Hazelcast

by Alexander Edinger, Jonas Eppard and Fernand Hoffmann

In recent years, many areas have been changed by digitalisation. The digitalisation 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.

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)

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)

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 Qbjects. 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)

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)

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)

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.

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)

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)

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:

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

For the following example the docker installation will be used.

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. 1. The data

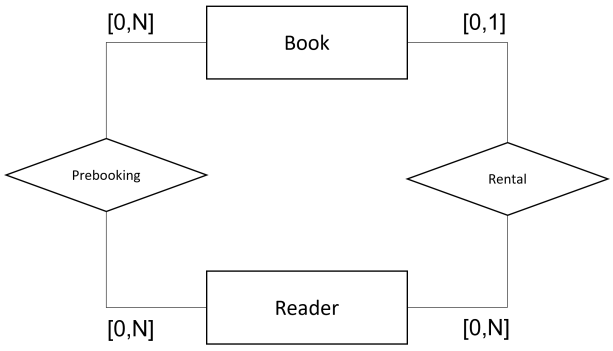

Figure 1: Example data structure (see also Mairhofer, 2021)

structure models a library with books. 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 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 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.

References

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