NoSQL-Databases

TINF20D

April 20, 2023

Contents

1 Introduction **4**

2 Cassandra **6**
2.1 Wide Column Databases . . . 6
2.2 History of Cassandra . . . 6
2.3 Implementation and Experience . . . 7
2.4 Classification of Cassandra in CAP-Theorem . . . 10
2.5 Conclusion . . . 10
Sources . . . 11

3 Redis **12**
3.1 Key-Value . . . 12
3.2 History . . . 13
3.3 Characteristic . . . 13
3.4 CAP-Theorem . . . 14
3.5 Database Design . . . 15
3.6 Redis in Practice . . . 19
3.7 Use Cases . . . 20
3.8 Other Key-Value Stores . . . 20
3.9 Conclusion . . . 21
Sources . . . 21

4 Hazelcast **24**
4.1 Key-Value-Stores . . . 24
4.2 Hazelcast in theory . . . 26
4.3 Hazelcast in Practice . . . 27
4.4 Conclusion and Recommendations . . . 30
Sources . . . 30

5 MongoDB **33**
5.1 Introduction to Document DBs . . . 33
5.2 Comparing document DB and relational DB . . . 34
5.3 Design and Structure of MongoDB . . . 34
5.4 API . . . 35
5.5 Installation . . . 36
5.6 Testing MongoDB . . . 37
5.7 Advantages and Disadvantages . . . 37
5.8 Position according to CAP . . . 38
5.9 Recommendation and Conclusion . . . 38

Sources . . . 39

6 Cloud Firestore **42**
6.1 History . . . 42
6.2 Architecture . . . 43
6.3 CAP-Theorem . . . 45
6.4 Installation and API access . . . 47
6.5 Advantages and Disadvantages . . . 49
6.6 Conclusion and Outlook . . . 50
Sources . . . 50

7 Neo4j **52**
7.1 What are Graph based Databases? . . . 52
7.2 Brief History of Neo4j . . . 55
7.3 Pros and Cons of graph based databases . . . 56
7.4 Neo4j Features . . . 57
7.5 CAP Theorem . . . 58
7.6 Using Neo4j . . . 59
7.7 Conclusion . . . 61
7.8 Recommendations . . . 62
Sources . . . 62

8 Conclusion **64**
8.1 Outlook . . . 64
8.2 Closing Remarks . . . 65

Sources - Introduction and Conclusion **66**

1 Introduction

Can relational databases (DBs) keep up with the ever-growing demands of modern big data architectures? Relational databases are the current state of the art when it comes to operational data. In the current years, the big data trends forced to reevaluate database structures. Therefore, NoSQL databases have gained traction in recent years. This book will deal with different NoSQL databases and analyze them in detail. It is a result of the course "Current database technologies" at the Corporate State University Stuttgart. The different chapters about the databases were written by small groups of students with limited interaction. Each chapter can be looked at as a small article or paper. It will explore Cassandra, Redis, Hazelcast, MongoDB, Cloud Firestore, and Neo4j. In general, the term NoSQL stands for "not only SQL". It reflects databases with alternative models outside the standard relational world.

SQL databases have been covered extensively in previous lectures. Principles such as ACID or BASE have been explained in more detail, and the limitations of SQL databases have been discussed. Today, more and more applications are based on NoSQL databases (solid IT GmbH, 2023) and there are many different topologies of NoSQL databases, such as wide column, document-oriented, key-value or graph- based databases. As a result, these database topologies need to be discussed and compared with conventional SQL and other NoSQL topologies. A theoretical basis for this comparison would be the CAP theorem introduced by Brewer (2000), which deals with the following three different main characteristics of databases: Consistency, Availability and Partition Tolerance. The theorem claims that only two of these features can be covered by one system.

As a result of our motivation and the context of this work, the following research question was formulated: "What are the advantages and disadvantages of current (open source) NoSQL DBs with a theoretical foundation of the CAP theorem (Brewer, 2000) and task/application-oriented recommendations?"

Exploring databases through sample projects can provide insights into the strengths and weaknesses of different databases and technologies. However, this approach has some limitations: We were only able to look at a limited number of database technologies. The sample of databases and projects chosen may be biased towards certain types of applications, and the analysis may be limited in depth due to time and resource constraints. In addition, the landscape of database technologies is constantly evolving. This may limit the relevance of the findings to current projects.

In this paper, we will explore six different NoSQL databases, namely Cassandra, Redis, Hazelcast, MongoDB, The Cloud Firestore, and Neo4j. We will examine the unique features and benefits of each database, and highlight how they can be leveraged to meet the needs of various applications. Additionally, we will explore the trade-offs that come

with different consistency, availability, and partition tolerance levels, as defined by the CAP theorem, and how each database balances these trade-offs. Through practical examples, readers will gain a clear understanding of the advantages and limitations of each NoSQL database and how they can be applied to real-world scenarios. By the end of this paper, readers will have a deeper understanding of the unique capabilities of each of these NoSQL databases, and be able to make informed decisions when choosing the right database for their specific needs.

2. Cassandra

by Simon Morgenstern, Theo Krinitz

2.1 Wide Column Databases

Wide column stores, also known as column-family stores, are a type of NoSQL database. This class of database system is designed to handle large amounts of semi-structured or unstructured data. They are often used for big data applications that require high scalability and flexibility. ("Wide Column Stores - DB-Engines Encyclopedia", 2023)

In a wide column store, data is stored in column families, which are groups of related columns that are stored together. Each column family can contain an unlimited number of columns, and each column can contain multiple values. This allows for a highly flexible data model, where each row can have a different set of columns. ("How Cassandra reads and writes data | Apache Cassandra 3.0", 2022)

Queries in a wide column store are typically performed using a key-value interface, where data is accessed using a unique key. This allows for fast and efficient data retrieval, as only the requested data needs to be accessed ("Wide Column Stores - DB-Engines Encyclopedia", 2023). Wide column stores also support secondary indexes, which allow for more complex queries to be performed. They come with some performance trade-offs though which is why they are only used if necessary. ("Guarantees | Apache Cassandra Documentation", 2023; "Secondary Indexes | Apache Cassandra Documentation", 2023)

Wide column stores use a distributed architecture, where data is partitioned across multiple nodes in a cluster. This allows for high scalability and fault-tolerance, as data can be replicated across multiple nodes to ensure availability in the event of a node failure. ("Dynamo | Apache Cassandra Documentation", 2023)

Google BigTable is considered the origin of this class of database. (Chang et al., 2008)

2.2 History of Cassandra

Apache Cassandra is a distributed NoSQL database management system that is designed to handle large amounts of data across multiple servers while providing high availability and fault tolerance. The project was originally developed at Facebook in 2008 to power its inbox search feature. In 2009, it was made open-source under the Apache License and became an Apache Incubator project (apache, 2023; "Overview | Apache Cassandra Documentation", 2023). Cassandra was influenced by Amazon Dynamo's design principles,

including its decentralized architecture and eventual consistency model. ("Facebook Releases Cassandra as Open Source – Perspectives", 2023)

Cassandra was designed to handle the massive amounts of data that Facebook was generating at the time, which was well beyond the capabilities of traditional relational databases. It was built to be highly scalable and fault-tolerant, with no single point of failure. This was achieved through the use of a distributed architecture. ("Overview | Apache Cassandra Documentation", 2023)

In 2010, Cassandra graduated from the Apache Incubator and became a top-level Apache project. Since then, it has continued to evolve and gain popularity among developers and organizations looking for a highly scalable and resilient NoSQL database. ("Cassandra is an Apache top level project", 2023)

Today, Cassandra is used by many large organizations, including Netflix, Apple and eBay, to power their data-intensive applications. Its popularity can be attributed to its scalability, high availability, and ability to handle large amounts of data with low latency. Cassandra is also highly customizable and flexible, making it a popular choice for many use cases, including real-time analytics, content management, and IoT applications. (Cassandra, 2014; "DB-Engines Ranking", 2023)

2.3 Implementation and Experience

To get a better understanding of the Cassandra database, we integrated the database into a former university project.

2.3.1 Installation

Cassandra can be installed via the official 'Cassandra' docker image ("Apache Cassandra | Apache Cassandra Documentation", 2/13/2023), from Debian packages or from RPM packages ("Apache Cassandra | Apache Cassandra Documentation", 2/14/2023).

In our case, Docker was the best option.

2.3.2 Integration

There are many projects that focus on building language specific drivers for the Cassandra database. Those languages include Java, NodeJS, Python, Ruby, C#, PHP, C++, Go and Rust ("Client drivers | Apache Cassandra Documentation", 2/13/2023).

2.3.3 Experience with a demo project

We took a project from the 'Webengineering 2' module. The project had an Angular frontend, a Express.JS backend and a SQLite (relational) database. We replaced the database with Cassandra and adjusted the backend to work with the new database.

Installing Cassandra

At first we created docker images for the frontend and the backend. Then we integrated the Cassandra image in our docker-compose file as shown in listing 2.1. To ensure data persistence we mounted a volume to the container.

```
version: '3'
services:

 Cassandra:
  image: Cassandra
  container_name: Cassandra
  hostname: Cassandra
  command: ["Cassandra", "-f"]
  ports:
   - "9042:9042"
  volumes:
   - ./data:/data
  restart: always
```

Listing 2.1: docker-compose.yml

Connecting to the database

Now that the database is up and running, we can connect to it using the 'Cassandra-driver' as shown in listing 2.2.

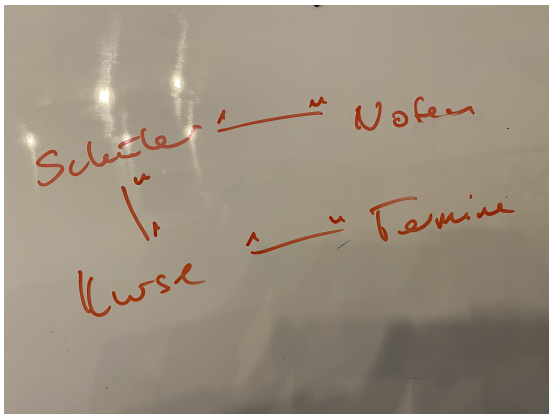
```
import { Client } from "Cassandra-driver";

export const Cassandra_db = new Client({
  contactPoints: ['Cassandra'], // hostname
  localDataCenter: 'datacenter1'
})
```

Listing 2.2: Cassandra-db.ts

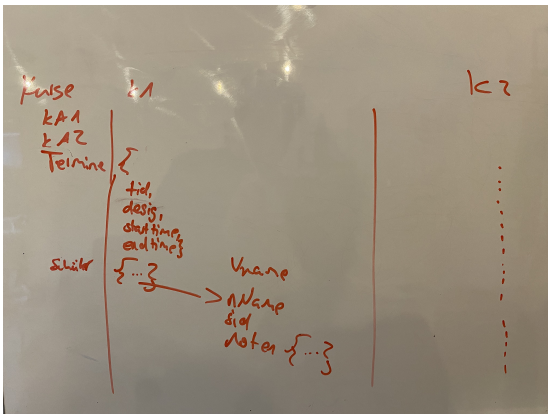
From that point we can use the Cassandra_db object to execute queries.

Designing the database

The design of a Cassandra database should start with the queries and not with the data model ("RDBMS Design | Apache Cassandra Documentation", 2/13/2023). For our example we took our existing model 2.1a and combined everything into one big table 2.1b.



(a) Relational Model



(b) New database model

Comparison with CRUD calls

We compared the calls that were made to the database of the original project with the calls to the Cassandra database.

Create: Inserting a new element into the two database systems is nearly the same. The only difference is that you have to create a primary key before inserting a new element into the Cassandra database. This is neccesary because Cassandra is a distributed system.

Read: Reading data was very different in our example. In the original project several queries were made to create one big object. In the Cassandra database we just have to make one query to get the big object directly. After receiving the object we than had to parse it into our model.

Update: It is important to know that Cassandra treats nested objects as so called frozen collections. Those collections are immutable and can just be changed by replacing them. That means that we can not just update a single property of a nested object in Cassandra.

Delete: Deleting an element in the relational database is more complex compared to Cassandra. To ensure referential integrity you have to delete all elements that reference to the element that should be deleted. In Cassandra you just need to delete the element itself.

2.4 Classification of Cassandra in CAP-Theorem

The CAP-theorem, also known as Brewer's theorem, is a principle in distributed computing that states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees (Brewer, 2000; Gilbert & Lynch, 2002):

- Consistency: Every read operation receives the most recent write or an error.
- Availability: Every request receives a response, without guarantee that it contains the most recent version of the information.
- Partition tolerance: The system continues to function despite arbitrary partitioning due to network failures.

In web-based applications, ensuring high availability is essential for providing uninterrupted service to users. Cassandra, therefore prioritizes Availability and Partition Tolerance from the CAP theorem guarantees. However, this choice comes at the cost of compromising on data Consistency to some extent. Cassandra's emphasis on availability and partition tolerance over strong consistency is intended to provide high performance and fault tolerance. Cassandra therefore is an AP-System. ("Guarantees | Apache Cassandra Documentation", 2023)

2.5 Conclusion

Apache Cassandra offers several key advantages that make it a popular choice for handling big data in a variety of industries. Its high scalability, fault tolerance, and flexible data model allow for efficient management of large amounts of unstructured or semi-structured data. Additionally, the ability to distribute data across multiple nodes improves performance and ensures low latency for read/write operations. Finally, the cost-effective nature of this open-source solution makes it an attractive option for organizations with big data needs. Overall, Apache Cassandra's unique features and benefits make it a powerful tool for managing and analyzing large-scale data sets. ("Dynamo | Apache Cassandra Documentation", 2023; "Guarantees | Apache Cassandra Documentation", 2023)

On the other hand, while Apache Cassandra offers many advantages for managing big data, it also has some downsides that users should be aware of. Firstly, Cassandra has limited support for complex transactions and relational queries, which can make it difficult for users who are used to working with traditional SQL databases. Additionally, the lack of native support for ACID transactions can lead to data inconsistency in certain scenarios. Furthermore, the learning curve for developers who are new to Cassandra can be steep, because Cassandra is build for query-first design. Finally, for small-scale applications with simple data models and low traffic volumes, Cassandra may not be the best fit. ("Guarantees | Apache Cassandra Documentation", 2023; "Secondary Indexes | Apache Cassandra Documentation", 2023)

It is important to weigh the benefits and drawbacks of Cassandra carefully before deciding if it is the right choice for a particular use case.

Sources

apache. (2023, March). cassandra [[Online; accessed 4. Mar. 2023]]. https://github.com/apache/cassandra/releases/tag/cassandra-4.1.0

Apache cassandra | apache cassandra documentation. (2/13/2023). https://cassandra.apache.org/_/quickstart.html

Apache cassandra | apache cassandra documentation. (2/14/2023). https://cassandra.apache.org/_/download.html

Brewer, E. A. (2000). Towards robust distributed systems. *PODC*, *7*(10.1145), 343477–343502.

Cassandra is an Apache top level project [[Online; accessed 4. Mar. 2023]]. (2023, March). https://www.mail-archive.com/cassandra-dev@incubator.apache.org/msg01518.html

Cassandra, A. (2014). Apache cassandra. *Website. Available online at http://planetcassandra.org/what-is-apache-cassandra*, *13*.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, *26*(2), 1–26.

Client drivers | apache cassandra documentation. (2/13/2023). https://cassandra.apache.org/doc/latest/cassandra/getting_started/drivers.html

DB-Engines Ranking [[Online; accessed 4. Mar. 2023]]. (2023, March). https://db-engines.com/en/ranking/wide+column+store

Dynamo | Apache Cassandra Documentation [[Online; accessed 13. Mar. 2023]]. (2023, February). https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html

Facebook Releases Cassandra as Open Source – Perspectives [[Online; accessed 4. Mar. 2023]]. (2023, March). https://perspectives.mvdirona.com/2008/07/facebook-releases-cassandra-as-open-source

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, *33*(2), 51–59.

Guarantees | Apache Cassandra Documentation [[Online; accessed 7. Mar. 2023]]. (2023, February). https://cassandra.apache.org/doc/4.1/cassandra/architecture/guarantees.html#what-is-cap

How Cassandra reads and writes data | Apache Cassandra 3.0 [[Online; accessed 10. Mar. 2023]]. (2022, February). https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlIntro.html#column-family-concepts

Overview | Apache Cassandra Documentation [[Online; accessed 4. Mar. 2023]]. (2023, February). https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html

Rdbms design | apache cassandra documentation. (2/13/2023). https://cassandra.apache.org/doc/latest/cassandra/data_modeling/data_modeling_rdbms.html

Secondary Indexes | Apache Cassandra Documentation [[Online; accessed 13. Mar. 2023]]. (2023, February). https://cassandra.apache.org/doc/4.1/cassandra/cql/indexes.html

Wide Column Stores - DB-Engines Encyclopedia [[Online; accessed 4. Mar. 2023]]. (2023, March). https://db-engines.com/en/article/Wide+Column+Stores

3. Redis

by Amos Groß, Tim Horlacher, Tim Kurfiss, Lukas Huida

Modern applications are running at an ever-increasing pace. Over time, users have come to expect razor-thin latencies while operating with ever-growing mounds of data. Traditional databases have almost all reached their physical limitations. Redis is part of a new generation of databases. It attempts to overcome these limitations and provide applications with the speed they require for data at scale.

3.1 Key-Value

Redis is a key-value database. Key-value databases, or key-value stores, are NoSQL databases that store data as key-value pairs. A key is a unique identifier used to retrieve and organize the associated value. The value can be anything from a simple datatype like a string to more complex data structures like a list or a JSON-document. The key-value store is a very simple data model and can be compared to a more commonly known dictionary or hash-map. (InfluxData Inc., 2022; MongoDB Inc., 2023; Han et al., 2011, p. 364)

Key	Vlaue
KeyString	String
KeyList	[A, B, C]
KeyHash	{ A: “String1”, B: “String2” }

Figure 3.1: Example of a key-value store.

Because of the simplicity of the data model, its implementations are mostly highly scalable and performant. The key-value store is commonly used for caching, session management and as a database for small applications. (InfluxData Inc., 2022; MongoDB Inc., 2023)

Outside from Redis, some of the most popular key-value stores are DynamoDB, CosmosDB and Memcached (solid IT gmbh, 2023a). This section will focus on Redis and give a brief overview of its history, characteristics, and use cases.

3.2 History

Redis was originally developed by Salvatore Sanfilippo (nickname: Antirez) to create a more scalable database for his start-up in Italy. The first version of Redis in 2009 was a tiny program written in *tlc* as a proof of concept. (Sanfilippo, 2017)

It was later translated to *C*. The project became open source and established itself as a popular database for in-memory caching. (Redis Ltd., 2009)

The company "Garantia Data" became one of its main sponsors and renamed itself "Redis Labs" while offering a more advanced enterprise version of redis (Redis Ltd., 2014). Today, the company owns the rights to the project and has renamed itself to "Redis" for better association (Ofer Bengal and Yiftach Shoolman, 2021). In 2020, Salvatore Sanfilippo, who continued to maintain the redis project, stepped down (Bengal, 2020).

3.3 Characteristic

"Redis is an open source (BSD licensed), in-memory data structure store used as a database, cache, message broker and streaming engine" (Redis Ltd., 2023l). Here are some of its main characteristics:

- In-memory: Redis is an in memory database, which means its data is primarily stored in the RAM (Random Access Memory). This makes it very fast for read and write operations, as well as scalability. However, this mode of operation is not persistent, which means that the data is lost after a shut-down. (Han et al., 2011, p. 365)
- Persistence: To prevent data loss, Redis provides two options for persisting data on disk: snapshots (RDB) and append-only files (AOF). By default, Redis creates snapshots of the database on disk. These can be triggered based on a predefined time period or number of write operations. While this is a good option for backups, it can result in small data losses in the time between snapshots. (Redis Ltd., 2023f)

  To mitigate this risk, Redis also offers the append-only file option, which must be enabled manually. The append-only file is a log file that contains all write operations that have been performed on the database. This log file can be used to restore the database to a previous state. The user can choose between those two options, use both, or none. (Redis Ltd., 2023f)
- Data-types: Redis supports a wide range of data types, including strings, lists, sets, sorted sets, hashes, bitmaps, hyperloglogs and geospatial indexes in its core functionalities. It can also be extended with modules to support additional data types. (Redis Ltd., 2023d)
- Scalability: For scalability, Redis offers two main options: sharding and replication. Replication is a way of creating a copy of the database on another server. This copy can be used for read operations, reducing the load on the master server. However,

the amount of data that can be stored does not scale, so it is not a good option for large databases. (Redis Ltd., 2023h)

In contrast, sharding, called *Redis Cluster*, distributes the data across multiple Redis instances or nodes. Each node is responsible for a subset of the data, allowing horizontal scaling. (Redis Ltd., 2023k)

Both options increase the availability, performance and redundancy of the database and can be used in combination. It must be determined which option is best for the use case. (Redis Ltd., 2023k)

- Pub/Sub: Redis offers a publish/subscribe messaging system. This allows clients to subscribe to channels and receive messages that are published to those channels. (Redis Ltd., 2023g)

3.4 CAP-Theorem

The CAP-Theorem deals with *Consistency*, *Availability* and *Partition-Tolerance*, where only two of the properties can be achieved in a distributed system (Brewer, 2000). Since Redis can run in a distributed System with multiple servers, this section will examine how the CAP-Theorem applies to redis.

Redis offers the option to use a cluster mode, where data is distributed via sharding. Different data points can partitioned into shards and made failure tolerant via replicas. With this feature *Partition-Tolerance* is given at any time. (Redis Ltd., 2023k)

Classifying Redis in terms of the CAP-Theorem therefore comes down to consistency and availability. Inconsistencies in a database can only occur when multiple writes to an identical data point occur on different servers at the same time. Because shards only accept writes on one node and shards don't duplicate their data, inconsistencies can't occur in regular operation. During a network fault, this changes. As an example we'll take a look at a system with three Redis instances. One of the instances is a primary node, while the others are replicas. When a network fault occurs, not all nodes can communicate with each other. In our case we will assume the primary node is cut off from both replicas. The two replicas assume the primary node is no longer operational and elect a new primary. The result is a system with two primaries, both accepting writes to the same data points. (Noonan, 2023)

In use cases like the one explained above, a database admin has to decide between availability and consistency. By configuring what happens during network faults, varying degrees of availability and consistency assurances can be set. There are two specific properties which can be configured; *min-replicas-to-write* and *min-replicas-max-lag*. The former defines the amount of minimum replicas a primary has to replicate its data to, for it to accept a write. If this were set to three in the example above, the system would remain consistent. However, it would be unavailable for the duration of the network fault.. The latter property defines the ping frequency in seconds of the replicas. Pings determine if nodes is marked as unavailable. The lower the ping, the higher the consistency, but also the more susceptible to faults it becomes. (Redis Ltd., 2023a, 2023b)

To summarize, a Redis cluster can be configured to allow for varying levels of divergence. With higher divergence, a cluster allows for availability, but sacrifices Consistency. Clusters configured in this manner are AP-Systems. With lower divergence, a cluster enforces consistency but sacrifices availability. Clusters configured in this manner are CP-Systems. (Redis Ltd., 2023b)

3.5 Database Design

Redis is designed to be a highly extensible database. At its core, Redis is a key-value store. Similarly to other key value stores, data entries in Redis consist out of keys and values. Optionally, Redis can attach expiration times to data entries (Redis Ltd., 2023e). Key expiration is especially useful for invalidating caches.

Redis further distinguishes itself from other key value stores through its wide range of special types and operations. These allow users of the database to efficiently interact with their data. This is why Redis is commonly referred to as a data structure store. (Redis Ltd., 2023l)

Using a vast module system, the database can be transformed to support a variety of other database models. For example RedisJSON and RedisGraph can be used to transform Redis into a document or graph database respectively. (Redis Ltd., 2022e)

3.5.1 Data Types

Keys for data entries are always strings. Strings are the most fundamental data type in Redis. They describe a sequence of arbitrary bytes that can be as long as 512 MB. This means that you could technically embed a JPEG as a key in Redis. Despite being an interesting capability, it is probably best to stay away from long keys as like JPEGs. Key lookups require a number of costly key-comparisons, which can be extremely taxing on performance for long keys. Strings are the default data type for values. Similarly to keys, string values can't be longer than 512 MB. (Redis Ltd., 2022a)

An ordered sequence of strings can be stored as a list. Lists are implemented using linked lists. This means that operations dealing with the head of lists happen in constant time ($O(1)$). Retrieving items from arbitrary locations in the list requires linear time ($O(n)$). Lists are thus best optimized to suit the needs of queues and stacks. This makes them ideal candidates for use cases such as message queuing systems. (Redis Ltd., 2023e)

Unordered collections of unique strings can be stored in sets. Sets allow for efficient tracking of unique items. Close to all actions for sets can be performed in constant time ($O(1)$). Sets of key-value pares can be represented with Hashes. Hashes are the Redis equivalent of dictionaries or HashMaps prevalent in most programming languages. They are especially useful for representing objects in Redis. Hashes have no size restrictions. (Redis Ltd., 2023e)

3.5.2 Special Use-Cases

Redis also offers a growing amount of other data types for special use cases. These include examples like streams or geospacial indexes (Redis Ltd., 2023e). If the available data types and operations don't suffice for a certain application, Redis can be extended using custom Lua scripts. These scripts execute atomically and can directly operate on the data present in Redis. This tight integration is very efficient. Developers can leverage Lua scripts to integrate parts of their logic into Redis directly and build more performant applications. (Redis Ltd., 2022d)

3.5.3 RESP Protocol

Redis operates in a client server model. Redis clients use a protocol called RESP (REdis Serialization Protocol) to communicate with the Redis server. The protocol is specifically designed for Redis and aims to be simple to implement, fast to parse and human-readable. (Redis Ltd., 2023i) RESP can serialize different data types like integers, strings, and arrays. There is also a specific type for errors.

RESP operates on a request-response basis. Requests are sent from the client to the Redis server as arrays of strings that represent the arguments of the command to execute. The server sends a response with a command-specific data type. The response can be an error, an integer, a string, or an array of strings. (Redis Ltd., 2023j)

3.5.4 Serialization

RESP uses a simple serialization format. The first byte of a RESP message determines the type of the data. The following bytes contain the actual data. The message is terminated by a special character sequence known as CRLF (Carriage Return Line Feed), which signals the end of the message. (Redis Ltd., 2023i, 2023j)

Simple Strings

Simple Strings are indicated by a plus sign as the first byte of the message. Followed by the string itself. It has to be noted, that simple strings are not binary safe. This means that they cannot contain a CR or LF character. Many commands in Redis return simple strings as a response. The most common example is the response "OK" to the command "SET". (Redis Ltd., 2023i)

Bulk Strings

Bulk Strings are indicated by a dollar sign followed by the length of the string in bytes and a CRLF. The string itself follows after the CRLF and is terminated by another CRLF. Bulk Strings are binary safe and can be up to 512 MB in size.

Null values are represented as bulk strings. The length of the string is set to -1 and the string itself is empty. (Redis Ltd., 2023i)

Integers

Integers are indicated by a colon followed by the integer itself and terminated by a CRLF. For example, the integer 42 is represented as `":42\r\n"`.

Boolean values are represented as integers. The integer 1 represents true and the integer 0 represents false. Commands like EXISTS or SISMEMBER will return such an integer as a response. (Redis Ltd., 2023i)

Arrays

Arrays are indicated by an asterisk followed by the number of elements in the array and a CRLF. Each element of the array is serialized as an RESP type. Arrays can contain mixed types, so it's not necessary for the elements to be of the same type. Nested arrays are also possible.

An empty array has a length of 0 and is represented as `"*0\r\n"`. A null array, on the other hand, has a length of -1 and is represented as `"*-1\r\n"`. (Redis Ltd., 2023i)

Errors

Errors are similar to simple strings, but they are indicated by a minus sign as the first byte of the message. Clients treat errors as exceptions with the string as the error message.

Redis uses a specific error format to indicate the type of error that occurred. The first word of the error message is called Error Prefix and is used to identify the kind of error. The rest of the message is the actual error message. (Redis Ltd., 2023i)

3.5.5 Communication

A client connects to a Redis server by creating a TCP connection to the port 6379. Technically RESP is not TCP specific, but is only used with TCP in the Redis context.

As mentioned earlier a client send commands as an array of bulk strings to the server. For example, the command `SET mykey "Hello"` is represented as `"*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$5`

The response from the server is also serialized as a RESP type. In our example, the server will respond with a simple string "OK" represented as `"+OK\r\n"`. (Redis Ltd., 2023i, 2023j)

3.5.6 Transactions

Redis transactions are a way to group multiple commands into a single transaction. All the commands in a transaction are serialized and executed sequentially. A request sent by another client will never be served in the middle of the execution of a Redis Transaction. This guarantees that the commands are executed as a single isolated operation.

To start a transaction, the client sends the MULTI command to the Redis server. All following commands will be queued up. The client triggers the transaction by sending the EXEC command to the server. This tells the server to start with the execution of the commands.

The WATCH command is used to monitor one or more keys for change before a transaction. If any of the watched keys is modified the transaction will not be executed. This is called optimistic locking. (Redis Ltd., 2022c)

3.5.7 Message Queuing with Redis

An additional feature of Redis is message queuing. Redis calls this Redis Pub/Sub. It provides a simple and efficient way to build real time messaging systems.

It's simple:

- Publishers publish messages to channels
- subscribers receive messages from channels they are interested in.

To commands are pretty self explainatory:

- SUBSCRIBE: to subscribe to a channel
- PUBLISH: to publish a message to a channel
- UNSUBSCRIBE: to unsubscribe a channel

(Redis Ltd., 2023g)

3.6 Redis in Practice

3.6.1 Setup

There are two ways to install Redis on a sever or computer. The recommended method is Docker. For this a working docker installation is necessary. Redis is set up by pulling the official Redis Docker image from Docker Hub and starting the container by using the `docker run` command. It is crucial to enable port-forwarding for the containers to be able to connect to the Redis instance. If persistance is enabled, the `/data` path should be mounted as a Docker volume. Otherwise data would be lost as soon as the container is restarted (Docker, 2023). Another option is native installation. This is only supported on Linux and MacOS. Here Redis is installed using a package manager like `brew`, `apt` or `df`. The operating system's init-manager (in most cases `systemd`) then takes care of launching Redis. In both cases, authentication isn't enabled by default. This has to be configured via a config file or the command line. (Redis Ltd., 2023c)

3.6.2 Example Use Case

To showcase how Redis works in practice and how it compares to relational databases, a simple geolocation game is devised. The game consists out of players that are assigned to teams. Players can move between different geolocations (tiles). The goal of every team is to have visited the most number of tiles last. An entity-relationship model (ERM) for the game can be seen in figure 3.2.

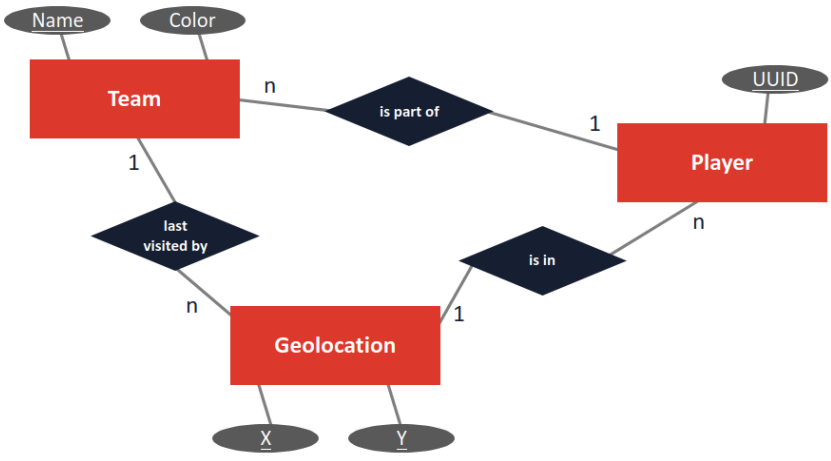


Figure 3.2: Entity Relationship Model for the Example Implementation

The implementation of the game is built in Go [1] using the Redis Golang client. This is similar to a potential implementation with a relational database management system (RDBMS), where an SQL client could be used.

[1] Go is a programming language created by Google in 2010 (Redis Ltd., 2022b)

The ERM consists out of three entities; teams, players and geolocations. In a RDBMS implementation, a table would be created for each of these entities. Redis offers no similar method to separate data. Instead, entities in Redis have to be separated using naming conventions (in this case *entity_type:entity_name*) and retrieved using wildcards.

Secondly, Redis does not force data to follow specific structures, as would be the case in a RDBMS. Instead, users are free to format data in any way they please. The example implementation serialized its data to JSON[2]. Other formats, such as protobufs, would also be conceivable. However, this freedom comes with its downsides, unlike RDBMSs, Redis doesn't make guarantees about the contents and structure of a value. Any player entry from our example could give back an image as well as it could give back a JSON object. Developers therefore have to keep this in mind while developing their applications. This can quickly turn into a problem as an applications' data management and data integrity requirements go up.

Furtherly problematic is Redis' lack of support for references. The ERM contains three relations between its entities. In order to model these relations, entities need a way to reference each other. Unlike with RDBMSs, an application relying on Redis has to implement these references manually. The example implementation embedds Redis keys into its JSON structure to model references. However, this doesn't guarantee referential integrity and burdens the developer with the task of keeping references valid. Similarly, Redis doesn't support joins. Displaying a team count for every tile in the example therefore requires the backend to fetch all data entries and compare them manually. Having to manually ensure referential integrity and implement joins within a backend can be very error-prone. Especially in applications with complex data management needs, this is suboptimal.

3.7 Use Cases

Knowing the advantages and disadvantages of Redis, where does it make sense to use the database? Redis is great at providing near instant responses. Because all data in Redis is stored in memory, retrieval times are extremely low. However, it lacks the extensive data management capabilities found in other databases. Even commonplace features like references or joins aren't present in Redis. Therefore, it mostly makes sense to deploy Redis in places where speed is paramount. Examples include caches or message queues. For these use cases Redis offers numerous features to increase performance wherever possible. In all other use cases, where data integrity and data management is more integral, other databases, such as RDBMSs make more sense.

3.8 Other Key-Value Stores

Beside Redis there are some competitors which are compared in the following. The top competitors are Amazon DynamoDB, Memcached and Hazelcast.

[2]Javascript Object Notation, a popular method for serializing data to strings

Amazon DynamoDB uses solid state drives for the main storage. This is slower than an in memory store like Redis has, but there is an option to enable in memory caching which improves the speed of the database. (Dynobase, 2023) Also, DynamoDB is not open source and only available at AWS, but it has similar data types to Redis. (AWS, 2023)

Memcached is another alternative. The database is an in-memory key value database like redis, but only supports basic data types. (IONOS, 2021), (memcached, 2023)

Hazelcast is in-memory like Memcached and Redis. It is programmed in Java and thus has to deal with the effects of its garbage collector. While running, the garbage collector can cause serious performance hits (Redisson, 2022). In regards to data types, Hazlecast offers simmilar support to Redis (Cahill, 2020). Furthermore, Hazelcast is open source like Redis and has cloud offerings. (solid IT gmbh, 2023b)

3.9 Conclusion

In conclusion, Redis is a fast database with extensible capabilities. Due to being an in-memory database it is great at providing speed. The key value functionality is simple to use due to its basic API, but lacks extensive data management capabilities. Redis is greatly partition tolerant and can shift its focus towards better availability or consistency dependent on the use case. Additionally, Redis provides many mechanisms to deal with the data inside its database. Overall, Redis is an excellent choice for applications that prioritize speed and simplicity over more advanced features.

Sources

AWS. (2023). Supported data types and naming rules in amazon dynamodb. Retrieved March 27, 2023, from https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.NamingRulesDataTypes.html

Bengal, O. (2020, June). Thank You, Salvatore Sanfilippo. Retrieved March 2, 2023, from https://redis.com/blog/thank-you-salvatore-sanfilippo/

Brewer, E. A. (2000). Towards robust distributed systems (abstract). *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*.

Cahill, J. (2020). Overview of hazelcast distributed objects. Retrieved March 27, 2023, from https://docs.hazelcast.com/imdg/4.2/data-structures/overview

Docker. (2023). Redis. Retrieved March 27, 2023, from https://hub.docker.com/_/redis

Dynobase. (2023). Dynamodb vs redis - the ultimate comparison. Retrieved March 27, 2023, from https://dynobase.dev/dynamodb-vs-redis/

Han, J., E, H., Le, G., & Du, J. (2011). Survey on NoSQL database. *2011 6th International Conference on Pervasive Computing and Applications*, 363–366. https://doi.org/10.1109/ICPCA.2011.6106531

InfluxData Inc. (2022). Key-Value Database: How It Works, Key Features, Advantages. Retrieved March 1, 2023, from https://www.influxdata.com/key-value-database/

IONOS. (2021). Memcached vs. redis: A comparison. Retrieved March 27, 2023, from https://www.ionos.com/digitalguide/hosting/technical-matters/memcached-vs-redis/

memcached. (2023). A distributed memory object caching system. Retrieved March 27, 2023, from https://www.memcached.org/

MongoDB Inc. (2023). What Is A Key-Value Database? Retrieved March 1, 2023, from https://www.mongodb.com/databases/key-value-database

Noonan, J. (2023). Database consistency explained. Retrieved May 27, 2022, from https://redis.com/blog/database-consistency/

Ofer Bengal and Yiftach Shoolman. (2021). From our founders: Becoming one redis. Retrieved February 20, 2023, from https://redis.com/blog/becoming-one-redis/

Redis Ltd. (2009). Redis GitHub. Retrieved March 2, 2023, from https://github.com/redis/redis

Redis Ltd. (2014). Garantia Data Changes Company Name to Redis. Retrieved March 2, 2023, from https://redis.com/press/garantia-data-changes-company-name-to-redis-labs/

Redis Ltd. (2022a). Redis strings. Retrieved March 27, 2023, from https://redis.io/docs/data-types/strings/

Redis Ltd. (2022b). Redis strings. Retrieved March 27, 2023, from https://redis.io/docs/data-types/strings/

Redis Ltd. (2022c). Resp protocol spec. Retrieved April 10, 2023, from https://redis.io/docs/manual/transactions/

Redis Ltd. (2022d). Scripting with lua. Retrieved March 27, 2023, from https://redis.io/docs/manual/programmability/eval-intro/

Redis Ltd. (2022e, March). Redis stack and modules. Retrieved March 27, 2023, from https://redis.io/resources/modules/

Redis Ltd. (2023a). Allow writes only with n attached replicas. Retrieved March 27, 2023, from https://redis.io/docs/management/replication/#allow-writes-only-with-n-attached-replicas

Redis Ltd. (2023b). Consistency under partitions. Retrieved March 27, 2023, from https://redis.io/docs/management/sentinel/#consistency-under-partitions

Redis Ltd. (2023c). Getting started with redis. Retrieved March 27, 2023, from https://redis.io/docs/getting-started/#install-redis

Redis Ltd. (2023d). Redis data types. Retrieved March 4, 2023, from https://redis.io/docs/data-types/

Redis Ltd. (2023e). Redis data types tutorial. Retrieved March 27, 2023, from https://redis.io/docs/data-types/tutorial/

Redis Ltd. (2023f). Redis persistence. Retrieved February 20, 2023, from https://redis.io/docs/management/persistence/

Redis Ltd. (2023g). Redis Pub/Sub. Retrieved March 5, 2023, from https://redis.io/docs/manual/pubsub/

Redis Ltd. (2023h). Redis replication. Retrieved March 4, 2023, from https://redis.io/docs/management/replication/

Redis Ltd. (2023i). Resp protocol spec. Retrieved April 10, 2023, from https://redis.io/docs/reference/protocol-spec/

Redis Ltd. (2023j). Resp3 specification. Retrieved April 10, 2023, from https://github.com/redis/redis-specifications/blob/master/protocol/RESP3.md

Redis Ltd. (2023k). Scaling with Redis Cluster. Retrieved March 5, 2023, from https://redis.io/docs/management/scaling/

Redis Ltd. (2023l, January). Introduction to Redis. Retrieved March 27, 2023, from https://redis.io/docs/about/

Redisson. (2022). Feature comparison: Redis vs hazelcast. Retrieved March 27, 2023, from https://redisson.org/feature-comparison-redis-vs-hazelcast.html

Sanfilippo, S. (2017). LMDB – First version of Redis written in Tcl. Retrieved February 19, 2023, from https://gist.github.com/antirez/6ca04dd191bdb82aad9fb241013e88a8

solid IT gmbh. (2023a). DB-Engines Ranking. Retrieved March 1, 2023, from https://db-engines.com/de/ranking/key-value+store

solid IT gmbh. (2023b). Vergleich der systemeigenschaften hazelcast vs. redis. hazelcast vs. redis vergleich. Retrieved March 27, 2023, from https://db-engines.com/de/system/Hazelcast%3BRedis

4. Hazelcast

by Alexander Edinger, Jonas Eppard and Fernand Hoffmann

In recent years, many areas have been changed by digitalization. The digitalization of areas has led to many changes and improvements. The possibilities of storing large amounts of data have become increasingly important. In 2020, a total of 64.2 zettabytes of data were generated in the world. Forecasts for 2025 expect this to increase to 181 zettabytes of data. In order to cope with these large amounts of data, new database management systems are constantly being developed and improved. However, rational databases such as MySQL are not suitable for storing this amount of data efficiently. For this reason, NoSQL database solutions are becoming increasingly important to deal with these data volumes. In this report, the Key-Value-Stores are analysed in more detail and explained using the example of Hazelcast. In the following section, the general concept of key-value databases is discussed. (F. Tenzer, 2022; Ionos, 2020) After that the Hazelcast platform is described. In this section the Hazelcast platform is sorted into the CAP-Theorem. After this we will take at the BASE (short for **B**asically **A**vailable, **S**oft-state and **E**ventual consistency) (Brewer, 2000) approach taken by Hazelcast (Hazelcast, 2022f). After that, we will take a look at Hazelcast in practice. In this section we will discuss the installation of Hazelcast and, the aforementioned example. At the end this seminar paper is a short conclusion and outlook with recommendations when to use Hazelcast.

4.1 Key-Value-Stores

Key-value stores are considered one of the simplest databases and belong to the group of NoSQL databases. Compared to rational databases, which work with tables, key-value stores use the key-value method to store the data. (Hazelcast, 2022v)

4.1.1 History of Key-Value-Stores

The history of the key-value store is difficult to trace due to poor documentation. The first mention of the key-value method was in connection with Charles Babbage's analytical engine in 1837, where the key-value method was to be used to enter values into the analytical engine. The first implementation of the same concept in computer science, however, was implemented in 1953 in the form of hashing. Hashing implements key-value in the same way as databases do. In the field of databases, the first implementation was in 1979 in the form of Ordered Key-Value-Stores. Ordered key-value stores are key-value stores that sort keys. Now that the history of key-value stores has been explained in more

detail, the main elements of the concept will be discussed. (John Walker, 2020; Kurt Mehlhorn; Peter Sanders, 2008; Terence Kelly, 2021)

4.1.2 Working principle of Key-Value-Stores

Key values are composed of two elements, a key and a value. The key element must fulfil the condition of being unique. This condition does not refer to the stored values, these do not have to be unique and can comprise different data types. The data types can range from integer to complex Objects. Data is thus always stored in with a value and a key. In addition to storing data, searches within the database are limited by the key-value method. All searches are always related to keys and cannot access values, so searches related to values are not possible as in rational databases. Now that the basics of key-value stores have been explained, the advantages and disadvantages compared to rational databases will be discussed next. (Hazelcast, 2022v; Ionos, 2020)

4.1.3 Comparison of Key-Value-Stores to relational databases

During the research, advantages and disadvantages of this database were identified. The first advantage of a key-value store is the high speed and performance compared to relational databases. The second advantage is the possibility to scale the database efficiently. All advantages can be traced back to the simple structure of the database. In contrast to rational databases, key-value stores do not require uniform patterns. Thus, the data can be retrieved quickly, and the databases can be extended to new servers. However, these characteristics are also a reason for the disadvantages of key-value stores. The first disadvantage relates to the problem of efficiently representing complex connections in the database. This disadvantage arises from the fact that data is only stored in key-value format and keys are only stored uniquely. Complex connections are much easier to represent in relational databases. The second disadvantage is the problem that no search queries can be made on the values. This disadvantage also stems from the key-value method.

In order to clarify the advantages and disadvantages, the areas of application of this database will now be discussed in more detail. The field of application of key-value stores is limited compared to relational databases. This can largely be explained by the limitations of the structure. Generally, key-value stores are used when large amounts of data need to be accessed quickly. This is a typical case for applications such as shopping carts or session data. Besides these applications, key-value stores can be used for in-memory data caching. This type of use reduces reading and writing on weaker hard disks. The Hazelcast database system also uses this system to retrieve data faster. Hazelcast is discussed in more detail in the next section. (Hazelcast, 2022v; Ionos, 2020)

4.2 Hazelcast in theory

Hazelcast, or formerly known as Hazelcast IMDG, is an In-Memory Data grid written in Java by the Hazelcast corporation. It is a cluster storage and combines "Data at Rest" with "Data in Motion" by providing a streaming architecture for dynamic data together with low-latency storage for static data. (Hazelcast, 2022k, 2022w)

4.2.1 Architecture of Hazelcast

At the core of the Hazelcast platform stands the distributed architecture of cluster nodes with the streaming engine and the low-latency storage. A cluster consists of at least one node. The streaming engine is in the form of a Kappa-Architecture which, compared to a Lambda-Architecture, provides the same methodology for data processing. The low-latency storage provides availability and partition tolerance with possible consistency through the use of the CP-subsystem. Hazelcast offers different possibilities for a client to connect to the cluster through the use of APIs and Connectors. Implementations exist for multiple languages, including Java, .NET, C++, Node.js, Python and Go. (Hazelcast, 2022w) (Hazelcast, 2022h) Encapsulating the Connectors and the cluster are the security features of Hazelcast for clients and nodes. This includes the authentication of new nodes and clients, as well as the encryption of traffic between the nodes or between a client and the cluster.

4.2.2 Hazelcast in CAP-Theorem

The CAP-Theorem states, that a distributed system can not be partition tolerant, consistent and highly available at the same time, but can only fulfil two of the three properties. A 2010 article further adds, that partition tolerance cannot be sacrificed, because sacrificing partition tolerance would require a network to never drop any messages, which in reality cannot exist. (Brewer, 2000; Hale, 2010)

Hazelcast is by default an AP-system, favouring availability over consistency. Since each client interacts with the cluster through a gateway node and node failures can happen in a distributed system, Hazelcast replicates the data on backup nodes to ensure availability. (Hazelcast, 2022b, 2022o)

Some distributed data structures, for example a Lock or a Semaphore, require strong consistency rather than high availability. For such cases, Hazelcast offers a CP-subsystem, which is built upon the cluster and allows for consistent storage of specified distributed objects. The subsystem is based on the RAFT (see Woos et al. (2016)) consensus algorithm, through which the participating nodes in the subsystem can reach a consensus on correct data by periodic heartbeats and votings. (Hazelcast, 2022b, 2022d; Woos et al., 2016)

Since Hazelcast is an In-Memory Data Grid, no persistence happens by default. To strengthen consistency of data structures, persistence can be configured for nodes in the CP-subsystem to persist the data to a non-volatile storage. (Hazelcast, 2022d)

4.2.3 Hazelcast is BASE

The acronym BASE stands for Basically Available, Soft-State and Eventual Consistency and is used to describe a system with those properties (Brewer, 2000). Hazelcast fits all three properties due to the following reasons: Due to Hazelcast being an AP-system, it prioritizes availability and can thus be considered basically available. Through periodic heartbeats between the cluster nodes, a failure in nodes can be detected and data from backup nodes can be fetched. (Hazelcast, 2022f)
Due to the high availability offered by Hazelcast, an exact state of the cluster cannot be determined to a certain point in time. This is further amplified when choosing one-phase commits for transactions, where the commit log is not copied to other nodes. Consequently, a node in the process of writing data can be interrupted due to a node failure and leave the system in an inconsistent state. (Hazelcast, 2022e)

To prevent increasing entropy in the cluster, Hazelcast synchronizes the nodes in periodic intervals. The primary node, the node which is delegated to store the data, sends a summary of the data to its backup nodes. The backup nodes compare the data with their version and in the case of an inconsistency, the synchronization process is triggered. (Hazelcast, 2022m)

4.3 Hazelcast in Practice

The goal of this section is to test Hazelcast in practice. With this goal in mind, we will first install Hazelcast, and then we will try to implement some example data structures. We will take a look at the Usability of the Hazelcast interface.

In this section following limitations are taken into account: First only the Open-Source Hazelcast Platform is used. Second, only the Hazelcast Command Line Interface (CLI) is used. Therefore, the Hazelcast cloud service Viridian is not used and will not be evaluated. Furthermore, the Hazelcast Clients outside the Hazelcast CLI are not used, and will not be evaluated. The data structure used for testing is taken from a previous lecture. TODO:

4.3.1 Installation of Hazelcast

There are multiple ways to install Hazelcast. Due to the fact that Hazelcast is open-source and provides Binaries. Therefore, it's possible to build the Hazelcast Platform from source. However, it is easier to download prebuild binaries. For Linux there is a Debian package available. For macOS there is a brew package available, but for Windows there is no package or installer available. Therefore, under Windows you have to download prebuild binaries, if you don't want to build Hazelcast from source. Another option is to use docker. (Hazelcast, 2022p)

We tried the Hazelcast installation on multiple operating systems, but weren't able to test the installation on macOS. Therefore, we will only describe the installation on Linux and

Windows, and will take a look at the installation via docker. The installation on Linux is very easy and straight forward: First you will download the public key of the Hazelcast repository and add the repository to your package manager sources:

```
wget -qO - https://repository.hazelcast.com/api/gpg/key/public | gpg
   --dearmor | sudo tee /usr/share/keyrings/hazelcast-archive-keyring
   .gpg > /dev/null
echo "deb [signed-by=/usr/share/keyrings/hazelcast-archive-keyring.
   gpg] https://repository.hazelcast.com/debian stable main" | sudo
   tee -a /etc/apt/sources.list
```

Listing 4.1: Adding the Hazelcast repository to the package manager sources under Linux (Debian) (Hazelcast, 2022p)

After that you will be able to install Hazelcast via the package manager like any other package:

```
sudo apt update && sudo apt install hazelcast
```

Listing 4.2: Installing Hazelcast under Linux (Debian) (Hazelcast, 2022p)

The installation under Windows was a bit more complicated. First Hazelcast did not provide a Windows installer. Therefore, we had to download the binary from the Hazelcast website. This binary wasn't able to run directly, because it needed the `JAVA_HOME` environment variable to be set. First we tried to install a Java Runtime Environment (JRE) and set the `JAVA_HOME` variable accordingly. However, the JRE was not able to run the Hazelcast binary and a Java Development Kit (JDK) was needed. Therefore, we installed the JDK set the path, and then the Hazelcast binary was able to run. However, personally we found the installation with docker the easiest. First, the Hazelcast image is downloaded from the docker hub:

```
docker pull hazelcast/hazelcast
```

Listing 4.3: Downloading the Hazelcast image from the docker hub (Hazelcast, 2022p)

After that the local Hazelcast Cluster is started as specified in the documentation:

```
docker network create hazelcast-network
docker run \
    -it \
    --network hazelcast-network \
    --rm \
    -e HZ_CLUSTERNAME=hello-world \
    -p 5701:5701 hazelcast/hazelcast
```

Listing 4.4: Starting the Hazelcast Cluster using the docker image (Hazelcast, 2022u)

For the following example the docker installation will be used.

4.3.2 Example in Hazelcast

In this section an example data structure will be implemented using the Hazelcast CLI. The Hazelcast CLI is run as a docker container. The data structure is taken from a previous lecture and shown in Fig. 4.1. The data structure models a library with books.

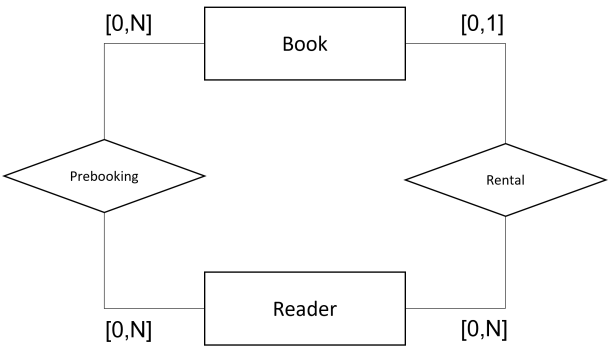


Figure 4.1: Example data structure (see also Mairhofer, 2021)

The library has multiple books and multiple readers. Each book can be rented by one reader. The reader can rent multiple books. And if a book is rented by a reader, another reader can't rent it, but order a prebooking for it. This data structure was given to us in an example in a previous lecture by Mairhofer (2021). The exact fields of the data structure are not important for this example. The important part is that the data structure requires foreign keys to model the relations.

Hazelcast is a distributed key-value store, as described in Section 4.2. It does not provide a relational database, and it has no support for foreign keys. It is possible to implement relations with embedded objects or with user defined fields. To define the Relations between the object classes we will use user defined fields. Embedded objects have the problem, that there are hard to manage by hand and difficult to implement many-to-many relations. In the user defined fields we can define the fields like a foreign key in a relational database, but Hazelcast does not provide any safety checks. Therefore, a user has to implement safety checks by hand, if the consistency in the relations are important. In this example we will not write an application and only take a look at the Hazelcast CLI. Therefore, we will not implement any safety checks. But Hazelcast provides a way to select and join different maps. Thus, it is possible to implement the relations and query those relations inside the Hazelcast CLI.

To conclude our example we have successfully implemented a data structure with foreign keys in the key-value store Hazelcast. However, the relations are not checked by the Hazelcast platform and the user has to implement the checks to ensure that the relations are consistent. Therefore, we advise using Hazelcast as a key-value store and not to try using the Hazelcast platform to implement complex relations.

4.4 Conclusion and Recommendations

Hazelcast is a database that can be used for a variety of different tasks. It offers scalability due to the cluster architecture, the streaming engine makes it viable for fast processing. Due to the default prioritization of availability over consistency, it is suited to handle high amounts of data, while also retaining the possibility of consistency through the CP-subsystem.

Hazelcast especially excels in dynamic environments due to its built-in streaming engine that allows dynamic data to be evaluated and transformed in real-time. The variety of clients and connectors allows for a simple integration of Hazelcast in an application.

Due to the many features that Hazelcast offers, an initial solution built with Hazelcast can grow complex quickly and switching databases from Hazelcast to another provider is coupled with difficulties, because Hazelcast does not offer a direct exportation of data.

In the future, NoSQL-databases are projected to rise in market size. Hazelcast will likely partake in this trend as a Multi-model database with a focus on Key-Value storage. The flexibility to process dynamic data through its streaming engine, as well as the integration of ML-Ops makes Hazelcast a unique database solution worth considering in the future.

Sources

baeldung. (2016, October 21). *An introduction to hazelcast | baeldung*. Retrieved April 10, 2023, from https://www.baeldung.com/java-hazelcast

Brewer, D. E. A. (2000). Towards robust distributed systems.

F. Tenzer. (2022, May 9). *Volumen der jährlich generierten/replizierten digitalen Datenmenge weltweit in den Jahren 2012 und 2020 und Prognose für 2025*. Retrieved April 9, 2023, from https://de.statista.com/statistik/daten/studie/267974/umfrage/prognose-zum-weltweit-generierten-datenvolumen/

Fischer, J., & Lynch, A. (1985). Impossibility of distributed consensus with one faulty process.

Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, *15*(4), 287–317. https://doi.org/10.1145/289.291

Hale, C. (2010, October 7). *You can't sacrifice partition tolerance* [Codahale.com]. Retrieved April 10, 2023, from https://codahale.com//you-cant-sacrifice-partition-tolerance/

Hazelcast. (2022a). An architect's view of the hazelcast platform [Medium: White Paper]. Retrieved March 9, 2023, from https://hazelcast.com/thank-you/?resource_id=2006&form_page=https%3A%2F%2Fhazelcast.com%2Fresources%2Farchitects-view-hazelcast%2F

Hazelcast. (2022b). *CAP theorem* [Hazelcast]. Retrieved April 10, 2023, from https://hazelcast.com/glossary/cap-theorem/

Hazelcast. (2022c). *Clients & languages* [Hazelcast]. Retrieved March 13, 2023, from https://hazelcast.com/clients/

Hazelcast. (2022d). *CP subsystem*. Retrieved April 10, 2023, from https://docs.hazelcast.com/imdg/4.2/cp-subsystem/cp-subsystem

Hazelcast. (2022e). *Creating a transaction interface*. Retrieved April 10, 2023, from https://docs.hazelcast.com/imdg/4.2/transactions/creating-a-transaction-interface

Hazelcast. (2022f). Failure Detector Configuration. Retrieved April 12, 2023, from https://docs.hazelcast.com/imdg/4.2/clusters/failure-detector-configuration

Hazelcast. (2022g). *Feature comparison* [Hazelcast]. Retrieved April 10, 2023, from https://hazelcast.com/product-features/feature-comparison/

Hazelcast. (2022h). Getting Started with a Hazelcast Client. Retrieved April 12, 2023, from https://docs.hazelcast.com/hazelcast/latest/clients/hazelcast-clients

Hazelcast. (2022i). *Hazelcast*. Retrieved March 7, 2023, from https://github.com/hazelcast

Hazelcast. (2022j). *Hazelcast - first application*. Retrieved April 10, 2023, from https://www.tutorialspoint.com/hazelcast/hazelcast_first_application.htm

Hazelcast. (2022k). *Hazelcast IMDG — leading open source in-memory data grid* [Hazelcast]. Retrieved April 10, 2023, from https://hazelcast.org/imdg/

Hazelcast. (2022l). *Hazelcast topologies*. Retrieved April 12, 2023, from https://docs.hazelcast.com/hazelcast/5.0/topologies

Hazelcast. (2022m). *Hazelcast's replication algorithm*. Retrieved April 10, 2023, from https://docs.hazelcast.com/hazelcast/5.0/consistency-and-replication/replication-algorithm

Hazelcast. (2022n). *Hazelcast/hazelcast* [GitHub]. Retrieved March 7, 2023, from https://github.com/hazelcast/hazelcast

Hazelcast. (2022o). *HazelVision episode 06 — CAP theorem — YouTube*. Retrieved April 10, 2023, from https://www.youtube.com/watch?v=6DHxX_-fA8Y&list=PLhAaDrEJmCb3lpFQ6kDOkf_9wSdRp1cgx&index=10

Hazelcast. (2022p). *Installing hazelcast open source*. Retrieved March 7, 2023, from https://docs.hazelcast.com/hazelcast/latest/getting-started/install-hazelcast

Hazelcast. (2022q). *Kappa architecture* [Hazelcast]. Retrieved April 10, 2023, from https://hazelcast.com/glossary/kappa-architecture/

Hazelcast. (2022r). *Micro batch processing* [Hazelcast]. Retrieved April 10, 2023, from https://hazelcast.com/glossary/micro-batch-processing/

Hazelcast. (2022s). *Overview of hazelcast distributed objects*. Retrieved April 10, 2023, from https://docs.hazelcast.com/imdg/4.2/data-structures/overview#:~:text=Hazelcast%20has%20two%20types%20of%20distributed%20objects%20in,stores%20the%20whole%20instance%2C%20namely%20non-partitioned%20data%20structures.

Hazelcast. (2022t). *SQL*. Retrieved March 9, 2023, from https://docs.hazelcast.com/hazelcast/latest/sql/sql-overview

Hazelcast. (2022u). *Start a local cluster in docker*. Retrieved April 8, 2023, from https://docs.hazelcast.com/hazelcast/latest/getting-started/get-started-docker

Hazelcast. (2022v). *What is a key-value store?* Retrieved April 13, 2023, from https://hazelcast.com/glossary/key-value-store/

Hazelcast. (2022w, November 3). *What is the hazelcast platform? | hazelcast explainer*. Retrieved April 10, 2023, from https://www.youtube.com/watch?v=UpIgHzKbMp0

Ionos. (2020, March 10). *Key Value Store: How do Key Value Databases work?* Retrieved April 10, 2023, from https://www.ionos.com/digitalguide/hosting/technical-matters/key-value-store/

John Walker. (2020). *The Analytical Engine Programming Cards*. Retrieved April 13, 2023, from https://www.fourmilab.ch/babbage/cards.html

Kreps, J. (2022). *Questioning the lambda architecture - o'reilly radar*. Retrieved April 10, 2023, from http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

Kurt Mehlhorn; Peter Sanders. (2008). *Hash Tables and Associative Arrays*. Retrieved April 13, 2023, from https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/HashTables.pdf

Mairhofer, K. (2021, April 14). Aufgabe SQL Buchausleihe.

Ongaro, D., & Ousterhout, J. (n.d.). In search of an understandable consensus algorithm.

solid IT gmbh. (2023). *DB-engines ranking* [DB-engines]. Retrieved April 14, 2023, from https://db-engines.com/en/ranking

Terence Kelly. (2021). *Crashproofing the Original NoSQL Key-Value Store*. Retrieved April 13, 2023, from https://mydigitalpublication.com/publication/?i=721263&article_id=4113294&view=articleBrowser

Woos, D., Wilcox, J. R., Anton, S., Tatlock, Z., Ernst, M. D., & Anderson, T. (2016). Planning for change in a formal verification of the raft consensus protocol. *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, 154–165. https://doi.org/10.1145/2854065.2854081

5. MongoDB

by Marina Chiselev, Sophie Gösch, Sabrina Ladner, Jannik Springer

5.1 Introduction to Document DBs

5.1.1 General information and characteristics

Document-oriented databases are a one type of NoSQL databases that have gained popularity in recent years (Bach et al., 2016, p. 487). Unlike relational databases, where data is organized in tables, document-oriented databases store data as documents in collections (Bach et al., 2016, p. 493).

In general these documents and collections are "schemaless", which means that the structure of documents can be flexible (Bach et al., 2016, p. 493). This property facilitates agility in application development, particularly in applications with a constantly changing data model (Bach et al., 2016, p. 486). Another strength of document-oriented databases is their ability to manage large amounts of data (Bach et al., 2016, p. 487). Document-oriented databases can distribute large amounts of data across multiple servers to ensure high scalability. This makes it possible to process large amounts of heterogeneous data (Meier & Kaufmann, 2016, p. 229).

5.1.2 History

A basic idea behind the development of document based databases is to store all data of an object in the same place, i.e. in a single document (Faeskorn-Woyke et al., 2013). One of the first document oriented databases is Lotus Notes from IBM, which has been developed in 1984 (Faeskorn-Woyke et al., 2013).

The development of modern document-oriented databases began in the late 2000s as part of the NoSQL movement, which focused on processing large amounts of data (Eisermann & Gerold, n.d., p. 41). MongoDB, CouchDB, and Firebase are among the significant document-oriented databases and are among the top 14 databases according to the 2021 StackOverflow survey (Eisermann & Gerold, n.d., p. 41).

5.2 Comparing document DB and relational DB

To compare the two types of databases, we will use the following table.

Category	Relational database	Document database
Language	SQL	Proprietary
Group of entries	Table	Collection
Entry	Row	Document
Schema	Fixed	Not fixed
Linking data	Join	Embedded documents

Table 5.1: Comparison of relational databases and document databases (Chauhan, 2019, p. 91; Ellingwood, 2022)

The main difference between relational and document databases is the way how data is stored. In relational databases, data is stored in tables with fixed schemas. Each table has a unique key to identify the rows (Chauhan, 2019, p. 91; Ellingwood, 2022). And to build relations between tables like 1:1, 1:n or n:m joins are used (Studer, 2019, pp. 25-31). In document databases, data is stored in schemaless documents which are grouped in collections. All data related to one object is stored in one document and no joins are needed to access its data (Meier & Kaufmann, 2016, pp. 18-20).

5.3 Design and Structure of MongoDB

According to db-engines.com, MongoDB is currently the most popular document database as well as the most popular non-relational database overall and has been for many years by now (solid IT GmbH, 2023a, 2023b).

As is the case in any other document database, a record in MongoDB is a document with pairs of fields and their values (Chauhan, 2019, p. 90). The database itself is a container for so-called collections, which in turn are containers for arbitrary groups of documents (Chauhan, 2019, p. 90). In the case of MongoDB the documents are similar to JSON (JavaScript Object Notation) objects, although instead of using native JSON MongoDB stores data in a format called BSON, which stands for binary JSON (Membrey et al., 2014, p. 4; Chauhan, 2019, p. 90). Contrary to what one might expect when thinking of BSON as a binary form of JSON, BSON is not necessarily more space efficient than JSON and can in many cases take up more storage space than JSON (Membrey et al., 2014, p. 11). The design of MongoDB prioritizes performance over space-efficiency. Thus, BSON sacrifices some disk space in order to gain indexing and query performance (Membrey et al., 2014, p. 11). Additionally, converting BSON to the native data format of a programming language is generally faster and easier than doing the same with JSON data (Membrey et al., 2014, p. 11).

Apart from being fast, MongoDB is also designed to be horizontally scalable (Membrey et al., 2014, pp. 2, 6–7). In contrast to vertical scaling, which is achieved by upgrading the resources of a single server to improve performance, horizontal scaling is achieved by dividing the data set or workload of a single server across multiple servers and adding servers to increase capacity as needed (MongoDB Inc., 2023x). To that end, MongoDB implements a process called "sharding". This involves splitting the data set at the collection level and distributing the documents of a collection across the shards (data nodes) in the cluster (MongoDB Inc., 2023x; Membrey et al., 2014, p. 7). In addition to the shards, a cluster then also requires config servers and query routers, which provide an interface between the client applications and the cluster (MongoDB Inc., 2023x).

Another major concept behind the design of MongoDB is that there should always be multiple copies of data (Membrey et al., 2014, p. 2). To achieve this, MongoDB provides replica sets. A replica set is a group of MongoDB nodes that contain the same data set (MongoDB Inc., 2023w; Chauhan, 2019, p. 90). Exactly one of the data nodes is considered the primary node, while the other data nodes are secondary nodes. Write operations to a replica set can only be received and executed by its primary node while read operations can be sent to any data node in the replica set (MongoDB Inc., 2023w). When the primary node becomes unavailable, the remaining nodes elect one of the secondary nodes to be the new primary node (MongoDB Inc., 2023w).

5.4 API

To interact with a database, MongoDB provides drivers for a sizeable collection of modern programming languages as well as a shell called `mongosh`, which itself is a fully functional Node.JS environment (MongoDB Inc., 2023q, 2023t). The shell and drivers can be used to execute CRUD (create, read, update and delete) operations on documents of the database. At that, these operations are performed in the context of collections (MongoDB Inc., 2023p; Truica et al., 2013, pp. 347-349).

When creating a new document, if the collection of the document does not yet exist, it is implicitly created (MongoDB Inc., 2023h). Additionally, each document requires a unique `_id` field, which is used as the primary key. This field will be generated, if it is not provided when inserting a new document (MongoDB Inc., 2023h). Once set, this field cannot be changed and is thus immutable (MongoDB Inc., 2023y). In general, all write operations (creating, updating or deleting documents) in MongoDB are atomic on the level of documents (MongoDB Inc., 2023e, 2023h, 2023y).

When reading documents from a collection, the MongoDB driver or shell returns a cursor which manages the query results (MongoDB Inc., 2023u). This cursor can then be iterated to access the documents that resulted from the query (MongoDB Inc., 2023n).

5.5 Installation

There is a significant list of packages that you can download on the website of MongoDB: www.mongodb.com. MongoDB provides a Community Edition as well as an Enterprise Edition (MongoDB Inc., 2023i).

5.5.1 Installation requirements

The MongoDB server's options and settings are specified using configuration files, which are YAML-files containing the relevant parameters. These configuration files allow for easy management and specification of options for the database server (MongoDB Inc., 2023c).

For guarantying that every function and feature can be fully used, there are some supported platforms, with a list including all information, so you know if a version is available on your system. You can view this information on the MongoDB installation page in the docs: https://www.mongodb.com/docs/manual/installation/. MongoDB can be installed natively on the following platforms:

- Amazon Linux V1 and V2
- Debian 9, 10, 11
- RHEL/CentOS, Oracle Linux, Rocky, Alm a 9.0+, 8.0+, 7.0+, 6.2+
- SLES 12, 15
- Ubuntu 22.04, 20.04, 18.04, 16.04
- Windows Server 2019, 2008/2012/2012R2, 2016
- Windows 10
- macOS 11, 10.14+, 10.13, 10.12

(MongoDB Inc., 2023i)

5.5.2 Installation on Linux and Docker

The preferred way to install MongoDB natively on a Linux desktop or server is by using the packages provided by MongoDB. MongoDB also provides guides detailing the installation process (MongoDB Inc., 2023j).

Both editions of MongoDB can also be run in a Docker Container. To that end, MongoDB maintains the respective Docker images and provides a detailed installation guides (MongoDB Inc., 2023k, 2023l).

5.6 Testing MongoDB

5.6.1 Relational example (own example)

To test MongoDB in practice, an example from the course "Web-Development II" was used. The example was a management system for shopping lists, in which shopping lists are objects with a name and a list of items. You can check the items on a list, to see if you bought it or not. In relational databases this example would be implemented using two tables ("lists" and "items") with a one-to-many relation where each list could contain multiple items. The test was performed on a locally installed MongoDB server using the native `mongosh`.

All CRUD operations have been tested. In the collection "lists", we created 2 documents using `db.lists.insertOne()`. Since items are contained in a list, in MongoDB these items are called "embedded documents" (Membrey et al., 2014, p. 6). To read all documents in a collection, you use the command: `db.lists.find()` and for a specific read you add a filter inside the find()-Method (MongoDB Inc., 2023u). The Update-Functionality has been tested too, by using `db.lists.updateOne()`, to update a single document. Deleting a document has been tested using `db.lists.deleteOne()` with a filter as a parameter.

5.6.2 User Experience during the test

Compared to a relational database inserting, querying and deleting documents is very easy, since the documents are basically JSON objects. At filtering the documents you recognize that the filters are objects (documents) themselves. Additionally, no object-relation mapping was required.

Since MongoDB implements relations with embedded documents, filtering on a parameter in an embedded document can be harder when compared to a relational approach, since the filters have to traverse every level of the containing document (MongoDB Inc., 2023v). This traversing also needs to be done, when overwriting (updating) values in an embedded document. Thus, Read and Update operations can become difficult in embedded documents (MongoDB Inc., 2023y).

5.7 Advantages and Disadvantages

Like any document oriented database, MongoDB is schemaless and not normalized. Thus, the structure of the stored data can be highly flexible (MongoDB Inc., 2023b). On the other hand this leads to more storage space being used (Cottrell, 2020, p. 2).

Another advantage is the ability to scale the database horizontally, i.e. by adding more servers instead of upgrading a single server. Thus, MongoDB can be run on commodity hardware, which reduces the cost of scaling (Cottrell, 2020, p. 1; MongoDB Inc., 2023x). In addition to increased storage capacity and performance, data is stored redundantly in

a MongoDB cluster. These aspects are important in a big data environment (MongoDB Inc., 2023b).

The storage of all data in the BSON data format enables easy access from any language, like for example Python, JavaScript, Java, etc. (MongoDB Inc., 2023b). Additionally, multi-document ACID transactions are supported to ensure data integrity (MongoDB Inc., 2023a).

Last but not least there are some limits in MongoDB. The maximum size of a document is 16 megabytes and the maximum levels of nesting is 100 for BSON documents (MongoDB Inc., 2023s).

5.8 Position according to CAP

According to Brewer the CAP theorem includes consistency (C), availability (A), and tolerance to network partitions (P) as its components, and it is not possible to achieve all three properties at the same time (Brewer, 2000).

The consistency property can be differentiated into strong consistency, where changes from one node are immediately visible on all nodes, and eventual consistency, where consistency does not have to be achieved immediately. Based on that it is necessary to build additional logic to handle potentially inconsistent data (MongoDB Inc., 2018, pp. 4-5).
MongoDB can provide either strongly consistent or eventually consistent data, depending on the use case (MongoDB Inc., 2018, p. 5).

Since sharding can be used with MongoDB, data can be stored on multiple nodes and thus be available even if one or more nodes are down or not reachable. Thus, when using sharding, MongoDB has high availability even in the event of a partition (MongoDB Inc., 2023x; Membrey et al., 2014, p. 7).

Replica sets lay the foundation for high availability and redundancy. With multiple replications of the data to different database servers a partition tolerance is provided. Maintaining data copies at different server locations can increase data locality and availability. In addition, data copies can be used for example as backups (MongoDB Inc., 2023w).

MongoDB is considered as a CP database, because it focuses on consistency across the entire system and partition tolerance (Jayasekara, 2021).

5.9 Recommendation and Conclusion

5.9.1 Recommendation for specific use cases

MongoDB is a good choice for applications that need to store large amounts of structured or unstructured data and thus require a scalable solution (MongoDB Inc., 2023z).
Since MongoDB is highly scalable and can handle large amounts of data, it is an excellent choice for analytics platforms (MongoDB Inc., 2023g).

MongoDB is also well-suited for IoT applications, as it can quickly and efficiently store many unstructured data points from a variety of devices (MongoDB Inc., 2023m).
MongoDB can also be useful for e-commerce applications, as it allows for fast transaction processing and provides flexible data modeling (MongoDB Inc., 2023f).
MongoDB is also well-suited for mobile applications, as it is lightweight and allows for fast data processing (MongoDB Inc., 2023r).
Furthermore, MongoDB is commonly used for real-time applications that are executed at different locations (Jayasekara, 2021).

5.9.2 Conclusion

MongoDB is a powerful and very popular NoSQL database. Like any document oriented database, it is very flexibility and thus allows developers to quickly create new features and models without worrying about strict schemas and relationships between the data (MongoDB Inc., 2023g).

Another important advantage of MongoDB is its integration with many modern technologies and frameworks. MongoDB provides drivers for many programming languages and is also available on many cloud platforms, making implementation easier and faster (MongoDB Inc., 2023o). Furthermore, MongoDB has also developed a cloud-based platform called Atlas that allows customers to host and manage their MongoDB instances in the cloud. Atlas also provides features such as automatic scaling and backups that make managing MongoDB databases easier (MongoDB Inc., 2023d).

Regarding the CAP theorem, MongoDB is considered as a CP database (Jayasekara, 2021).

Overall, MongoDB is a powerful database that is suitable for many use cases and is continuously evolving to meet the requirements of modern applications.

Sources

Bach, C., Kundisch, D., Neumann, J., Schlangenotto, D., & Whittaker, M. (2016, August). *Dokumentenorientierte nosql-datenbanken in skalierbaren webanwendungen.*

Brewer, E. A. (2000). Towards robust distributed systems (abstract). *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing.*

Chauhan, A. (2019). A review on various aspects of mongodb databases. *International Journal of Engineering Research & Technology (IJERT)*, *8*(05), 90–92.

Cottrell, N. (2020, September). *Mongodb topology design* (1st ed.). APress.

Eisermann, T., & Gerold, M. (n.d.). Dokumentenorientierte datenbanken. *13.2022*, 41–47. https://nbn-resolving.org/urn:nbn:de:bsz:fn1-opus4-87596

Ellingwood, J. (2022). *Relational databases vs document databases.* Retrieved March 14, 2023, from https://www.prisma.io/dataguide/types/relational-vs-document-databases

Faeskorn-Woyke, H., Prof. Dr., Bertelsmeier, B., Liß, N., Gawenda, D., & Kasper, A. (2013). *Dokumentenorientierte datenbank*. Retrieved March 18, 2023, from https://wikis.gm.fh-koeln.de/Datenbanken/DokumentenorientierteDatenbank

Jayasekara, R. (2021). *Related*. Retrieved March 7, 2023, from https://rukshanjayasekara.wordpress.com/2021/03/06/cap-theorem-and-how-its-applied-in-mongodb/

Meier, A., & Kaufmann, M. (2016, July). *Sql- & nosql-datenbanken* (8th ed.). Springer.

Membrey, P., Hows, D., & Plugge, E. (2014, December). *Mongodb basics* (1st ed.). Apress.

MongoDB Inc. (2018, June). *Top 5 considerations when evaluating nosql databases*.

MongoDB Inc. (2023a). *Acid transactions in mongodb*. Retrieved March 12, 2023, from https://www.mongodb.com/transactions

MongoDB Inc. (2023b). *Advantages of mongodb | mongodb*. Retrieved March 12, 2023, from https://www.mongodb.com/advantages-of-mongodb

MongoDB Inc. (2023c). *Configuration file options*. Retrieved March 18, 2023, from https://www.mongodb.com/docs/manual/reference/configuration-options/

MongoDB Inc. (2023d). *Database. deploy a multi-cloud database*. Retrieved March 15, 2023, from https://www.mongodb.com/atlas/database

MongoDB Inc. (2023e). *Delete documents*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/v6.0/tutorial/remove-documents/

MongoDB Inc. (2023f). *E-commerce in mach-geschwindigkeit mit mongodb und commercetools*. Retrieved March 15, 2023, from https://www.mongodb.com/collateral/ecommerce-at-mach-speed-with-mongodb-and-commercetools-german

MongoDB Inc. (2023g). *How to scale mongodb*. Retrieved March 15, 2023, from https://www.mongodb.com/basics/scaling

MongoDB Inc. (2023h). *Insert documents*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/v6.0/tutorial/insert-documents/

MongoDB Inc. (2023i). *Install mongodb*. Retrieved March 13, 2023, from https://www.mongodb.com/docs/manual/installation/

MongoDB Inc. (2023j). *Install mongodb community edition on linux*. Retrieved March 18, 2023, from https://www.mongodb.com/docs/manual/administration/install-on-linux/

MongoDB Inc. (2023k). *Install mongodb community with docker — mongodb manual*. Retrieved March 18, 2023, from https://www.mongodb.com/docs/manual/tutorial/install-mongodb-community-with-docker/

MongoDB Inc. (2023l). *Install mongodb enterprise with docker*. Retrieved March 18, 2023, from https://www.mongodb.com/docs/manual/tutorial/install-mongodb-enterprise-with-docker/

MongoDB Inc. (2023m). *Internet of things (iot) databases*. Retrieved March 15, 2023, from https://www.mongodb.com/use-cases/internet-of-things

MongoDB Inc. (2023n). *Iterate a cursor in mongosh*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/v6.0/tutorial/iterate-a-cursor/

MongoDB Inc. (2023o). *Mongodb compatibility & integration*. Retrieved March 15, 2023, from https://www.mongodb.com/compatibility

MongoDB Inc. (2023p). *Mongodb crud operations*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/v6.0/crud/

MongoDB Inc. (2023q). *Mongodb drivers api documentation*. Retrieved March 12, 2023, from https://api.mongodb.com/

MongoDB Inc. (2023r). *Mongodb for mobile*. Retrieved March 15, 2023, from https://www.mongodb.com/use-cases/mobile

MongoDB Inc. (2023s). *Mongodb limits and thresholds*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/upcoming/reference/limits/#mongodb-limit-BSON-Document-Size

MongoDB Inc. (2023t). *Mongodb shell*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/mongodb-shell/

MongoDB Inc. (2023u). *Query documents*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/v6.0/tutorial/query-documents/

MongoDB Inc. (2023v). *Query on embedded/nested documents*. Retrieved March 18, 2023, from https://www.mongodb.com/docs/v6.0/tutorial/query-embedded-documents/

MongoDB Inc. (2023w). *Replication*. Retrieved March 7, 2023, from https://www.mongodb.com/docs/v6.0/replication/

MongoDB Inc. (2023x). *Sharding*. Retrieved March 7, 2023, from https://www.mongodb.com/docs/manual/sharding/

MongoDB Inc. (2023y). *Update documents*. Retrieved March 12, 2023, from https://www.mongodb.com/docs/v6.0/tutorial/update-documents/

MongoDB Inc. (2023z). *Why use mongodb*. Retrieved March 15, 2023, from https://www.mongodb.com/why-use-mongodb

solid IT GmbH. (2023a). *Db-engines ranking - popularity ranking of database management systems*. Retrieved March 7, 2023, from https://db-engines.com/en/ranking

solid IT GmbH. (2023b). *Historical trend of the popularity ranking of database management systems*. Retrieved March 7, 2023, from https://db-engines.com/en/ranking_trend

Studer, T. (2019, October). *Relationale datenbanken: Von den theoretischen grundlagen zu anwendungen mit postgresql*. Springer-Verlag.

Truica, C.-O., Boicea, A., & Trifan, I. (2013). Crud operations in mongodb. *Proceedings of the 2013 International Conference on Advanced Computer Science and Electronics Information*. https://doi.org/10.2991/icacsei.2013.88

6. Cloud Firestore

by Maximilian Nagel, Marcel Fleck, Tom Freudenmann

Cloud Firestore is a cloud hosted document-oriented database. This means, the data is saved and queried as JavaScript Object Notation (JSON)-like documents. It allows the users to store and sync application data in real-time, offers security rules to easily manage secure access and the ability to scale automatically if needed. Furthermore, Firestore is part of Googles developer suit "Firebase". Thus, it is possible to use other Firebase functions like more advanced authentication using Google, Facebook or Twitter accounts, making it very versatile for users (Firebase, 2017).

Firestore is mostly used for web and mobile applications for all kinds of devices. Additionally, it offers an offline mode (Google, 2023b), which allows apps to work with the data of the database without a permanent connection to the internet. This ultimately optimizes Firestore for precisely these areas of application.

To emphasize the business model of Firestore, the developers had the following slogan:

> "Spend less time on infrastructure issues and more time on developing your app" (Firebase, 2017)

With an integration of Firestore into Firebase, this statement is very accurate in its meaning. Creating your own login and handling the associated credentials can be very tricky. If the credentials are outsourced to Firebase, a safe implementation can be used, saving a lot of time.

Moreover, how Firestore can be classified in the context of this statement will be explained in the course of this elaboration.

The following paper will give an understanding of Cloud Firestore, starting with a brief overview over its history, followed by an explanation of the architecture and the categorization of Firestore in the CAP-Theorem. Finally, the advantages and disadvantages of Firestore are discussed, and the most important takeaways are pointed out condensed.

6.1 History

On October 3, 2017, Cloud Firestore had its first release as a beta product to extend Firebase and its existing simple Database with new, more complex functionalities. Almost half a year later, in February 2018, Google released new libraries for C#, PHP and Ruby, giving the Developers a possibility to choose in which programming language they want to access their Firestore database (Google, 2023e).

In April 2018, an SDK for Cloud Functions was released for Firestore. That meant, it was from then on possible to handle events without the need of updating the client. With this addition, Firestore can now completely be managed server sided (Google, 2023e).

Since August 2018, Firestore has been extended to export and import data. With these functionalities, switching from one database to another is now easier and less time-consuming. In the same breath, Firestore rolled out to more servers, including Western Europe and the Eastern US, making it accessible for more users (Google, 2023e).

Approximately two years after the beta release, Firestore was finally released to the public in January 2018. With this release, the "Cloud Firestore Service Level Agreement" (Google, 2020) was put into effect, guaranteeing an availability of 99.999% on multi-region servers and an availability of 99.99% for regional servers. Additionally, Firestore was extended to even more servers, including, a multi-regional server in Europe and three new regional servers in Asia (Google, 2023e).

To this date, Firestore has expanded to more servers around the world, making it available on two multi-region servers and 23 regional servers (Google, 2023f). From a technical point of view, Firestore also improved the possible queries by adding new options and functions. This opens up more possible use cases for developers (Google, 2023e).

6.2 Architecture

Having a general overview of Firestore and its advantages for users, this chapter explains the technical architecture of Firestore to understand how it works and why this results in the mentioned features of Firestore.

A first basic information to clarify are data types supported by Firestore. These embrace boolean, number, string, geo point, binary blob and timestamp. Arrays and nested objects, called maps, are available as well and represent structures of multiple attributes of these data types (Google, 2023a).

6.2.1 Documents

The basic instance in Firestore is named *Document*. A document has a (not mandatory unique) name and represents "a lightweight record that contains fields which map to values" (Google, 2023d). As a result, a document is comparable to an object in a SQL-Database. This is confusing, as the name convention for Firestore defines the attributes of a document as objects (Google, 2023d).

As already indicated, objects can be more complex with arrays and maps. For example, instead of providing a document representing a person the separate object's first name and surname, it is possible to create an object name as a map of the two objects (Google, 2023d).

6.2.2 Colletions

Collections are the exposition to build the shape of a Firestore structure. They are the place for documents to live in. The special thing about collections is that there is no structure predefined. Documents can have objects with data types completely flexible. Even in the same collection, it is not necessary to build documents with the same shape. Actually, it is technically also no problem to store two completely different documents in one collection. However, it is highly recommended to structure documents within one collection similar as far as possible to make querying information easy (Google, 2023d). It is important to mention that collections can only contain documents and nothing else. This implies that it is not possible to store plain values (objects) directly in a collection (Google, 2023d).

Moreover, collections do not need to be created nor deleted. Whenever the first document of a collection is created, the collection automatically gets generated. The same applies to the deletion of a collection: Whenever the last document of an existing collection is extinguished, the collection is no longer existent as well (Google, 2023d).

6.2.3 References

Of course, referencing is also very important in Firestore. As each document is uniquely identifiable, referencing is quite easy. The definite implementation is logically related to the programming language used, but it always just uses the collection identifier and the document identifier. Referencing collections themselves works the same, just without the document ending (Google, 2023d).
A reference is very loose. It is possible to create a reference without the referenced object even existing. Additionally, when a referenced object is renamed or deleted, this does not result in a conversion of the reference (Google, 2023d).

6.2.4 Hardware Architecture

Another important part of the architecture of Firestore is the hosting of the cloud database. As this book is only about databases, explaining cloud computing would be too in-depth. If necessary, a good definition and explanation of cloud computing is provided by IBM (2023). As the technology is not locally installed, inspecting the location of servers in order to evaluate the quality of the cloud database is required.

Firestore databases can be set up in two different ways: multi-region locations and regional locations. Regional locations represent one region (e.g. Sydney). This does not mean, that there is one datacenter in the region of Sidney, instead there are multiple ones, located at least 100 miles away from each other. Thus, if one sets up a regional database, it is stored as replicas in the datacenters of the region.
The other option are multi-regional databases. With multi-region, one can choose between *eur3* and *nam5*. *eur3*, for example, means that the database is hosted in datacenters located in different regions of the sphere of Europe. The database chosen to be hosted

in *eur3* can have its replicas in Spain, Finland, Georgia and Switzerland. On the one hand, the multi-region hosting is more stable against breakdowns due to the geographical distribution. On the other hand, requests from e.g. Sydney will take longer to a multi-regional database hosted in North America than a regional server in Sydney (Kerpelmann, 2017).

Moreover, the Google network is designed in a way that in case of the breakdown of one datacenter, upcoming requests are scaled automatically and distributed to the other servers. As a result, Google can ensure the uptime of 99.999 percent in their SLA (cmp. chapter 6.1). Multi-regional locations, logically, are in the aspect of reliability stronger (Kerpelmann, 2017).

Briefly stated, a Firestore database is always created with multiple synchronized copies so that data can never get lost in normal circumstances.

To summarize the architecture and technology of Firestore, it can be stated that the Google NoSQL database tries to offer the most flexible and general data storage possible. However, this offers almost infinite possibilities and can therefore be used for a variety of applications. This liberate structure, in return, also requires the people dealing with the system to know what they are doing as technical misunderstandings can result in complicated or even unsolvable problems. As the name states, Cloud Firestore is a Cloud-offered database and therefore brings the known advantages of cloud computing.

6.3 CAP-Theorem

The CAP-Theorem was introduced by E. A. Brewer (2000) and defines a triangle of properties. The three letters mean Consistency, Availability, and Partition Tolerance.

Firstly, **Consistency** describes that every request gets the latest atomic result of the system. As a result, all requests get a consistent equal answer at the same time (E. A. Brewer, 2000; Gilbert & Lynch, 2012).

Secondly, **Availability** has the goal to always receive an answer from the system. Today, this includes a small latency of the system (E. A. Brewer, 2000; Gilbert & Lynch, 2012).

Finally, a distributed system must be robust to failures of individual network nodes, partitions, or connections to meet the **Partition Tolerance**. This includes no data loss after a failure or natural disaster (E. A. Brewer, 2000; Gilbert & Lynch, 2012).

The theorem asserts that all three properties can never be satisfied simultaneously by a distributed system. Therefore, a distinction is made between CA, AP and CP systems, where every system satisfies two of the three properties (E. A. Brewer, 2000; Gilbert & Lynch, 2012).

6.3.1 Availability of Firestore

Firestore is part of Google Cloud Firebase, which is mostly used for web and mobile app development. Furthermore, Google sells Cloud Firestore as a database including real-time updates and SDKs for different programming languages, such as Flutter, Python, or JavaScript (Google, 2023e). Looking closer into the SLA of Cloud Firestore, Google promises a 99.999% availability of the database in the multi region cluster (Google, 2020).

Furthermore, Google does not provide an agreement on the latency of Cloud Firestore. The latency of the database used for the project done in this lecture is 0.049 to 0.500 seconds. This means that a request to Cloud Firestore to read a document takes 49 to 500 milliseconds. As a result, the database feels fast and suggests the user a good availability of the service.

To always provide the client an answer, Cloud Firestore uses an offline mode, in which the current state of the database is cached. New write operations are then buffered and sent when the client is connected to the database (Google, 2023e). This should improve the feeling of availability, but the fact that the client does not get the actual current state of the database indicates that this method is not really accurate.

Moreover, Firestore builds on top of the Google Cloud and uses the same technologies and infrastructures (Kerpelman, 2019). As a result, the high availability of 99.999% can be guaranteed because Cloud Firestore uses the own network architecture of Google (Kerpelman, 2019).

Finally, the technical architecture of Cloud Firestore is not designed to have a high availability or low latency (Kerpelman, 2019). Instead, Firestore feels very fast because it has an offline cache and uses the private network of Google for fast respond times. The good availability and low latency is important to develop responsive web and mobile applications.

6.3.2 Firestore as CP-System

One of the key features described in the release blog entry of the Firestore developers is that Cloud Firestore provides “Automatic, multi-region data replication with strong consistency” (Dufetel, 2017). Dividing this key feature into two properties, they describe the Partition Tolerance and Consistency of Cloud Firestore.

Firstly, looking deeper into the architecture of Cloud Firestore, the database is distributed and replicated on multiple servers in the world (Google, 2020). Moreover, Cloud Firestore is a fully managed product, scaling its resources automatically (Dufetel, 2017). Thus, Cloud Firestore is robust against regional disasters and can restore the data easily (Kerpelman, 2019). As a result, Cloud Firestore can be seen as partition tolerant from a technical point of view.

Secondly, Cloud Firestore is built on top of existing Google Cloud technologies, such as the SQL database solution Cloud Spanner (Kerpelman, 2019). Therefore, Firestore “offers

a strong consistency" (Kerpelman, 2019), so that the user can be sure to always receive the latest version of the data.

Assuming Cloud Firestore is based on Spanner, it also uses the 2-phase commit protocol (2PC) and the *TrueTime* technology of Google Cloud Spanner (E. Brewer, 2017). In addition, Cloud Firestore supports transactions that are applied only when all steps can be executed. If a transaction request fails, the database will perform a rollback and the transaction will be retried if necessary (Google, 2023c, 2023g). This is reinforced by synchronized clocks because each request or operation gets a server timestamp indicating the time when the server receives the request. This allows the requests to be put in chronological order (Google, 2023d). If a transaction is now requested, only operations with a timestamp before the transaction are considered. Thus, the transaction remains isolated from the rest of the database and is consistent (Google, 2023d).

Finally, considering all mentioned arguments, it can be taken into conclusion that Cloud Firestore is technically a CP-System, due to the strong consistency ensured by transactions and server timestamps and the partition tolerance, which is guaranteed by the multi-regional server structure

6.4 Installation and API access

Cloud Firestore is a cloud-based database solution, thus a local installation is not required. Therefore, the following section explains how to set up Cloud Firestore correctly and connect to it via a Python script.

6.4.1 Create Firestore Instance

Unlike other databases, Cloud-Firestore is not installed on a local system. Instead, Cloud Firestore is part of Google Cloud Firebase, a wrapper for different app and web development tools. To create a Cloud Firestore instance, a Firebase project is required. Therefore, follow the steps of creating a Firebase project in the Firebase Console (URL: https://console.firebase.google.com).

After creating the project, different services, like Cloud Firestore, can be added to the project. In the menu on the left-hand side, Firestore can be selected and added to the Project. Additionally, services like authentication, functions, or analytics can be activated to enrich the functionality of the application. Adding Cloud Firestore to the project, an initialization window will pop up.

Firstly, the user will be asked whether it wants to use the production or test mode of the database. In the production mode, all requests of third parties will be blocked. Instead, the test mode allows all requests in the next 30 days.

Secondly, the location of the database must be selected. Important is that the location should be in a minimal distance to the end-user of the system. In this chapter, the test

mode with the European location is chosen because the database should allow all requests and be located in Europe.

Finally, Cloud Firestore is now configured and documents can be created and rules and indexes defined through the online graphical user interface of the Firebase Console.

6.4.2 Connect to the Instance

Cloud Firestore can be controlled and managed by the Firebase Console by default. Instead, to use the database in an application, different SDKs exist for several programming languages like Python, Flutter, or JavaScript. In this section, the process of using the Python SDK and connecting to Cloud Firestore will be explained.

Firstly, the SDK must be installed on the local machine. For this, it is recommended to use pip, a package manager for Python. The pip command is shown in Listing 6.1 and installs the *firebase-admin* package.

```
pip install firebase-admin
```

Listing 6.1: Install the firebase-admin package via pip

Secondly, it is required to have the user credentials for Cloud Firestore. Therefore, a private communication key can be created in the Firebase Project Account Settings. By clicking on "create new private key", a JSON-file including the private key will be created.

Finally, to use the firebase-admin SDK, the credentials must be provided to the *firebase admin* object. Listing 6.2 shows the code in Python. First, it imports all required packages and classes, and then the credentials are loaded from the JSON-file and provided to the *firebase admin* object. The final object for interacting with the firestore instance itself is the client returned by the *firestore.client* method.

```
import firebase_admin

from firebase_admin import credentials
from firebase_admin import firestore
from google.cloud.firestore import Client

# Use a service account.
cred = credentials.Certificate('/path/to/file/account.json')

firebase_admin.initialize_app(cred)

db: Client = firestore.client()
```

Listing 6.2: Code for connecting with to a Cloud Firestore instance

6.5 Advantages and Disadvantages

Cloud Firestore has advantages and disadvantages differing from other NoSQL databases because of its hosting in the cloud. Some advantages and disadvantages are based on a project example implemented and presented in the presentation (Presentation-Link) of this chapter.

Firstly, Cloud Firestore has very loose references that have already been mentioned in chapter 6.2.3. Through these, it is possible that relationships between documents are preserved, even though one of these documents has already been deleted.

Combined with Firestore's second disadvantage of not providing explicit policies, the developer must implement their own rules and protections using code on the client side. An alternative can be Cloud Functions provided by Google to manage the constraints server sided.

Another disadvantage of Firestore is that it is hosted and provided by Google. Therefore, one might have concerns regarding the privacy of the saved data. Also, one is completely dependent on Google and their potential downtime, outages, or other technical issues. Although Google is known for its stability in this regard, it can still happen and thus limit the accessibility of the Firestore database.

The last disadvantage of Cloud Firestore relates to the complexity of the query structure. For example, a simple database query that retrieves an employee's name from the database requires significantly more code and knowledge in Firestore compared to the same query in SQL.

On the other hand, Firestore has advantages due to its cloud-hosting and integration into Google Firebase, which can be a good trade-off for the above-mentioned disadvantages. Furthermore, Firestore gets enhanced by additional functionalities of Firebase. For example, it is possible to enable access rules based on the authentication plugin of Firebase or to use Google Cloud, for saving bigger linked files.

Moreover, the Cloud Functions of Firebase can be used to implement a server-side logic of the Firestore. For example, Cloud Functions can be used to define functions, which will be executed if a new account is created, and the document is added to Cloud Firestore.

Furthermore, Firestore has a flexible document structure, which provides system flexibility in the creation of its data structure. It is possible to add not required fields to a document, after the structure of the document changes. This is a big advantage for smaller projects with a dynamic data structure in order to evolve flexibly in the future.

Another important advantage is that Cloud Firestore is a fully managed system due to its distributed server structure of regional and multi-regional servers. It can scale by itself and, as a result, the limit of free data storage is endless. This is again an essential advantage for smaller projects, aiming to grow in the future. They can even begin to use Cloud Firestore for free until the maximum number of free requests is used up.

Finally, Cloud Firestore is a fast, consistent and partition tolerant system, which is really significant for mobile applications, enforcing a low latency with high consistency and

partition tolerance. For example, a single request done in the context of this research takes around 50 milliseconds after the service was contacted once before.

6.6 Conclusion and Outlook

Cloud Firestore is a NoSQL database. It stores data in documents, which are grouped into collections. With subcollections, data can be structured as needed to represent complex systems (cmp. chapter 6.2). However, the central point of Firestore, which distinguishes it from most other databases, is its offering as a Cloud Service. The NoSQL database is a fully managed system in the Google Ecosystem, more precisely it is part of the Firebase network (cmp. chapter 6.2). This offers advantages such as strong authentication enabling and strong availability. On top of that, Google wants to make developing inside Firebase the easiest possible and makes a variety of SDK available (cmp. chapter 6.5).

Positioning Firestore in the CAP-Theorem returns a marker between consistency and partition tolerance. However, as the database offers nearly all-time-availability due to the Google's network, a user-perspective can also highlight the Availability-part and categorize Firestore as an AP-system (cmp. chapter 6.3).

To give an outlook, it is best to integrate an inspiration of the presentation held during the lecture. One student came up with an Open Source alternative to Firebase named Supabase. Supabase offers many of the core features of Firebase and really is moving upmarket (Lardinois, 2022). However, it is questionable, where Google can take advantage of its position in the market, its already high-end network and the trustworthiness of its customers and how Supabase, even though it is Open-Source, can compete with that power.

Sources

Brewer, E. (2017). Spanner, TrueTime & the CAP theorem.

Brewer, E. A. (2000). Towards robust distributed systems [Number: 10.1145 tex.organization: Portland, OR]. *PODC*, *7*, 343477–343502.

Dufetel, A. (2017, October 3). *Introducing cloud firestore: Our new document database for apps* [The firebase blog]. Retrieved March 10, 2023, from https://firebase.blog/posts/2017/10/introducing-cloud-firestore

Firebase. (2017, October). Introducing Cloud Firestore. Retrieved March 23, 2023, from https://www.youtube.com/watch?v=QcsAb2RR52c

Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP theorem [Conference Name: Computer]. *Computer*, *45*(2), 30–36. https://doi.org/10.1109/MC.2011.389

Google. (2020, January 9). *Firestore service level agreement (SLA)* [Google cloud]. Retrieved March 10, 2023, from https://cloud.google.com/firestore/sla

Google. (2023a). Supported data types | Firestore. Retrieved March 24, 2023, from https://firebase.google.com/docs/firestore/manage-data/data-types

Google. (2023b, March). Access data offline | Firestore. Retrieved March 23, 2023, from https://firebase.google.com/docs/firestore/manage-data/enable-offline

Google. (2023c, March 3). *Transactions and batched writes | firestore | firebase*. Retrieved March 10, 2023, from https://firebase.google.com/docs/firestore/manage-data/transactions

Google. (2023d, March 14). *Cloud firestore data model* [Firebase]. Retrieved March 15, 2023, from https://firebase.google.com/docs/firestore/data-model

Google. (2023e, March 14). *Firestore release notes* [Google cloud]. Retrieved March 15, 2023, from https://cloud.google.com/firestore/docs/release-notes

Google. (2023f, March 14). *Locations | firestore* [Google cloud]. Retrieved March 15, 2023, from https://cloud.google.com/firestore/docs/locations

Google. (2023g, October 3). *Transaction serializability and isolation | firestore* [Google cloud]. Retrieved March 11, 2023, from https://cloud.google.com/firestore/docs/transaction-data-contention

IBM. (2023). What is cloud computing? | IBM. Retrieved March 24, 2023, from https://www.ibm.com/topics/cloud-computing

Kerpelman, T. (2019, August 5). *"why is my cloud firestore query slow?"* [Firebase developers]. Retrieved March 10, 2023, from https://medium.com/firebase-developers/why-is-my-cloud-firestore-query-slow-e081fb8e55dd

Kerpelmann, T. (2017, October). Cloud Firestore vs the Realtime Database: Which one do I use? Retrieved March 24, 2023, from https://firebase.blog/

Lardinois, F. (2022, May 10). *Supabase nabs $80m for its open source firebase alternative* [TechCrunch]. Retrieved March 28, 2023, from https://techcrunch.com/2022/05/10/supabase-raises-80m-series-b-for-its-open-source-firebase-alternative/

7. Neo4j

by Marcel Alex, Edmund Krain, Simon Weiss

Graph-based databases have emerged as a powerful tool for data storage and retrieval. Unlike traditional relational databases, graph-based databases represent data in the form of nodes and edges, making it easier to represent and analyze complex relationships between data entities. In this paper, we aim to provide a comprehensive overview of Neo4j, a popular graph database. We will explore its features, query language, performance, and scalability. Furthermore, we will examine the CAP theorem and its application to Neo4j, followed by a use case example to illustrate its practical applications.

7.1 What are Graph based Databases?

Briefly, a graph based database is a combination of edges and vertices which can be used to represent real world objects and their relationships. It stores data in form of vertices which are called nodes and edges which are called relationships. Nodes and relationships can be connected to each other in a graph. The nodes are the objects and the relationships are the connections between the objects. Nodes and relationships can have properties. The properties can be used to store metadata about the node or relationship (Jordan, 2014, P. 6f.). You can think of a graph based database as a network.

A good example of a real life graph based database is a crime diagram (example Figure 7.1). In a crime diagram you can see the different people and their relationships to each other. The people are the nodes and the relationships are the connections between the people. (Kemper, 2015, compare P. 6f.)

1

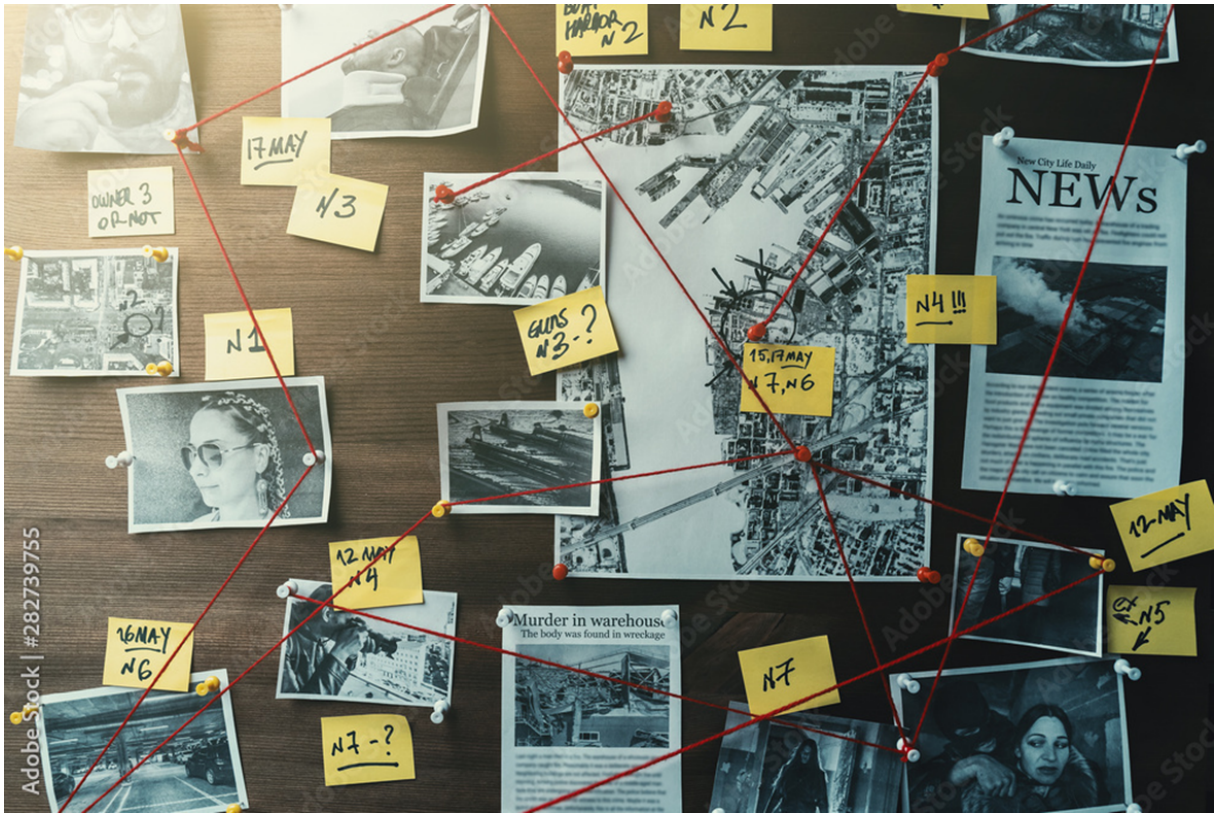


Figure 7.1: Crime diagram (Stock., 2020)

7.1.1 Nodes

Nodes can be seen as the objects in a graph based database. They can be tagged with labels (see section 7.1.4) to describe their different roles and tasks in the graph model. Nodes can also hold an arbitrary number of properties (see section 7.1.3). The following example in Figure 7.2 shows two nodes with the label "Person", the properties "name", "lastname" and "email" and their relationship to each other (Jordan, 2014, compare P. 6f.).

7.1.2 Relationships

Relationships provide connections between nodes. They are named and can also have properties like nodes. A single node can have multiple relationships to other nodes, these relationships can occur in any number or type without sacrificing performance (Neo4j, 2023e, compare).

7.1.3 Properties

Properties are key-value pairs that can be attached to a node or a relationship. Their purpose is to store metadata on them. They can hold different types of data like strings,

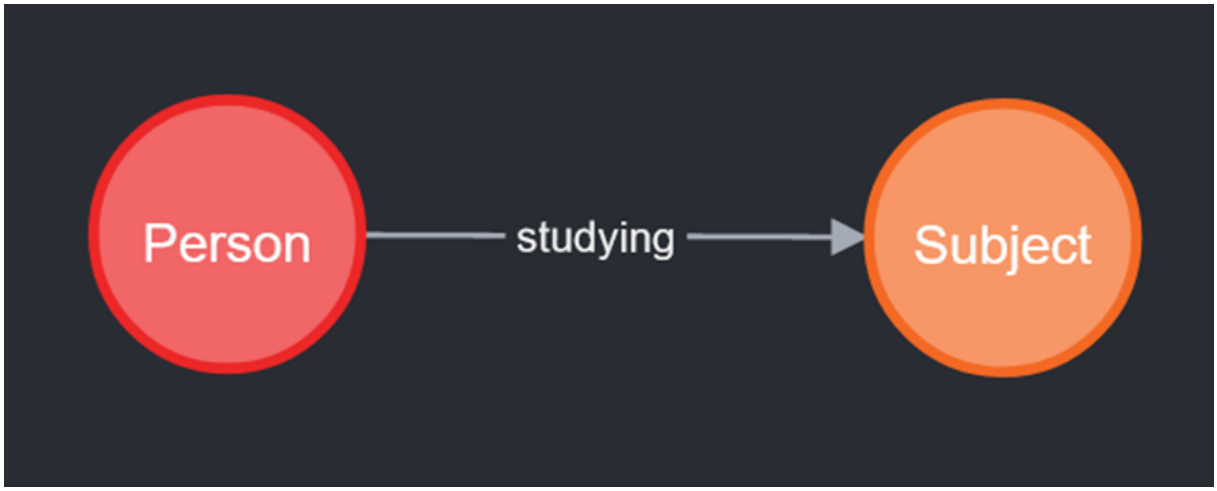


Figure 7.2: Node sample

numbers, arrays, booleans, dates, times, points, and even other nodes and relationships (Neo4j, 2023d, compare) (Neo4j, 2023b, compare).

7.1.4 Labels

Labels can be seen as a way to group nodes. Therefor a node with a specific label can provide additional values for each node. For example, if you add a label called ”Student” to a some nodes, they can have the same properties with different values. A good example for this is the following graph model (see Figure 7.3). Labels are often used to show common

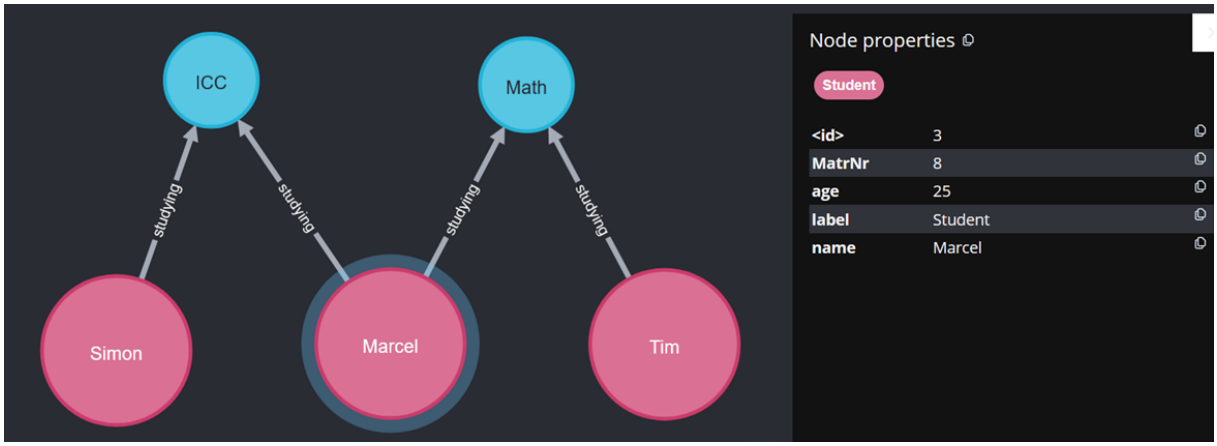


Figure 7.3: Label sample

subsets of essential node types. Labels also offer a way to enforce modeling constraints when they are necessary, as well as increasing the speed at which data can be accessed through improved indexing. (Jordan, 2014, compare P. 6f.)

7.1.5 Traversal and indexing

Traversal is a common way of querying a graph in order to find a specific answer to a question or more precisely to find a specific node or relationship. The traversal is done by visiting nodes and following their relationships according to a set of specified rules (Jordan, 2014, compare P.7). This can be done in all sorts of algorithms, which can be roughly divided into two categories: breadth-first and depth-first. The breadth-first search begins with all neighboring nodes of the starting node and then continues with all neighbors of the neighboring nodes. The depth-first search follows a path of the starting node until it reaches a dead end, then it backtracks and continues with the next path. Explicit examples of traversal algorithms are the Hamiltonian path, the Eulerian path and the Dijkstra algorithm (Schiele, 2019, compare). Indexing is also a way to find a specific node or relationship. It is a way to speed up the search for a specific node or relationship by using specific indexes on a labeled node or relationship (Jordan, 2014, compare P.7).

7.2 Brief History of Neo4j

Neo4j was founded in 2007 by Emil Eifrem and Johan Oskarsson and is written in Java. However, the fist usable version of Neo4j was released in 2010. It was the birth of the graph based Databases with nodes and relationships. The first version of Neo4j was called Neo4j Community Edition. Over the years, Neo4j has been developed and improved. In the following list you can see the different versions of Neo4j, their pros and improvements and their release dates . **Neo4j 1.0** Was released 2010

- It was the birth of graph based databases with nodes and relationships (Neo4j, 2022a).

Neo4j 2.0 Was released 2013.

- First version of Neo4j with a Cypher Query Language.
- It gave Neo4j a better interface.
- Addition of a REST API (N., 2015).

Neo4j 3.0 Was released 2016.

- Higher upper scaling limits.
- Faster access to data-graphs.
- Access to frameworks and Web interfaces (Neo4j, 2021a).

Neo4j 4.0 Was released 2019.

- Availability of multiple database at runtime in standalone.
- Schema-based security and Role Based Access Control.
- Memory constrain control for transactions (Neo4j, 2021b).

Neo4j 4.4 Was released 2021.

- Long term support for enterprise.
- Cypher Shell Enhancements.
- HTTP API with cloud-native interface (Neo4j, 2021c).

Neo4j 5.5 Was released 2023.

- Current version at the time of this paper.
- New community Libraries.
- More visualization tools
- And much more (Neo4j, 2023i).

7.3 Pros and Cons of graph based databases

In this section we will discuss the pros and cons of graph based databases.

7.3.1 Pros of a graph based database

- **Good for relationships** Modeling of relations is easy in a graph based database because you can implement them as you imaging them
- **Easy to scale** Another advantage of a graph based database is that the performance of a graph based database is not affected by the number of nodes and relationships, so it can grow year over year without any performance issues.
- **Flexibility** The structure of graph based databases are easy to edit and easy to extend without endangering the current functionality.
- **Agile** The last named advantage provides another big advantage of graph based databases, their flexibility leads to an advantage in Agile-teams, because the graph based database can grow with the tasks and goals of them.
- **You get what you see** Last but not least, in a graph based database you can see the functionality and relations between nodes by an instance without even thinking about it.

7.3.2 Cons of a graph based database

- **Not often used** Graph based databases are used much less than traditional databases. Therefore, it is sometimes difficult to get support, help, or answers to certain questions.
- **Slower** Graph based databases are significantly slower in processing transactional.
- **Security** They also have no security on the data level.

7.4 Neo4j Features

In this section, we discuss the features and capabilities of Neo4j, including its query language, drivers, scalability, and optimizations.

Neo4j uses a query language called Cypher, which is a declarative language designed specifically for working with graph data. Cypher provides an intuitive way to express complex graph queries using patterns and traversals. In the following section 7.4.1, we will explain the syntax and semantics of Cypher in detail.

One of the key features of Neo4j is its support for native drivers in most popular programming languages. This means that developers can interact with Neo4j using their preferred language, without the need for complex APIs or libraries. Native drivers are available for Java, Python, .NET, JavaScript, and many other languages. (Neo4j, 2022c)

Neo4j is designed to work in cloud environments, with a focus on both vertical and horizontal scalability. It can be deployed in a variety of cloud platforms, including AWS, Microsoft Azure, and Google Cloud Platform. Neo4j's architecture is also optimized for cloud-based deployments, with features like automatic failover and backup, multi-zone replication, and load balancing. (Neo4j, 2022c)

Neo4j employs a number of optimizations to efficiently handle complex graph data structures. One of these is in-memory graph processing, which allows Neo4j to store and manipulate graphs in RAM, resulting in significant performance gains. Other optimizations include caching, indexing, and compression. Compared to traditional relational databases, Neo4j outshines in query performance by orders of magnitude. This is because graph databases are specifically designed to handle complex relationships between data points, which can be challenging to model and query using traditional relational databases. Neo4j's performance advantage is especially evident when dealing with large, interconnected datasets. (Guegan, 2021)

To support large-scale deployments, Neo4j includes built-in support for clustering and replication. This allows Neo4j to seamlessly scale horizontally, by distributing data across multiple nodes in a cluster. Neo4j's clustering and replication features are designed to be highly available and fault-tolerant, ensuring that data is always accessible and consistent. (Neo4j, 2023f)

In conclusion, Neo4j is a powerful and efficient graph database that offers a number of unique features and capabilities. Its support for Cypher, native drivers, cloud deployments, optimizations, and clustering and replication make it an ideal choice for developers and organizations working with complex graph data structures.

7.4.1 Cypher

Cypher is the query language of Neo4j. It is a declarative language, which means that you describe what you want to achieve, but not how it should be done. As it is a graph query language, it is designed to work with graph data structures. On the first look it has similarities with ASCII-art.

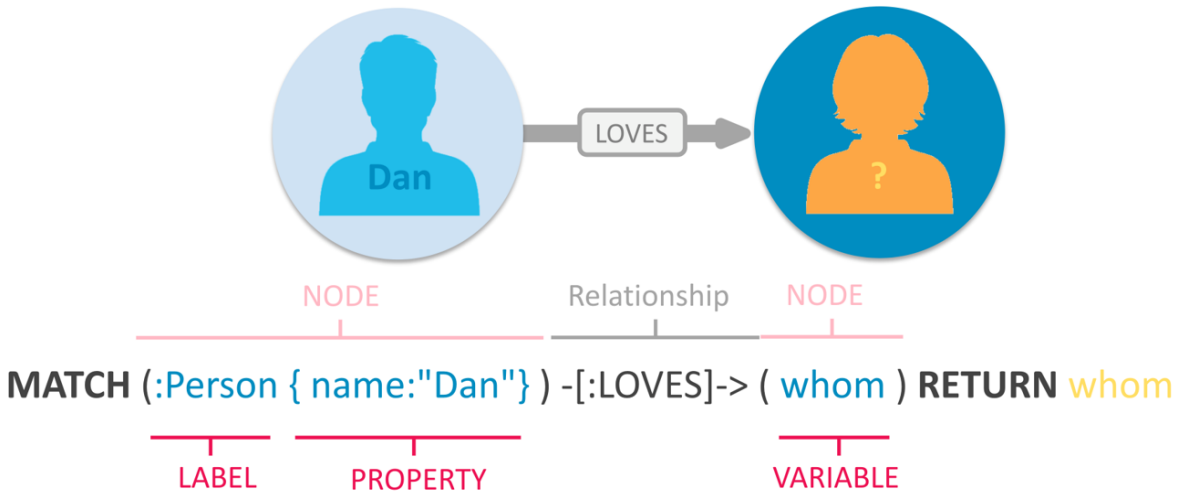


Figure 7.4: Cypher Query Language Basics (Neo4j, 2023h)

In the example in Figure 7.4 you can see the basic structure of a Cypher query. It starts with a "MATCH" statement, which is used to find the nodes and relationships you want to work with. Nodes are represented by round brackets and relationships by square brackets with ASCII-art like arrows to show the direction of the relationship. Inside those brackets you can specify the labels of the nodes and the type of the relationship. Additional properties can be specified with curly brackets. In this example the second node in the query has a variable name, which is used to refer to it in the following statements. After the "MATCH" statement you can specify the "RETURN" statement, which is used to return the results of the query. In this example the "RETURN" statement returns the variable name of the second node.

7.5 CAP Theorem

Neo4j is a popular graph database that supports the ACID properties of databases, including atomicity, consistency, isolation, and durability. These properties ensure that the database remains consistent and reliable even in the face of unexpected failures or errors ("What is Neo4j (Graph Dabatase)? Complete Overview of Neo4j", 2022). Neo4j maintains consistency through its built-in consistency checker, which helps to ensure that all nodes are in sync and that data is not lost or corrupted during updates or queries (Neo4j, 2023g).

Another major benefit of Neo4j is its high availability, which is achieved through horizontal scaling across a cluster of machines. This allows for seamless failover and replication, ensuring that the database remains up and running even in the event of node failures or other issues. One optional method to achieve horizontal scaling is data sharding, which involves partitioning the data across multiple nodes. However, without data sharding, there is only one partition, which can result in greater partition tolerance, but less availability in the event of a failure of a node or network partition. (Krishnan, 2020)

The choice of whether to use data sharding or not ultimately depends on the specific needs and priorities of the user. Furthermore, the placement of Neo4j on the CAP theorem can vary depending on the architecture used.

7.6 Using Neo4j

In this chapter we will explain how to install Neo4j in a Docker container and how to use it together with the Neo4j web interface. Afterwards a simple use case example will be used to show the possibilities of Neo4j and it's tools.

7.6.1 Installation

There are a lot of different ways to install Neo4j. Neo4j provides a lot of different options for installation. These options include Linux, macOS, Windows, Cloud, Kubernetes, and Docker (Neo4j, 2022b).

For this thesis the Neo4j Docker image will be used, because it is the easiest way to get started with Neo4j. Dockerhub provides an official Neo4j image. For starting Neo4j with Docker the following command (Listing 7.1) can be used:

```
docker run \
    --publish=7474:7474 --publish=7687:7687 \
    --volume=/path/to/your/data:/data \
    --volume=/path/to/your/logs:/logs \
    --env NEO4J_AUTH=neo4j/SecretPassword \
    neo4j:latest
```

Listing 7.1: Neo4j docker run command

The publish option exposes the (default) ports 7474 and 7687 to the host system. The volume options mount the host directories `/path/to/your/data` and `/path/to/your/logs` to the container. The environment variable `NEO4J_AUTH` sets the username and password for the website access. If a specific version of Neo4j is needed the tag `latest` can be replaced with the version number necessary.

After the container starts, Neo4j can be accessed via the browser at `http://localhost:7474`. On the first start Neo4j will ask a username and password, which are both set in the command above to `neo4j` and `SecretPassword`.

The following image shows the web interface of Neo4j after.

In general the Neo4j Docker image is an easy and fast way to get started with Neo4j. It is our recommendation to use the Docker image for testing and development (Neo4j, 2023a).

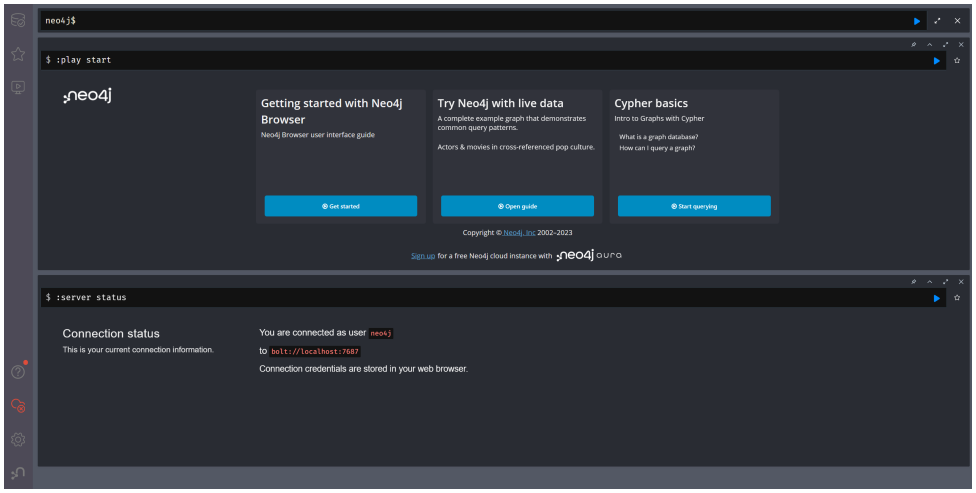


Figure 7.5: Neo4j web interface

7.6.2 Use case example

In this section a use case example will be created and analyzed. The example will be introduced as a SQL database and then be adapted to Neo4j. This will emphasize the differences between traditional relational databases and Neo4j.

The database describes the relationship between professors, students, and subjects.

SQL Tables

To get a better understanding of the Neo4j data model SQL tables will be used as the basis for the example. The SQL tables `professor`, `student`, `subject`, `professor_subject` and `student_subject` as seen in Table 7.1 are used.

Student ID	Student name
0	Tim
1	Bob
2	Anja

(a) Table students

Professor ID	Professor name
0	Müller
1	Mayer
2	Völlinger

(b) Table professors

Student ID	Subject ID
0	2
1	1
1	2
2	0
2	3

(c) Table student-subject

Subject ID	Subject name	Professor ID
0	WebDev	0
1	ICC	1
2	AWS	2
3	English	2

(d) Table professor-subject

Table 7.1: SQL tables

The `professor_subject` and `student_subject` tables are junction tables to connect the professors and students with the subjects. Even with this simple example it is already not easily possible to read the relations between the tables. For example if you want to know which student is taught by which professor you have to use at least three tables to get this information.

Neo4j

In comparison to the SQL tables the Neo4j data model is much easier to read. Relationships are directly visible and the direction of the relationship is clear. The following image Figure 7.6 shows the example data in the Neo4j web interface.

The web interface also allows the graph to be easily rearranged by dragging the nodes or edges around. This makes it easy to get a better overview of the graph. The direct integration of a Cypher query editor makes it easy to test queries and visualize the results without any additional tools.

7.7 Conclusion

Graph databases are a good have a very different data model than relational databases. Because of this a lot of common rules and best practices do not apply to graph databases. This leads to some advantages and disadvantages of graph databases. Probably the biggest advantage of graph databases is their performance with deeply nested data. On the other hand their performance lacks in traditional relational data.

Neo4j is not without reasons the most popular graph database (DB-Engines, 2023). As our brief history showed Neo4j is a very fast developing database. While there are some basic similarities between Cypher and SQL, like the use of `WHERE` clauses, both query

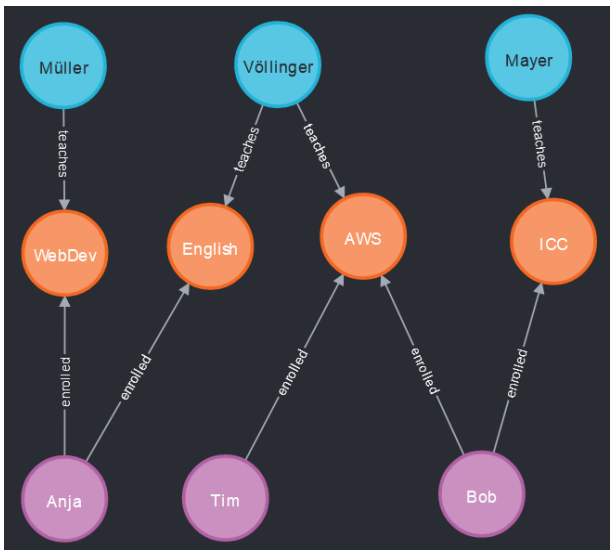


Figure 7.6: Neo4j data model

languages are mostly very different. The main difference is the graph like structure of the data. This leads to a very different way of thinking about the queries. While Cypher may be different to the more common SQL, after a short time it is very easy to get used to it. Another advantage of Neo4j is the great visualization of the data.

7.8 Recommendations

As mentioned above Cypher is very different to SQL, because of that we would recommend you to get familiar with the query language before you start using it in production. The official documentation is a good place to start learning the language (Neo4j, 2023c).

We wouldn't recommend replacing every relational database with a Neo4j. Instead there are three main use cases for graph databases:

- **Easy setup and debugging:** Especially with the official docker image Neo4j is easy to set up. The integrated Neo4j web interface is a good tool for debugging the database.
- **Visualization:** Neo4j is a good choice for visualization of data because of the graph structure. In comparison to relational databases the visualization of data is much easier. The graph like structure of the data is far more intuitive than the table structure of relational databases.
- **Deeply nested data:** Graph databases in general are a good choice for deeply nested data. In these cases they outperform relational databases in terms of performance and simplicity.

Sources

DB-Engines. (2023). *Db-engines ranking*. https://db-engines.com/en/ranking/graph+dbms

Guegan, B. (2021, September). *Neo4j Performance Architecture Explained & 6 Tuning Tips*. https://www.graphable.ai/blog/neo4j-performance

Jordan, G. (2014). *Practical neo4j*. Apress.

Kemper, C. (2015). *Beginning neo4j*. Apress.

Krishnan, K. (2020). *Building Big Data Applications*. Elsevier, Academic Press. https://doi.org/10.1016/C2017-0-03463-8

N., S. (2015, August). *Neo4j 2.0 ga – graphs for everyone. neo4j graph data platform.* https://neo4j.com/blog/neo4j-2-0-ga-graphs-for-everyone/

Neo4j, I. (2021a, June). *Neo4j 3.0.0 - neo4j graph data platform. neo4j graph data platform.* https://neo4j.com/release-notes/database/neo4j-3-0-0/

Neo4j, I. (2021b, June). *Neo4j 4.0.0 - neo4j graph data platform. neo4j graph data platform.* https://neo4j.com/release-notes/database/neo4j-4-0-0/

Neo4j, I. (2021c, December). *Neo4j 4.4 - neo4j graph data platform. neo4j graph data platform.* https://neo4j.com/release-notes/database/neo4j-4-4-0/

Neo4j, I. (2022a). *Graph database company | leaders in graph technology | neo4j. neo4j graph data platform.* https://neo4j.com/company/

Neo4j, I. (2022b, November). Installation. https://neo4j.com/docs/operations-manual/current/installation/

Neo4j, I. (2022c, November). Neo4j Graph Database. https://neo4j.com/product/neo4j-graph-database

Neo4j, I. (2023a). *Docker, introduction - operations manual*. https://neo4j.com/docs/operations-manual/current/docker/introduction/

Neo4j, I. (2023b). *Graph database concepts*. https://neo4j.com/docs/getting-started/current/appendix/graphdb-concepts/

Neo4j, I. (2023c). *The neo4j cypher manual v5*. https://neo4j.com/docs/cypher-manual/current/

Neo4j, I. (2023d). *Values and types*. https://neo4j.com/docs/cypher-manual/current/syntax/values/

Neo4j, I. (2023e). *What is a graph database?* https://neo4j.com/docs/getting-started/current/get-started-with-neo4j/graph-database/#:~:text=A%20Neo4j%20graph%20database%20stores,thinking%20about%20and%20using%20it.

Neo4j, I. (2023f, March). Clustering - Operations Manual. https://neo4j.com/docs/operations-manual/current/clustering

Neo4j, I. (2023g, March). *Consistency checker - Operations Manual*. https://neo4j.com/docs/operations-manual/current/tools/neo4j-admin/consistency-checker

Neo4j, I. (2023h, March). Cypher Query Language - Developer Guides. https://neo4j.com/developer/cypher

Neo4j, I. (2023i, March). *Neo4j 5 - neo4j graph data platform. neo4j graph data platform.* https://neo4j.com/release-notes/database/neo4j-5/

Schiele, V. (2019). *Graph database systems*. https://jupyter-tutorial.readthedocs.io/en/stable/data-processing/nosql/graph-db.html

Stock., A. (2020). *Detective board with photos of suspected criminals, crime scenes and evidence with red threads, toned stock-foto.* https://stock.adobe.com/de/images/detective-board-with-photos-of-suspected-criminals-crime-scenes-and-evidence-with-red-threads-toned/282739755

What is Neo4j (Graph Dabatase)? Complete Overview of Neo4j. (2022, December). https://www.graphable.ai/software/what-is-neo4j-graph-database

8 Conclusion

After dealing with all the different NoSQL databases, this chapter is about summarizing the most important aspects.

NoSQL is an umbrella term for a variety of different kinds of databases. Examples of such database systems include key-value stores, document-oriented databases, wide column databases and graph databases. Historically, SQL databases were created with the aim of being a "one size fits all" solution for all applications. Although this might have been realistic at the time of their inception, nowadays such a solution is infeasible due to the sheer variety of use cases for a database ranging from storing highly structured data to completely unstructured data and from low to enormous amounts of data. Most NoSQL databases on the other hand are designed to be a perfect fit for some use cases while accepting that this makes them a terrible choice for most other use cases. Due to that, developers need to choose the right database for their needs and may need to use multiple different databases in a single application.

From the different databases discussed in this book, some application specific recommendations are given below: Graph databases are well-suited for databases with many relationships. CloudFirestore is a fully hosted document database managed by google. It offers great integration, especially in the world of mobile development. An open-source alternative would be MongoDB. MongoDB is another document database and a great choice, if the application has large amounts of different or changing data models. Use wide column databases like Cassandra if your system must deal with large amounts of unstructured data. Use key-value stores if you need a fast data access of data. With regards to the CAP theorem, we have shown that scalable NoSQL databases cannot forfeit partition tolerance and are therefore either AP or CP systems.

The book compared the structured approach of SQL databases with the unstructured approach of NoSQL databases. Furthermore, it named specific advantages and disadvantages of popular representatives for the different NoSQL databases. The goal of this paper is to ease the search for a suitable database for given requirements or specific architectures.

8.1 Outlook

As data continues to grow in importance in our daily lives, the amount of data being generated is growing exponentially, much of which is unstructured data. This has led to the need for more flexible databases, such as NoSQL databases. With the rise of big data, more data is being stored in the cloud, which has further increased the popularity of

NoSQL databases. The fact that many NoSQL databases are open-source projects also makes them more appealing, as there is a move towards more open-source databases.

Sometimes SQL databases are still preferable, particularly when dealing with highly structured data. In the future, we can expect continued development and improvement of NoSQL databases, with efforts to address some of their limitations, such as improving transaction support and standardization. Additionally, we may see further advancements in the integration of NoSQL databases with other technologies, such as big data platforms and cloud computing. As organizations increasingly adopt digital technologies and generate larger amounts of data, NoSQL databases will continue to play a key role in managing and processing data efficiently and effectively.

8.2 Closing Remarks

In summary, while NoSQL databases have a common approach to handling big data and providing flexible schema designs, they differ significantly in their placement on the CAP theorem, which represents their strengths and weaknesses in terms of consistency, availability, and partition tolerance. As a result, selecting the right NoSQL database for a particular use case requires careful evaluation of its unique features and characteristics. However, NoSQL databases have proven to be a valuable alternative to traditional SQL databases and organizations can be confident that there is almost always a NoSQL solution that will meet their data storage needs.

Sources - Introduction and Conclusion

Brewer, D. E. A. (2000). Towards robust distributed systems.

solid IT GmbH. (2023, March 28). *DB-engines ranking* [DB-engines]. Retrieved March 28, 2023, from https://db-engines.com/en/ranking