieved 13.03.2017, from http://docs.couchdb.org/en/2.0.0/api/server/common.html

Apache Software Foundation. (2017d). *10.3. databases — apache couchdb 2.0 documentation.* Retrieved 13.03.2017, from http://docs.couchdb.org/en/2.0.0/api/database/common.html

Apache Software Foundation. (2017e). *10.4. documents — apache couchdb 2.0 documentation.* Retrieved 13.03.2017, from http://docs.couchdb.org/en/2.0.0/api/document/common.html

Apache Software Foundation. (2017f). *1.5. the core api — apache couchdb 2.0 documentation.* Retrieved 13.03.2017, from http://docs.couchdb.org/en/2.0.0/intro/api.html#replication

Banker, K. (2016). *MongoDB in action : covers MongoDB version 3.0.* Retrieved from https://www.safaribooksonline.com/library/view/mongodb-in-action/9781617291609/

Basho. (01.04.17a). *Companies using riak.* Bellevue, Washington. Retrieved 01.04.17, from http://basho.com/about/customers/

Basho. (01.04.17b). *Installation of riak on ubuntu.* Bellevue, Washington. Retrieved 01.04.17, from http://docs.basho.com/riak/kv/2.2.2/setup/installing/debian-ubuntu/

Basho. (01.04.17c). *Riak use cases.* Bellevue, Washington. Retrieved 01.04.2017, from http://basho.com/use-cases/

Basho. (01.04.2017). *Riak kv.* Bellevue, Washington. Retrieved 01.04.2017, from http://basho.com/products/riak-kv/

Basho. (06.04.2017). *Riak kv documentation.* Bellevue, Washington. Retrieved 06.04.2017, from http://docs.basho.com/riak/kv/2.2.3/

Bertozzi, M. (2012). *Apache hbase i/o - hfile [web log post].* Retrieved 2017-03-21, from http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/

Boncz, P. A. (1999). Database architecture optimized for the new bottleneck: Memory access.

Brown, M. (2012). *Getting started with couchdb.* Sebastopol: O'Reilly Media Inc. Retrieved from http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10759088

Brück, F. (2015). *Dokumentenorientierte datenbanken [wirtschaftsinformatik wiki - kewee].* Retrieved 30.03.2017, from http://wi-wiki.de/doku.php?id=bigdata:dokumentdb

Bruno Pedro. (2013). *Which startups use couchdb?* Retrieved 06.04.2017, from https://www.quora.com/Which-startups-use-CouchDB#

Bryan ssm. (06.04.2017). *Java management extensions.* San Francisco, California. Retrieved 06.04.2017, from https://en.wikipedia.org/wiki/Java_Management_Extensions

bsonspec.org. (2017). *Specification version 1.1.* (Retrieved March 23, 2017, from http://bsonspec.org/spec.html;)

Cap theorem. (n.d.). Retrieved from http://cs.uoi.gr/~apappas/projects/Raft&Rethinkdb/img/cap.jpg

CNCF. (2017). *Cncf purchases rethinkdb source code and contributes it to the linux foundation under the apache license.* retrieved on 10.04.2017 from https://www.cncf.io/blog/2017/02/06/cncf-purchases-rethinkdb-source-code-contributes-linux-foundation-apache-license/.

Cocomore. (2017). *Couchdb – einfache, aber mächtige datenbank für webanwendungen.* Retrieved 30.03.2017, from https://www.cocomore.de/leistungen/software/couchdb-einfache-aber-maechtige-datenbank-fuer-webanwendungen

Cohen, J. (2013). *Cohen: Using redis as a secondary index for mysql.* Retrieved 2017-03-31, from http://code.flickr.net/2013/03/26/using-redis-as-a-secondary-index-for-mysql/#

Dalal, D. (2014). *Neo4j mysql ms-sql comparison.* Retrieved April, 9, 2017, from https://de.slideshare.net/DhavalDalal/neo4j-my-sqlmssqlcomparisonfinal.

Das, V. (2015). *Learning redis.* Packt Publishing. Retrieved from https://books.google.de/books?id=ONODCgAAQBAJ

DataStax, Inc. (2016a). *Allow filtering explained.* retrieved on 01.04.2017 from https://www.datastax.com/dev/blog/allow-filtering-explained-2.

DataStax, Inc. (2016b). *Data replication.* retrieved on 01.04.2017 from https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureDataDistributeReplication_c.html.

DataStax, Inc. (2016c). *Introduction to cassandra query language.* retrieved on 01.04.2017 from https://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html.

Dede, E. (2013). Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. *Science Cloud 13*.

DeWitt, D. J. (1991). The Wisconsin Benchmark: Past, Present, and Future. *Computer Sciences Department*.

Edge, M. (2015). *Architectureoverview.* retrieved on 04.04.2017 from https://wiki.apache.org/cassandra/ArchitectureOverview.

Edward, S. G., & Sabharwal, N. (2015). *Practical MongoDB : architecting, developing, and administering MongoDB.* Retrieved from https://www.safaribooksonline.com/library/view/practical-mongodb-architecting/9781484206478/

Emin Gun Sirer. (2012). *What are the advantages and disadvantages of using MongoDB vs CouchDB vs Cassandra vs Redis? - Quora.* Retrieved 2017-03-23, from https://www.quora.com/What-are-the-advantages-and-disadvantages-of-using-MongoDB-vs-CouchDB-vs-Cassandra-vs-Redis

Erlang - Wikipedia. (22.03.2017). *Erlang (programming language).* Retrieved from https://en.wikipedia.org/wiki/Erlang_(programming_language)

FH Koeln. (2013). *Datenbanken / dokumentenorientierte datenbank | datenbanken online lexikon.* Retrieved 30.03.2017, from http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/DokumentenorientierteDatenbank

FH Köln, C. G. (01.04.17). *Datenbanken online lexikon - riak.* Gummersbach. Retrieved 01.04.17, from http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Riak/

Finley, K. (2014). *Out in the open: The abandoned facebook tech that now helps power apple.* retrieved on 01.04.2017 from https://www.wired.com/2014/08/datastax/.

George, L. (2011). *Hbase: The definitive guide.* Sebastopol, CA: O'Reilly Media.

Github. (2009). *Github: How we made github fast.* Retrieved 2017-03-31, from https://github.com/blog/530-how-we-made-github-fast

Google Developers. (06.04.2017). *Protocol buffers.* Mountain View, California. Retrieved 06.04.2017, from https://developers.google.com/protocol-buffers/

Greiner, R. (2014). *Cap theorem: Revisited.* Retrieved 2017-04-01, from http://robertgreiner.com/2014/08/cap-theorem-revisited/

Gupta, S. (2015). *Neo4j essentials.* Retrieved from https://books.google.de/books?id=WJ7NBgAAQBAJ.

Haber, I. (2016). *Haber: Why redis beats memcached for caching.* Retrieved 2017-03-31, from http://www.infoworld.com/article/3063161/application-development/why-redis-beats-memcached-for-caching.html

Harizopoulos, S., Abadi, D., & Boncz, P. (2009). *Column-oriented database systems.* Retrieved 2017-03-21, from http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf

Hasker, C. (2014). *What on earth are people using cassandra for anyway?* retrieved on 01.04.2017 from https://www.datastax.com/2014/06/what-are-people-using-cassandra-for.

Hortonworks Community. (2017). *What hbase does.* Retrieved 2017-03-21, from https://de.hortonworks.com/apache/hbase/

Hunger, B. R. . L. W., M. (2016). *The definitive guide to graph databases for the rdbms developer.* Retrieved April, 10. 2017, from http://info.neotechnology.com/rs/773-GON-065/images/Definitive-Guide-Graph-Databases-for-RDBMS-Developer.pdf.

Hunger, M. (2013). *Data modeling with neo4j.* Retrieved from the SlideShare website: https://de.slideshare.net/neo4j/data-modeling-with-neo4j.

Inc., M. (2016). *The mongodb 3.4 manual.* (Retrieved March 23, 2017, from https://docs.mongodb.com/manual/;)

jepsen.io. (2016). *Rethinkdb 2.1.5.* retrieved on 11.04.2017 from https://jepsen.io/analyses/rethinkdb-2-1-5.

Jochen Schnelle. (2012). *Sicherheit, round-trips.* Retrieved 2017-04-02, from http://www.pro-linux.de/artikel/2/1590/10,sicherheit.html

Juravich, T. (2012). *CouchDB and PHP Web Development Beginner ' s Guide.* Packt Publishing. Retrieved from https://www.safaribooksonline.com/library/view/couchdb-and-php/9781849513586/

juristat.com. (11.04.2017). *A comparison of mongodb and rethinkdb with patent data.* retrieved on 10.04.2017 from https://juristat.com/blog/a-comparison-of-mongodb-and-rethinkdb-with-patent-data.

Key-value database explained. (n.d.). Retrieved 2017-04-01, from http://basho.com/resources/key-value-databases/

Kincaid, J. (2009). *Yc-funded rethinkdb: A mysql storage engine built from the ground up for solid state drives.* retrieved on 10.04.2017 from https://techcrunch.com/2009/07/28/yc-funded-rethinkdb-a-mysql-storage-engine-built-from-the-ground-up-for-ssds/.

Kingsbury, L. (2013). *Jepsen: Redis.* Retrieved 2017-04-02, from https://aphyr.com/posts/283-jepsen-redis

KLINT FINLEY. (2010). *Why large hadron collider scientists are using couchdb.* Retrieved 07.04.2017, from http://readwrite.com/2010/08/26/lhc-couchdb/

L8 ManeValidus. (06.04.2017). *Simple network management protocol.* San Francisco, California. Retrieved 06.04.2017, from https://de.wikipedia.org/wiki/Simple_Network_Management_Protocol

Lee, S. (2009). W. . Advances in flash memory SSD technology for enterprise database applications. *SIGMOD 09*.

Li, Y. S., & IEEE. (2013). A performance comparison of SQL and NoSQL databases. *Rim Conference*.

Mahler, D. (2014). *Graphendatenbank: Flexible datenabfragen mit neo4j.* Retrieved April 9, 2017, from https://www.heise.de/developer/artikel/Graphendatenbank-Flexible-Datenabfragen-mit-Neo4j-2176439.html.

Marcel Wolf. (2016). *Elixir, phoenix und couchdb – eine einführung: Couchdb.* Retrieved from https://blog.codecentric.de/2016/01/elixir-phoenix-couchdb-eine-einfuehrung/

Mehra, A. (2015). *Introduction to apache cassandra's architecture.* retrieved on 01.04.2017 from https://dzone.com/articles/introduction-apache-cassandras.

Membrey, P., Hows, D., & Plugge, E. (2014). *Mongodb basics.* Apress.

Messinger, L. (2013). *Better explaining the cap theorem.* Retrieved 2017-04-01, from https://dzone.com/articles/better-explaining-cap-theorem

MongoDB. (2016). Unlocking Operational Intelligence from the Data Lake. (June). Retrieved from https://webassets.mongodb.com/mongodb{_}operational{_}data{_}lake.pdf?{_}ga=1.55917355.1638928680.1491468650https://www.mongodb.com/hadoop-and-mongodb

MongoDB Inc., a. M. W. P. (2013a). *Big Data: Examples and Guidelines for the Enterprise Decision Maker* (Tech. Rep.). Retrieved from http://s3.amazonaws.com/info-mongodb-com/10gen{_}Big{_}Data{_}White{_}Paper.pdf

MongoDB Inc., a. M. W. P. (2013b). MongoDB Architecture Guide. (August). Retrieved from https://webassets.mongodb.com/{_}com{_}assets/collateral/MongoDB{_}Architecture{_}Guide.pdf?{_}ga=1.253106510.679038577.1489829792

Morony Josh. (2017). *CouchDB, PouchDB, and Ionic 2: An Introduction | josh-morony - Build Mobile Apps with HTML5.* Retrieved 2017-03-23, from https://www.joshmorony.com/couchdb-pouchdb-and-ionic-2-an-introduction/

Mscdex. (n.d.). *Node.JS MySQL client library benchmarks* (Vol. 2017). Retrieved 23.03.2017, from http://mscdex.github.io/node-mysql-benchmarks/

Neo Technology, Inc. (2017a). *Cypher query language developer guides & tutorials.* Retrieved April 9, 2017, from https://neo4j.com/developer/cypher.

Neo Technology, Inc. (2017b). *For relational database developers: A sql to cypher guide.* Retrieved April 9, 2017, from https://neo4j.com/developer/guide-sql-to-cypher.

Neo Technology, Inc. (2017c). *Graph data modeling guidelines.* Retrieved March 29, 2017, from https://neo4j.com/developer/guide-data-modeling.

Neo Technology, Inc. (2017d). *Graph db vs rdbms.* Retrieved April 9, 2017, from https://neo4j.com/developer/graph-db-vs-rdbms/.

Neo Technology, Inc. (2017e). *Intro to cypher.* Retrieved March 29, 2017, from https://neo4j.com/developer/cypher-query-language.

Neo Technology, Inc. (2017f). *Neo4j and mongodb.* Retrieved April, 9, 2017, from https://neo4j.com/developer/mongodb/.

Neo Technology, Inc. (2017g). *Neo4j licensing.* Retrieved April 9, 2017, from https://neo4j.com/licensing/.

Neo Technology, Inc. (2017h). *Neo4j: The world's leading graph database.* Retrieved April 9, 2017, from https://neo4j.com/product/.

Neo4j Staff. (2011). Intro to graph databases. Retrieved April, 9, 2017, from https://neo4j.com/blog/recap-intro-to-graph-databases-webinar-series-1/.

Oliver, A. (2014). *Get to know cassandra, the nosql maverick.* retrieved on 01.04.2017 from http://www.javaworld.com/article/2365720/big-data/get

-to-know-cassandra-the-nosql-maverick.html.

O'Neil, P. (1997). Database Performance Measurement. *CRC Handbook for Computer Science and Engineering.*

Parthasarathy, V. (2013). *Learning cassandra for administrators.* Packt Publishing Limited.

Plugge, E., Hows, D., Membrey, P., & Hawkins, T. (2015). *The definitive guide to mongodb: A complete guide to dealing with big data using mongodb* (3rd ed.). Berkely, CA, USA: Apress.

Redis Labs, Inc. (2017a). *An introduction to redis data types and abstractions.* Retrieved 2017-04-02, from https://redis.io/topics/data-types-intro

Redis Labs, Inc. (2017b). *Redislabs: Expire key seconds.* Retrieved 2017-03-25, from https://redis.io/commands/expire

Redis Labs, Inc. (2017c). *Redislabs: Pub/sub.* Retrieved 2017-03-26, from https://redis.io/topics/pubsub

Redis Labs, Inc. (2017d). *Redislabs: Replication.* Retrieved 2017-03-29, from https://redis.io/topics/replication

Redis Labs, Inc. (2017e). *Redislabs: Solutions.* Retrieved 2017-03-26, from https://redislabs.com/solutions/use-cases/

Redis Labs, Inc. (2017f). *Redislabs: Transaction.* Retrieved 2017-03-27, from https://redis.io/topics/transactions

Redis Labs, Inc. (2017g). *Redislabs: Using redis as a lru cache.* Retrieved 2017-03-26, from https://redis.io/topics/lru-cache

Redis Labs, Inc. (2017h). *Redislabs: Who's using redis.* Retrieved 2017-03-31, from https://redis.io/topics/whos-using-redis

Redis Labs, Inc. (2017i). *Redis security.* Retrieved 2017-04-02, from https://redis.io/topics/security

Robinson, W. J. . E. E., I. (2013). *Graph databases.* Sebastopol, CA: O'Reilly Media.

Rodriguez, P., M. A. & Neubauer. (2010). *Constructions from dots and lines.* Retrieved from the Cornell University website: https://arxiv.org/abs/1006.2361.

Rouse, M. (2005). *B-tree.* retrieved on 10.04.2017 from http://searchsqlserver.techtarget.com/definition/B-tree.

Rouse, M. (2006). *Acid (atomicity, consistency, isolation, and durability).* retrieved on 10.04.2017 from http://searchsqlserver.techtarget.com/definition/ACID.

Rouse, M. (2016). *Definition: graph database.* Retrieved April 5, 2017, from http://whatis.techtarget.com/definition/graph-database.

Sabo, A. (2016). *Sabo: Queuing tasks with redis.* Retrieved 2017-03-31, from https://blog.logentries.com/2016/05/queuing-tasks-with-redis/

Scalability, H. (2011). *High scalability: Stack overflow architecture update.* Retrieved 2017-03-31, from http://highscalability.com/blog/2011/3/3/stack-overflow

-architecture-update-now-at-95-million-page-vi.html

Scalability, H. (2014). *High scalability: How twitter uses redis to scale.* Retrieved 2017-03-31, from http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html

Scheliga, M. (2010). *Couchdb: Kurz & gut.* Köln: O'Reilly.

Seeger, M. (2009). *Key-value stores: a practical overview* [paper]. Retrieved 2017-04-02, from http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_Stores.pdf

Sheehy, J. (2013). *Quora: What is redis in the context of the cap theorem.* Retrieved 2017-04-02, from https://www.quora.com/What-is-Redis-in-the-context-of-the-CAP-Theorem

Shriparv, S. (2014). *Learning hbase.* Packt Publishing. Retrieved from https://www.safaribooksonline.com

Socialpoint. (2013). *Socialpoint: Using redis to build your game leaderboard.* Retrieved 2017-03-31, from http://www.socialpoint.es/blog/using-redis-to-build-your-game-leaderboard/

Solid IT GmbH. (2017). *Db-engines ranking von graph dbms.* Retrieved April 8, 2017, from https://db-engines.com/de/ranking/graph+dbms.

StackOverflow. (2015). *When not to use cassandra.* retrieved on 01.04.2017 from http://stackoverflow.com/questions/2634955/when-not-to-use-cassandra.

Techstacks. (2017). *Techstacks: Redis.* Retrieved 2017-03-31, from http://techstacks.io/tech/redis

TH Köln, Campus Gummersbach. (2011). *Redis.* Retrieved 2017-03-30, from http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Redis

Thangarajan, N. (2015). *Mongodb vs. rethinkdb.* retrieved on 11.04.2017 from http://ntvita.com/2015/06/04/mongo-vs-rethink.html.

The Apache Software Foundation. (06.04.2017). *Apache solr.* Forest Hill, Maryland. Retrieved 06.04.2017, from http://lucene.apache.org/solr/

The Apache Software Foundation - Apache HBase Team. (2017). *Apache hbase reference guide.* Retrieved 2017-03-21, from https://hbase.apache.org/book.html#datamodel

The Linux Foundation. (n.d.-a). *Introduction to reql.* retrieved on 11.04.2017 from https://rethinkdb.com/docs/introduction-to-reql/.

The Linux Foundation. (n.d.-b). Limitations in RethinkDB - RethinkDB. Retrieved 2017-04-10, from https://rethinkdb.com/limitations/

The Linux Foundation. (n.d.-c). *Rethinkdb 2.1.5 performance & scaling report.* retrieved on 11.04.2017 from https://rethinkdb.com/docs/2-1-5-performance-report/.

The Linux Foundation. (n.d.-d). *Rethinkdb create table.* retrieved on 11.04.2017 from https://rethinkdb.com/api/javascript/table_create/.

The Linux Foundation. (2017a). CAP Theorem. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#cap-theorem

The Linux Foundation. (2017b). Changefeeds in RethinkDB - RethinkDB. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/changefeeds/javascript/

The Linux Foundation. (2017c). How is data stored on disk? , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#data-storage

The Linux Foundation. (2017d). Query Execution. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#query-execution

The Linux Foundation. (2017e). ReQL data types - RethinkDB. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/data-types/

The Linux Foundation. (2017f). Sharding and replication. , *2017*. Retrieved 2017-04-10, from https://rethinkdb.com/docs/architecture/#sharding-and-replication

Tiwari, A. (2015). *Why cassandra is an excellent choice for real time aalytics workloads.* retrieved on 01.04.2017 from http://blogs.shephertz.com/2015/04/22/why-cassandra-excellent-choice-for-realtime-analytics-workload/.

Tpc.org. (n.d.). *About the TPC* (Vol. 2017). Retrieved 2017-03-21, from http://www.tpc.org/information/about/abouttpc.asp

Tutorialspoint. (n.d.). *CouchDB Creating a Document.* Retrieved 2017-03-10, from https://www.tutorialspoint.com/couchdb/couchdb_creating_a_document.htm

tutorialspoint.com. (2017). *Redis - sets.* Retrieved 2017-04-02, from https://www.tutorialspoint.com/redis/redis_sets.htm

Vohra, D. (2016). *Apache hbase primer.* Apress. Retrieved from https://www.safaribooksonline.com/

W3C. (2014). *Resource description framework (rdf).* Retrieved April 9, 2017, from https://www.w3.org/RDF/.

Wadkar, S., Siddalingaiah, M., & Venner, J. (2014). Pro Apache Hadoop. , 405. Retrieved from https://www.safaribooksonline.com/library/view/pro-apache-hadoop/9781430248644/ doi: 10.1007/978-1-4302-4864-4

Wang, M. (2001). Main Memory Databases.

Weil, K. (2017). Rainbird: Realtime Analytics at Twitter (Strata 2011). Retrieved 21.03.2017, from https://www.slideshare.net/kevinweil?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

Wikipedia. (2016a). *Column family.* retrieved on 01.04.2017 from https://en.wikipedia.org/wiki/Column_family.

Wikipedia. (2016b). *Keyspace (distributed data store).* retrieved on 01.01.2017 from https://en.wikipedia.org/wiki/Keyspace_(distributed_data_store).

Wilson, J. (2008). *Understanding hbase and bigtable [web log entry].* Retrieved 2017-03-21, from `https://dzone.com/articles/understanding-hbase-and-bigtab`