UNIVERSITÉ LIBRE DE BRUXELLES

ÉCOLE POLYTECHNIQUE DE BRUXELLES

Implementation of the YCSB benchmark for *etcd* and *Hazelcast*

ADVANCED DATABASES

Fall 2022

Authors:
Aliakberova, Liliia,
Gepalova, Arina,
Lorencio Abril, Jose Antonio,
Mayorga Llano, Mariana

Professor: **Zimányi, Esteban**

Contents

List of Figures

List of Tables

Listings

Abstract

In this report we explain our work for the project of the course *INFO-415: Advanced Databases*, in which we have to choose a NoSQL technology and two tools that implement it, understand their internals and perform a benchmark of both of them.

In Chapter 1 we give an overview on NoSQL databases, focusing on our chosen technology: Key-Value store. We end this introductory chapter explaining a common algorithm used for consistency maintenance among distributed databases: RAFT Consensus Algorithm. We also detail how our chosen tools, *etcd* and *Hazelcast* make use of this algorithm.

In Chapter 2 we explain how *etcd* works, both conceptually, logically and physically. We assess its main characteristics and objectives. We do the same with *Hazelcast* in Chapter 3.

We continue explaining the standard benchmark that we have used to assess the performance of our two tools, YCSB, in Chapter 4. In this chapter we explain the design and ideas behind this benchmark tool, as well as its architecture and how it can be extended. This extensibility property is the one that allowed us to perform the benchmarking of our two tools, that were not implemented into YCSB before. This implementation is explain in detail in Chapter 5.

Chapter 6 is devoted to explain an imaginary use case that we made up to assess the performance of our tool. This chapter is interesting because we show how YCSB is extensible also in terms of the use cases modeled.

The results and presented and analyzed in Chapter 7, in which we present the tests performed and our insights about them for three different workloads, including the customized one. In this section we demonstrate that *etcd* is suitable for persistent non changing data and *Hazelcast* is, contrarily, suitable for in-memory fast changing data.

To finish the report, we state some brief conclusion thoughts and a table summing up the characteristics of both tools, in Chapter 8.

All the work done can be directly accessed via our GitHub repository: *Implementation of the YCSB benchmark for etcd and Hazelcast*.

Chapter 1

Introduction

1.1 NoSQL databases

The SQL paradigm is backed up with a consistent theory and many technological and engineering designs developed along 50 years since its invention. Undoubtedly, SQL provides us with a set of rules and operations that can model most situations regarding data storage and querying. Nonetheless, SQL also puts restrictions in what can be done, and specially in how it should be done. This restrictions make it complex to apply the SQL theory in some applications, mainly because the natural form of the data is not tabular, but rather an inherently different one. For example, a typical dataset with features in the columns and records as rows is naturally modeled with SQL; but the case is different if we think of tree-structured data: even if we can model a tree as a SQL table or set of tables, a native tree-like structure would make better use of the characteristics of the structure of the data.

Thus, if the most restrictive aspect of SQL is the need to adhere to the precise data model of tables and relations, then it is natural to try to develop new data models that can cope with a wider variety of data structures. For instance, graph databases are specially good dealing with graph-like data. In [JA13], a comparative study between graph databases and relational databases is done, concluding faster performance as well as increased flexibility.

Another type of NoSQL database is Key-Value Stores, which are the main topic of the present work and are explained in the next section.

1.2 Key-Value stores

Key-value databases, are certain type of NoSQL databases, which employ a schemaless data model. Stored data is represented in form of key-value pairs, where values can be arbitrary binary objects, which makes it the most flexible data store from application perspective. Thanks to this simple structure, key-value stores can offer very high performance in comparison to traditional SQL databases as well as other types of NoSQL databases.

The schemaless property makes key-value stores really flexible in terms of the structure of the data to be stored, but every data stored needs to be uniquely identified by its key, which reduces the applicability of the model. For instance, the data need to have an identificator property, or at least it should be possible to artificially add one.

According to [SU09], the data retrieval speed of key-value stores is higher than that of SQL databases,

Figure 1.1: KV-Store conceptual structure.

and also the code tends to look cleaner, specially in those cases where the data is structured as a key-value store, but in a relational database, which makes it sometimes complex from a design point of view.

The conceptual structure of the data in a key-value store is as simple as in Figure 1.1. The key is usually a string which identifies the data stored in the corresponding value, which can be as complex as needed.

Key-value stores become particularly interesting in cloud architectures, in which availability is many times preferred to consistency, which are the C and A of the CAP theorem[1]. The partition tolerance is usually considered a must in distributed architectures (what would be the point of distributing without partitioning?), so the decision is to choose between C or A. As can be read in [Liu+15], many key value store provide high availability levels, even though they can only ensure softer forms of consistency. The default implementation of *Hazelcast* is one of this kind.

However, there are also key-value stores focused on consistency, rather than in availability, because sometimes simply consistency is a requirement of the problem to be solved. Two examples of this kind of key-value store are *etcd* and *Hazelcast CP Subsystem*, which make use of the RAFT consensus algorithm to ensure consistency among different clusters, while maintaining a reasonable availability.

1.3 RAFT Consensus Algorithm

The RAFT Consensus Algorithm was introduced by Diego Ongaro and John Ousterhout in [OO14] in 2014. Since then, the algorithm has been refined and improved, as well as implemented by different technologies, such as the two of our concern, *etcd* and *Hazelcast CP Subsystem*. The technologies also introduced some changes to the algorithm. We are going to explain both the basic algorithm and the modifications made by the two tools.

The first thing to know about RAFT is that it assigns a role to each node by labeling them, which will define its behavior in the system. 'Leader', 'Follower' and 'Candidate' are the main roles in this algorithm, however, some variations may include additional labels. As an example, etcd introduces the Learner role, which is described in section 1.3.1.

Additionally, RAFT can be divided into two iterative cases: when we have a Leader node and when RAFT has to assign one. When there is no leader or when new nodes are added or removed from the system, RAFT enters an 'Election' phase where all the remaining nodes change their label to 'candidates' and are voted by each other to be the new leader based on availability. A node only does not accept a leader in two scenarios: if this itself is more updated than the proposer, and when a new leader has not been elected yet by most members.

This possibility of entering into election phases provides high availability since the system can keep

[1]The CAP theorem states that a distributed database can only provide two characteristics out of Consistency, Availability and Partition tolerance. It was proved by Brewer in [Bre00].

working without information loss even when multiple nodes present failures. Formally, the system must have $n = 2f + 1$ replicas in the system, on which up to f replicas can fail.

Finally, a new leader is elected when it is accepted by a majority of replicas, and subsequently, the rest of the nodes are labeled as 'followers'. Once the leader is elected, its objective is to track active nodes and guarantee that all of them share the same information. To achieve this, the leader is constantly sending a pin, also called a "heartbeat", to the rest of the nodes to validate if they are active. When a node receives a pin from the leader, it answers to confirm it is still alive.

RAFT's main objective is to achieve consistency among a system of multiple clusters. This implies that any change done in one of them must be replicated along the entire structure. For this, if the leader receives a request to update its data, it will send the information to the rest of nodes, who will update their information and sent it back to the leader as a change confirmation. The leader itself does not update its value until the majority of followers have done it. This way of decision-making is why RAFT is considered to be a consensus algorithm.

1.3.1 RAFT modifications for *etcd*

Previously, *etcd* availability was affected when a restarting node has not received the leader's heartbeats in time, which triggered the election phase. However, to ensure better availability with rejoining nodes, *etcd* now adjusts election by providing more time for nodes to answer implying that it is prioritizing availability over response time.

Additionally, in previous versions of *etcd*, when a new *etcd* member joined the cluster with no initial data, it requested all historical updates from the leader until it catched up to the leader's logs making leader's network more likely to be overloaded, blocking or dropping leader heartbeats to followers. In such cases, a follower may had experienced election-timeout and started a new leader election. As a result, a cluster with a new member was more vulnerable to leader election risking cluster availability.

In order to address such failure modes, *etcd* introduced a new node label called "Learner", which joins the cluster as a non-voting member until it catches up to leader's logs. This means the learner still receives all updates from leader but it is not considered as a follower yet, so the Leader does not expect a response from it until the Learner has all the information needed to become a Follower and starts sending heartbeats confirmations.

1.3.2 *Hazelcast CP Subsystem*

Hazelcast also provides an implementation of the RAFT Algorithm, even though it is not in its default configuration, but rather in the *Hazelcast CP Subsystem*, which clearly alludes to the C and P in the CAP theorem.

In this subsystem, the data structures run in **CP groups**, each of which elects its own RAFT leader and runs an independent RAFT algorithm. By default, 2 CP groups are used:

- The first one is the **METADATA CP group**, responsible of managing the rest of CP groups and members. It is initialized during startup if we set up the CP subsystem to work.
- The second one is the **DEFAULT CP group**, which is a usual group of members managing data in the scope of the RAFT algorithm.

More groups can be added manually. This way, each CP group manage its internal members using an independent RAFT algorithm instance, and the set of all members is managed by the METADATA

CP group.

Chapter 2

etcd

In this chapter, we will briefly comment the main aspects of *etcd* and we refer the reader to the official *etcd* documentation, [Aut21a] for further details.

etcd is an open-source, key-value database written in Go language, currently developed under Cloud Native Computing Foundation. The name *etcd* comes after the Linux directory 'etc', where configuration files are stored, plus a 'd' from distributed. This name is a statement of the objective of the designers of *etcd*: they wanted to design a distributed way of storing configuration information.

etcd offers strict data consistency, high availability and performance. For this reason, *etcd* is mainly used as a tool in distributed systems, a topic in which we will elaborate in the subsequent section followed by a brief explanation of the RAFT algorithm, which is used by this tool.

2.1 Introduction

A **distributed system** is the one that uses resources located on different computers connected through a network. Components interact with each other in order to achieve a common goal. A distributed system needs a reliable coordinator who tracks the changes in each node and communicates the same to the entire cluster in a timely and reliable manner.

The most challenging task in distributed systems is to ensure data consistency across all the distributed nodes and manage the state of each node in a distributed cluster. Here is where *etcd* takes place, since this tool was developed to fulfill the need in distributed systems of using a single source of information which stores all the critical data that is needed to keep the distributed functionality properly.

Its objective is to provide support to the platforms to achieve coordination amongst all of its clusters and pods by managing containerized workloads as it scales, by taking care of the process across all the clusters.

etcd uses the Raft algorithm for management of highly available replicated log, being able to tolerate node failures. Furthermore, it offers dynamic cluster membership reconfiguration and enables distributed coordination by implementing primitives for distributed locking, write barriers and leader election, as we explained in 1.3.

2.2 The data model

etcd is designed to reliably store infrequently updated data and provide reliable watch queries. *etcd* exposes previous versions of key-value pairs to support inexpensive snapshots and watch history events.

etcd stores data in a multiversion persistent key-value store. This means that when data is updated, past values are not deleted, but stored as a snapshot in time of what we had, which will still be accessible. To prevent the data store from growing indefinitely over time and from maintaining old versions, the store may be compacted to shed the oldest versions of superseded data.

The **logical view** of the *etcd* store is a flat binary key space with a lexically sorted index on byte string keys, making range queries inexpensive[1].

The store keeps **revisions**. When created, the initial revision is 1. This value increases whenever an atomic mutative operation (which can consist of a set of different operations). Data from older revisions is accessible, but the current data is considered to be that of the newest revision.

Something similar is done with keys, maintaining a **version**, which increases with each change of the key and becomes 0 when the key is deleted from the store.

Regarding the **physical view**, the data is stored as key-value pairs in a persistent **b+tree** structure. Each revision only stores the deltas from the previous revision, decreasing the needed space for them. *etcd* also keeps a btree in memory to speed up range queries over keys.

2.3 Advantages of using etcd

Among the benefits of using *etcd* for managing distributed workloads are:

- **Availability** — By managing hardware failures and network partitions easily. *etcd* is designed to tolerate various system and network faults. By design, even if one node goes down, the cluster "appears" to be working normally, by providing one logical cluster view of multiple servers. So, even though we know for the CAP theorem that it cannot ensure total availability, *etcd* does a good work in this matter.
- **Replicability** — Each node in the *etcd* cluster can access the entire data store.
- **Performance** — Up to 10,000 writes per second.
- **Consistency** — By returning the most updated data value during each operation by using RAFT algorithm, as explained in Section 1.3.
- **Security** — It supports configurations for automatic TSL and SSL, as well as providing the possibility to assign roles for access and privilege control.
- **Fault Tolerance** — Allowing the system to remain functional as long as more than half of its nodes are still active.
- **Simplicity** — Allows using HTTP/JSON tools, any web application can perform read–write operations.

[1] This will make *etcd* perform well in the scan operation of YCSB, in contrast to *Hazelcast*.

Chapter 3

Hazelcast

This fourth chapter will focus on the explanation of the *Hazelcast* functioning, architecture and implementation. In this case, the reader is referred mainly to the *Hazelcast IMDG* (In-Memory Data Grid) docs [Aut21b], in which the key-value store capabilities of *Hazelcast* are detailed.

3.1 Introduction

Hazelcast is more than a key-value store, presenting other data storage models, but it implements a strict key-value store with the interface *Hazelcast IMDG* and, more concretely, with the IMap data structure. Thus, it will be in this interface in which we will focus our benchmarking.

Hazelcast IMDG is an open source in-memory object store. This means that *Hazelcast* tries to store in RAM as much information as it can, which translates into fast querying. Note that it is capable of storing data using different models, not only the key-value store model. The key-value store model is implemented by means of the **Map** interface[1]. Note also that persistence mechanisms can be configured, so that data of one cluster is not lost after shutting it down, but the focus of Hazelcast is not this, neither consistency, in principle. *Hazelcast* aims to be used by distributed applications as their data store, focusing on scalability and high availability. It is developed in Java and there are official clients in many other languages, such as REST, Python or Go. This increases the attractive of the tool for a wider variety of use cases.

Hazelcast uses the concept of cluster to describe the different groups of members in the net. Members interact with each other, forming a distributed network, storing and processing the data. When a new member is added, it automatically discovers the cluster and linearly increases memory and processing capacity. Members maintain a TCP connection between them through which communication is performed. The leader node is set to be the oldest one in the network, which means that this node is the one whose data is considered real and the one who is intended to serve queries. To maintain a relaxed consistency, data is replicated through the different members in the cluster and high availability is ensured by setting automatically the second oldest member as new leader when the current one goes down. The possibility to provide consistency is oferred by means of the CP Subsystem, which uses the RAFT Consensus Algorithm[2] and needs to be intentionally used by the developer: it is not a default feature of *Hazelcast*.

[1] Other data structures are implemented, such as lists, queues, sets... But these are, of course, not representative of a strict key-value store, so they will be excluded from our analysis.

[2] See 1.3

3.2 Topology

A *Hazelcast* cluster can be deployed in two different ways:

- **Embedded**: members include both the application and *Hazelcast* data and services. It is suitable for asynchronous or high performance computing. A visualization of this topology can be seen in Figure 3.1 (left).

- **Client/Server**: in this topology, *Hazelcast* data and services are centralized in one or more servers, which are accessed by the applications through the client API. This topology presents a more predictable and reliable performance, as well as better scalability. This topology is depicted in Figure 3.1 (right).

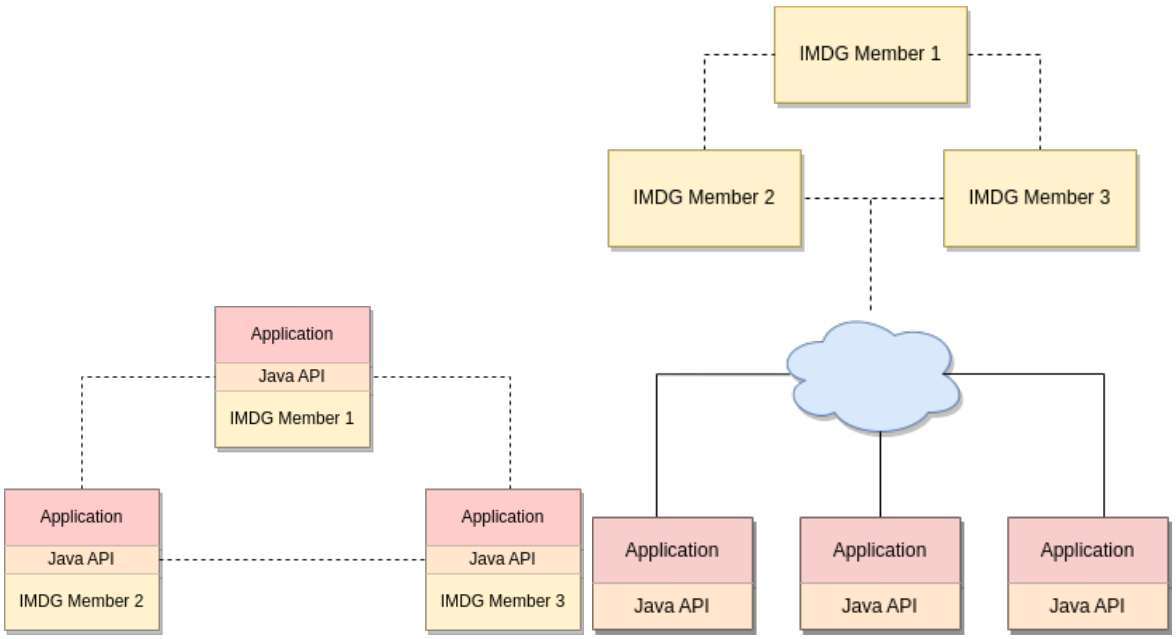


Figure 3.1: Topologies of *Hazelcast IMDG*: embedded (left) and client/server (right).

3.3 Data Partitioning

The memory segments in *Hazelcast IMDG* are called **segments**, whose size is limited by the capacity of the system. The partitions are distributed equally among the members of the cluster, as well as backups of the partitions. By default, for each partition, a single replica is created, but the amount can be increased. One of the replicas will be the primary one, and the others will be backups. The member with the primary partition is called the **partition owner**.

The default configuration works with 271 partitions, which are distributed equally among the members. This means that each member will possess a similar amount of primary replicas. For example, if there are 3 members, one would have 91, and the other two will have 90 partitions each.

The data is partitioned using a hashing algorithm which we are not detailing here.

3.4 *Hazelcast IMDG* as a key-value store: IMap

The *Hazelcast* Map, IMap, is an implementation of the Java interface *java.util.concurrent.ConcurrentMap*, which is the distributed implementation of a Java map. This implies that the typical Map methods of Java are directly used, and also that we can use arbitrary objects as keys and values.

An important technical aspect of the implementation is that the keys are hashed to improve throughput. This has the downside that querying *Hazelcast* using properties of the keys is very costly, because all the keys needs to be checked until the desired conditions are fulfilled. This makes it impossible, as we will see, to implement the whole YCSB benchmark efficiently, because a scan operation based on key prefixes is needed.

3.5 Advantages of using *Hazelcast*

The main strengths of *Hazelcast* as a key-value store is:

- It is open source.
- It can be used with a simple installation of Java, without further software requirements.
- It uses in-memory storage.
- It is peer-to-peer: all members are connected and are functionally the same.
- Scalability can be performed dynamically as the requirements of the application increase.
- It is resilient to failure: backups are distributed across the cluster. If an stronger consistency is required, the CP Subsystem can be used.

Chapter 4

YCSB

The *Yahoo! Cloud Serving Benchmark* (YCSB) was developed by Yahoo! Research and was first exposed to the public in 2010, as a release paper by Cooper et al., [Coo+10]. In this paper, an overview of cloud systems is given to justify the need for a benchmarking tool and YCSB is presented as a solution for this problem. The metrics that YCSB measures are explained as well as the different workloads that are used. The architecture of the benchmark is detailed, remarking on its extensibility. Finally, they showed the results that were obtained when applying this tool to benchmark four NoSQL systems, namely *Cassandra*, *HBase*, *PNUTS* and a sharded version of *MySQL*.

In this chapter, we will review this paper to get a profound understanding of the benchmark that we are going to deal with in this project. However, we will not show their results regarding the four systems analyzed by them, as we will do our own analysis for *etcd* and *Hazelcast*.

4.1 Introduction to YCSB

We know the importance of benchmarking in the field of cloud systems, and it is because of this importance and the fact that in the company Yahoo! they needed to study which tools they wanted to use for different applications that they decided to develop a benchmark that would serve them for this purpose, but also for anyone with similar needs.

More precisely, their **goal** when developing YCSB was to *"create a standard benchmark and benchmarking framework to assist in the evaluation of different cloud systems, focusing on serving systems, which are systems that provide online read/write access to data"*. Thus, they developed YCSB as an open source benchmark, keeping one eye on its extensibility, so it could be easily extended to newer and different technologies that could arise later.

Currently, YCSB is mainly implemented for experimental evaluation regarding distributed data management as well as for studies aiming to investigate the performance behavior of certain system or protocol designs.

4.2 Benchmark tiers

A **tier** is a series of workloads and metrics that focuses on one particular aspect to be measured. They proposed two tiers to evaluate the performance and the scalability of a given system.

4.2.1 Tier 1 - Performance

This tier measures the latency[1] of the requests when the database is under load. The YCSB developers argue that there is a tradeoff between latency and throughput[2], and define a system to have better performance when it is able to achieve the desired latency and throughput using fewer servers. Thus, the Performance tier objective is to characterize this tradeoff, measuring latency as throughput increase. To achieve this goal, they developed the **workload generator**, which:

- Defines the dataset and loads it into the database.
- Executes operations against the dataset while measuring performance.

This is implemented as the **YCSB Client**.

4.2.2 Tier 2 - Scaling

This tier examines the impact on performance as more machines are added to the system. This is done in two different scenarios:

- Scaleup: this measures performance as the number of server increases. To do this, different setups are measured, increasing each time the number of servers and the amount of data in the same proportion. A system with good scaleup properties should maintain a constant performance.
- Elastic speedup: this setup tries to measure the elasticity of the system, i.e., how well it behaves when adding new servers as the workload is running. A flexible system should improve its performance as the number of servers increases, because data remains constant but computing resources increase. The reconfiguration phase after a new system is added should be as short as possible.

4.3 Benchmark workloads

In the YCSB framework, a **package** is a collection of related **workloads**, which represent a particular set of read/write operations, data sizes,... and are used to evaluate systems at a particular setup. Thus, a package is intended to examine a broader range of setups, providing more information about the performance of the system.

4.3.1 The core package

The **core package** is the basic benchmark package provided by YCSB, developed with the goal of examining a wide range of workload characteristics. The workloads in the core package are variations of a same application type, which consists in:

- A table of records, where:
  - Each record has F fields.
  - Each record is identified by a primary key, which is a string lie 'user342314'.
  - Each field is named *fieldn*, $n = 1, ..., F$.

[1] Latency is a synonym of response time.

[2] Throughput refers to the number of transactions produced over a certain period of time. Thus, the mentioned tradeoff becomes obvious: the higher the latency, the lower throughput can be provided.

  - The values of each field are a random string of ASCII characters of length L.

- Operations chosen randomly from s given distribution of the following operations:
  - **Insert** a new record.
  - **Update** a record replacing the value of one field.
  - **Read** a record, either a randomly chosen field or all of them.
  - **Scan** records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen.
  - **Delete** a single record.

4.3.1.1 Predefined Workloads from YCSB

There are official predefined workloads for YCSB which differentiate from one another in terms of the percentage of execution of each of the operations to be performed. These predefined workloads are described below:

- **Workload A: Update heavy workload -** This workload is focused on reading (50 percent) and writing (50 percent), intended to simulate the process for an update requirement. However, updates in this workload do not presume you read the original record first, since the assumption is all update writes contain fields for a record that already exists.
- **Workload B: Read mostly workload -** This scenario has a 95 percent reads against 5 percent write operations. As in Workload A, these writes do not presume you read the original record before writing over it.
- **Workload C: Read only -** For this workload, functionality is exclusively focused on reading. No other operation is done for this workload.
- **Workload D: Read latest workload -** Given this workload, new records are inserted (5 percent), as well as read (95 percent) .
- **Workload E: Short ranges -** Short ranges of records are queried, instead of individual records. Scanning consists on 95 percent of the operations done and insert involves the remaining 5 percent.
- **Workload F: Read-modify-write -** In this workload, the client will read a record, modify it, and write back the changes. This is the usual case for a user database request. This workload forces a read of the record from the underlying datastore prior to writing an updated set of fields for that record. A random delta is used for the write, rather than some value derived from the current record.

Nevertheless, a customized workload can be performed as well by modifying the operations to execute and their respective percentage. In this specific benchmark, for both *etcd* and *Hazelcast*, the predefined workloads A and E were performed, as well as an additional customized workload. The details regarding the custom workload can be found in Chapter 6.

All of these workloads use the same dataset. Therefore, for this case, the database was loaded before the execution of the tests, which means that for each workload and workload size, we obtain measures

for both the load phase and the run phase.

4.4 The Benchmark Tool

4.4.1 Architecture

The main elements for this benchmark's architecture can be appreciated in the Figure 4.1. We have the YCSB Client, which receives the information needed from the workload parameter file and from the command-line parameter. After that, the database client connects to the clusters of the database being tested (*etcd* or *Hazelcast*, in this case), runs the operations, and records the statistics of the results.

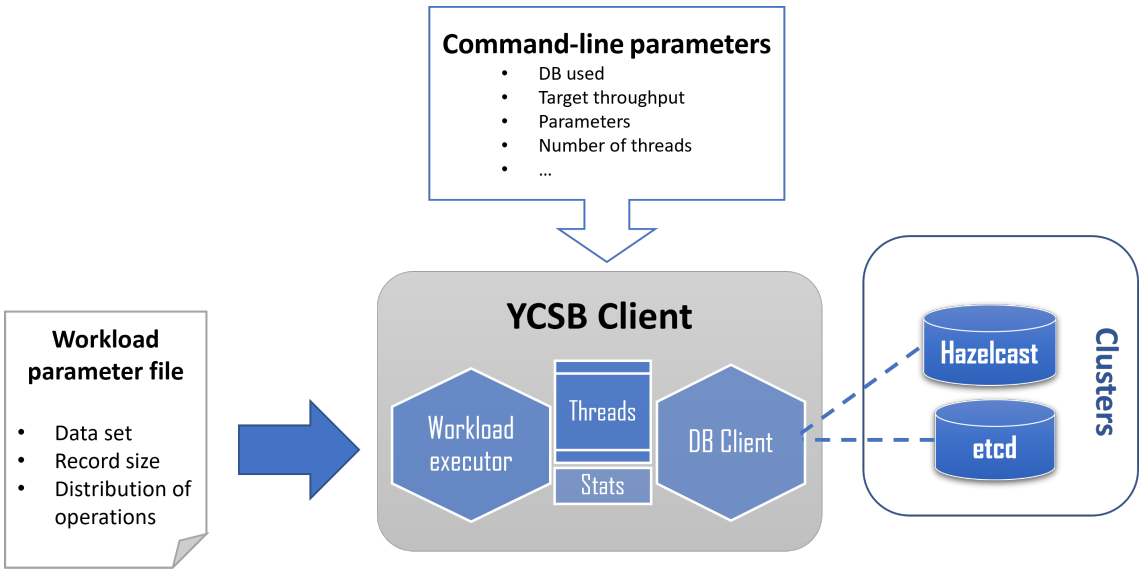


Figure 4.1: YCSB Architecture

The YCSB Client is a program coded in Java to enable the generation of the data to be loaded to the database as well as the operations which make up the workload. A sequential series of operations is executed by each thread by making calls to the database interface layer to, firstly, load the database for the load phase, and then to execute the workload in the transaction phase. The threads control requests generation rate, measure the latency as well as the achieved throughput of their operations, and report these measurements to the statistics module. At the end of the process, the statistics module aggregates the measurements and reports average, percentile latencies, and either a histogram or time series of the latencies.

The inputs received by the client are the properties needed to define its operation. On one hand, the workload parameter file gives the information regarding the workload, independent of the given database. On the other hand, the command-line parameters provide the runtime properties, which are specific to a given experiment. For example, here is where the database interface layer to use (*etcd* or *Hazelcast* in these scenarios) is specified. Other information that can be included in the command-line are the properties used to initialize that layer or the number of client threads, among other information.

For this particular benchmark, the workload and run-time properties given for *etcd* and *Hazelcast* testing remained static to allow both databases to be evaluated under the same conditions. The

details about the properties used and a more detailed description about the specifications considered for this operations can be found in the Subsection of each database, being *Hazelcast*'s specifications in Section 5.2.1, and *etcd*'s details in Section 5.2.2.

4.4.2 Extensibility

Extensibility is one of the primary objectives of YCSB. Any class in the YCSB tool can be replaced, but the most advisable way to achieve this objective is by modifying the elements shown in the hexagonal boxes in the Figure 4.1: The Workload Executor and the DB Client. These elements are specified as properties, and they are loaded dynamically when the client starts.

4.4.2.1 Workload Executor Modifications for Extensibility

The Workload Executor contains code to execute both the load and transaction phases of the workload. The YCSB package includes a standard workload executor called CoreWorkload, which can be completely substituted by another workload executor, or can also be used as it is and achieve extensibility by just defining a set of workloads that use this default executor with different parameters.

By keeping the CoreWorkload as it is and modifying only its parameters, users are allowed to vary several axes of the core package (which operation to perform, the skew in record popularity, the size and number of records, among others).

On the other hand, by defining a new workload executor class and associated parameters, more complex operations can be incorporated, and different tradeoffs can be explored. One instance of the workload object is created and shared among the worker threads, allowing the threads to share common elements like distributions, counters and so on. For each operation, if the client is in the load phase, the thread will execute the workload object's doInsert(). Otherwise, if the client is in the transaction phase method, the workload object's doTransaction() method will be used.

These both ways of achieving extensibility are available for YCSB. However, creating a new workload executor implies greater effort compared to the former approach.

4.4.2.2 Database Client Modifications for Extensibility

The Database Client receives requests from the client threads and translates them into calls against the database. These operations represent the standard "CRUD" operations: Create, Read (read() and scan() functions), Update, Delete. The YCSB Client allows to benchmark new database systems by writing a new class to implement these methods, previously mentioned in the Section 4.3.1

Chapter 5

Implementation

In this chapter, we are going to explain how the two database layer interfaces have been implemented over YCSB, as well as how the configuration was done. This implementation basically consists of the following steps:

- Downloading YCSB [Section 5.1];
- Installation of the database tool (*etcd* / *Hazelcast*)[Section 5.2];
- YCSB development for the DB layer (*etcd* / *Hazelcast*) [Section 5.3];

To replicate this benchmark, we present a list of additional installations required and suggested for conducting this implementation:

- Java;
- Maven;
- Python v2;

5.1 Downloading YCSB

The latest release of YCSB (0.17.0) is downloaded via command line interface (CLI). The commands are presented below.

- For MAC

```
$ curl -O --location https://github.com/brianfrankcooper/YCSB/releases/download
    /0.17.0/ycsb-0.17.0.tar.gz
$ tar xfvz ycsb-0.17.0.tar.gz
$cd ycsb-0.17.0

```

- For Linux

```
$ git clone http://github.com/brianfrankcooper/YCSB.git
$ cd YCSB

```

5.2 Installation of the database tool

In this section we will present the process required for the installation of each of the databases tested for this benchmark. Specifications for *Hazelcast* can be found in Section 5.2.1 and for *etcd* steps are described in Section 5.2.2.

5.2.1 Installation of *Hazelcast*

There are several ways to install *Hazelcast* using CLI, Docker, the binary, or Java [Haz22]. In the project, *Hazelcast* was installed utilizing Java. Additional prerequisite before starting the set up of the Hazelcast via Java is having Maven installed. Subsequently, in the downloaded YCSB folder was created a *Hazelcast* folder in which a Maven project was built with the structure shown in Figure 5.1.

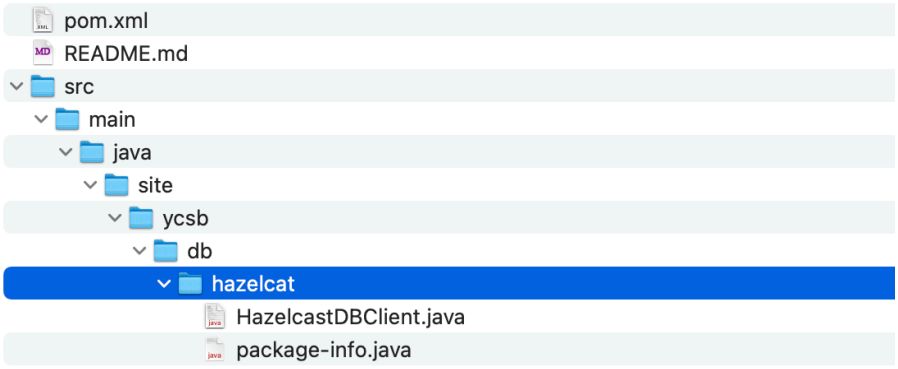


Figure 5.1: Hazelcast project structure

5.2.2 Installation of etcd

The installation of *etcd* in Ubuntu can performed issuing the following commands:

```
$ sudo apt update
$ export RELEASE=$(curl -s https://api.github.com/repos/etcd-io/etcd/releases/latest|grep
    tag_name | cut -d '"' -f 4)
$ wget https://github.com/etcd-io/etcd/releases/download/${RELEASE}/etcd-${RELEASE}-linux
   -amd64.tar.gz
$ tar xvf etcd-${RELEASE}-linux-amd64.tar.gz
$ cd etcd-${RELEASE}-linux-amd64
$ sudo mv etcd etcdctl etcdutl /usr/local/bin
```

Once we have it installed, we create a Maven project with the same structure of the *Hazelcast* one.

5.3 YCSB development for the selected DB layer

There are several parts of YCSB development for any DB layer:

- Modifications in pom.xml file;
- Extending ycsb.DB class and init method implementation;
- Adaptation of the YCSB core for the selected DB layer.

5.3.1 Modifications in pom.xml file

The pom.xml file of both *Hazelcast* and *etcd* project were adapted from the YCSB format. There were appended a parent and dependencies and was modified an artifact of the project.

The following code presents all changes that were done in pom.xml. The sections written as *(selectedDB)* represent the values "hazelcast" or "etcd" depending on the corresponding Database to use. For example, the line *com.(selectedDB)* should be written as *com.hazelcast* or *com.etc*.

```
<parent>
    <groupId>site.ycsb</groupId>
    <groupid>com.(selectedDB)</groupid>
    <artifactId>binding-parent</artifactId>
    <version>0.18.0-SNAPSHOT</version>
    <relativePath>../binding-parent</relativePath>
</parent>

<artifactId>(selectedDB)-binding</artifactId>
    <name>(selectedDB) Binding</name>
    <packaging>jar</packaging>

<dependencies>
    <dependency>
        <groupId>site.ycsb</groupId>
        <artifactId>core</artifactId>
        <version>0.17.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

The YCSB benchmark has been implemented using *etcd* 0.0.2 and *Hazelcast* 5.2.1.. The presented below dependency was included in the pom.xml of the project in order to install Hazelcast and etcd Java clients specifically via Maven. As can be observed, for *etcd* we have used the jetcd Java client, which is the one developed by the *etcd* developers, nonetheless, we are aware that there are other implementations of a Java client for etcd addressing some of the defficiencies of jetcd[1], but we decided to stick with jetcd because we thought it made more sense to benchmark the tools as provided by the developers.

- For *Hazelcast*:

```
<dependencies>
    <dependency>
        <groupid>com.hazelcast</groupid>
        <artifactid>hazelcast</artifactid>
        <version>5.2.1</version>
    </dependency>
</dependencies>

```

- For *etcd:*

```
<dependencies>
    <dependency>
        <groupId>com.coreos</groupId>
        <artifactId>jetcd-core</artifactId>
        <version>0.0.2</version>
```

[1]For instance, the etcd-java Java client for etcd developed by IBM is a common alternative.

```
    </dependency>
</dependencies>
```

Below we can see more additional items that can be found in our *Hazelcast* implementation in <dependencies>, however, these were not required for *etcd*.

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.1</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.13</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.3</version>
    <type>jar</type>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.1.3</version>
    <type>jar</type>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Finally, once changes have been done to the pom.xml file and the clients have been implemented, we compile it with maven by using the following instruction in command line:

mvn -pl site.ycsb:(selectedDB)-binding -am clean package

5.3.2 Extending ycsb.DB class

The YCSB development for the DB layer includes part of extending the ycsb.DB class to work with the desired database. Extending of ycsb.DB class consists of implementation of 5 database methods (insert, update, read, scan and delete) mentioned in the Section 4.3.1, as well as optionally implement the init() and cleanup() methods.

We will present how we have developed an implementation of the YCSB client to be able to work with both our tools.

- For *Hazelcast* (HazelcastDBClient)

  In the project initialization of *Hazelcast*'s cluster was performed via init() method, which can

be consulted in 5.1. As can be seen, we just access a Cluster named *"YCSB-hz"*.

```
public class HazelcastDBClient extends DB{
    private static HazelcastInstance hz;
    @Override
    public void init() throws DBException{
        Config confYCSB = new Config();
        confYCSB.setClusterName("YCSB-hz");
        hz = Hazelcast.newHazelcastInstance(confYCSB);
    }
    ...
```

Listing 5.1: init() Method

Next, we implement the 5 methods required by YCSB, i.e., read(), insert(), update(), delete() and scan(). All of them can be seen in the following code Listings.

```
@Override
public Status read(String table, String key, Set<String> fields, Map<String,
    ByteIterator> result) {
    if(table==""){
      table="usertable";
    }
    try{
      IMap<String, Map<String, String>> myMap = hz.getMap(table);
      Map<String, String> resultMap = myMap.get(key);
      if(fields != null){
        for(String f : fields) {
          result.put(f, new StringByteIterator(resultMap.get(f)));
        }
      } else if(resultMap != null){
        for(Map.Entry<String, String> entry : resultMap.entrySet()) {
          result.put(entry.getKey(), new StringByteIterator(entry.getValue()));
        }
      }
      return Status.OK;
    } catch (Exception e) {
      e.printStackTrace();
      return Status.ERROR;
    }
}
```

Listing 5.2: read() Method

```
@Override
public Status insert(String table, String key, Map<String, ByteIterator> values){
    Map<String, String> strValues = StringByteIterator.getStringMap(values);
    IMap<String, Map<String, String>> myMap = hz.getMap(table);
    try{
      myMap.put(key, strValues);
      return Status.OK;
    }catch(Exception e){
      e.printStackTrace();
      return Status.ERROR;
    }
}
```

Listing 5.3: insert() Method

```
@Override
```

```
public Status delete(String table, String key) {
    ConcurrentMap<String, Map<String, String>> myMap = hz.getMap(table);
    try{
      myMap.remove(key);
      return Status.OK;
    }catch(Exception e){
      e.printStackTrace();
      return Status.ERROR;
    }
}
```

Listing 5.4: delete() Method

```
@Override
public Status update(String table, String key, Map<String, ByteIterator> values){
    return insert(table, key, values);
}
```

Listing 5.5: update() Method

```
@Override
public Status scan(String table, String startkey, int recordcount,
  Set<String> fields, Vector<HashMap<String, ByteIterator>> result) {
    IMap<String, Map<String, String>> myMap = hz.getMap(table);
    int count = 0;
    try{
      for (String key : myMap.keySet()) {
        if(count >= recordcount){
          break;
        }
        if(key.compareTo(startkey) >= 0){
          count++;
          HashMap<String, ByteIterator> values = new HashMap<String, ByteIterator>()
    ;
          Map<String, String> res = myMap.get(key);
          StringByteIterator.putAllAsByteIterators(values, res);
          result.add(values);
        }
      }
      return Status.OK;
    } catch(Exception e){
      e.printStackTrace();
      return Status.ERROR;
    }
}
```

Listing 5.6: scan() Method

In this last Listing 5.6, we can see how the scan operation has been implemented in a naïve way, traversing the whole database until we find all the desired values. Of course, this is not the idea behind the scan operation, but the characteristics of Hazelcast as a key value store make it impossible to perform this operation in a different manner, because the keys are hashed, so prefix-search is not possible.

- For etcd (etcdDBClient) The same has been done with *etcd*, with the different that *etcd* allows an efficient scan operation. In this case we present the whole class in Listing 5.7.

```
public class EtcdClient extends DB{
  private static Client client;
```

```
3   private static KV kvClient;
4
5   public void init() throws DBException{
6     client = Client.builder().endpoints("http://localhost:2379").build();
7     kvClient=client.getKVClient();
8   }
9
10  @Override
11  public Status read(String table, String key, Set<String> fields, Map<String,
      ByteIterator> result){
12    if(table==""){
13      table="usertable";
14    }
15    CompletableFuture<GetResponse> futureResponse = new CompletableFuture<
      GetResponse>();
16    try {
17      if(fields != null){
18        for(String f : fields) {
19          futureResponse = kvClient.get(ByteSequence.fromString(table + "." + key +
      "." + f));
20        }
21      }else{
22        ByteSequence fullkey = ByteSequence.fromString(table+"."+key);
23        GetOption option = GetOption.newBuilder().withPrefix(fullkey).build();
24        futureResponse = kvClient.get(fullkey, option);
25      }
26      GetResponse response = futureResponse.get();
27      for(KeyValue kv:response.getKvs()){
28        result.put(kv.getKey().toString(), new StringByteIterator(kv.getValue().
      toString()));
29      }
30      return result.isEmpty() ? Status.ERROR : Status.OK;
31    } catch (InterruptedException e) {
32      e.printStackTrace();
33      return Status.ERROR;
34    } catch (ExecutionException e) {
35      e.printStackTrace();
36      return Status.ERROR;
37    }
38  }
39
40  @Override
41  public Status insert(String table, String key, Map<String, ByteIterator> values) {
42    Map<String, String> strValues = StringByteIterator.getStringMap(values);
43    for(String k : strValues.keySet()) {
44      try {
45        kvClient.put(ByteSequence.fromString(table + "." + key + "." + k),
46            ByteSequence.fromString(strValues.get(k))).get();
47      } catch (InterruptedException e) {
48        e.printStackTrace();
49        return Status.ERROR;
50      } catch (ExecutionException e) {
51        e.printStackTrace();
52        return Status.ERROR;
53      }
54    }
55    return Status.OK;
56  }
57
58  @Override
59  public Status delete(String table, String key) {
```

```
60      if(table==""){
61        table="usertable";
62      }
63      ByteSequence fullkey = ByteSequence.fromString(table+"."+key);
64      DeleteOption option = DeleteOption.newBuilder().withPrefix(fullkey).build();
65
66      try {
67        return kvClient.delete(fullkey, option).get().getDeleted() > 0 ? Status.OK :
        Status.ERROR;
68      } catch (InterruptedException e) {
69        e.printStackTrace();
70        return Status.ERROR;
71      } catch (ExecutionException e) {
72        e.printStackTrace();
73        return Status.ERROR;
74      }
75    }
76
77    @Override
78    public Status update(String table, String key, Map<String, ByteIterator> values) {
79      ByteSequence fullkey = ByteSequence.fromString(table+"."+key);
80      GetOption option = GetOption.newBuilder().withPrefix(fullkey).build();
81      try {
82        return kvClient.get(fullkey, option).get().getCount() > 0 ? insert(table, key,
         values) : Status.ERROR;
83      } catch (InterruptedException e) {
84        e.printStackTrace();
85        return Status.ERROR;
86      } catch (ExecutionException e) {
87        e.printStackTrace();
88        return Status.ERROR;
89      }
90    }
91
92    @Override
93    public Status scan(String table, String startkey,
94        int recordcount, Set<String> fields,
95        Vector<HashMap<String, ByteIterator>> result) {
96      ByteSequence fullkey = ByteSequence.fromString(table+"."+startkey);
97      GetOption option = GetOption.newBuilder().withLimit(recordcount).withPrefix(
        fullkey).build();
98      try {
99        GetResponse response = kvClient.get(fullkey, option).get();
100       for(KeyValue kv:response.getKvs()){
101         HashMap<String, ByteIterator> map = new HashMap<String, ByteIterator>();
102         map.put(kv.getKey().toString(), new StringByteIterator(kv.getValue().
        toString()));
103         result.add(map);
104       }
105       return Status.OK;
106     } catch (InterruptedException e) {
107       e.printStackTrace();
108       return Status.ERROR;
109     } catch (ExecutionException e) {
110       e.printStackTrace();
111       return Status.ERROR;
112     }
113   }
114
115   @Override
116   public void cleanup(){
```

```
117         client.close();
118       }
119     }
```

Listing 5.7: EtcdClient Class

5.3.3 Adaptation of the YCSB core for the DB layer

To be able to perform the load and run phases of YCSB benchmark for both databases there were done additional changes in the core YCSB files. *Hazelcast* and *etcd* databases were added as a databases parameter in the main YCSB Unix Executable File using standart format of YCSB benchmark:

```
1
2     "hazelcast": "site.ycsb.db.hazelcast.HazelcastDBClient"
3     "etcd": "site.ycsb.db.etcd.etcdDBClient"
```

Listing 5.8: ycsb python script properties.

Into the bindings.properties file the same parameter was added without using quotation marks. Subsequently, after the adjustments, both DB layers were ready for performing load and run phases of YCSB.

Chapter 6

A custom use case

In this chapter, we explain the made-up scenario for which there is no suitable predefined workload in YCSB, so we aim at defining it for benchmarking this use case.

For this scenario, three of the use cases tested are typical use cases from the standard YCSB workloads: Workload A, workload F and workload E. A description regarding these workloads can be found in Section 4.3.1.1.

6.1 Process Orchestrator

In most modern operating systems, processes are the logical processing unit, and their management is crucial for the performance of every machine. This relevance is even more important in multicore machines and reaches its peak for decentralized processing servers. It is an important task to distribute processes correctly between the different servers in such a way that we keep them balanced - i.e. all of them carry a similar workload at every moment - as well as least idle as possible. These two characteristic are intertwined and must be taken into consideration. Here, we are going to describe a simplistic process orchestrator based on a key-value store that keeps track of the process and the servers where they can be executed.

6.1.1 Set up

We suppose a cluster of S servers, each of them with $C_i,\ i \in 1, ..., S$ cores, a central machine producing processes to be executed and the orchestrator, to whom the processes are sent and which must decide in which server should they execute.

We are going to suppose that the processes are independent between each other, meaning we are not going to model the fact that some processes might need to wait for other to finish before starting. In addition, each process will have the ability to ask for only one core, so the orchestrator needs to find a server that is able to allocate it.

Another simplifying assumption is that, in case there is no server ready to allocate the current process, we wait until its execution is possible. In a real scenario, this would not be the case, as we are trying to optimize the usage of CPUs and the time it takes to complete all processes, it is possible to skip some processes to start with others that might have been generated later.

Finally, we assume that the processes return a simple result, that must be preserved in the database

for consistency purposes and returned to the central machine.

6.1.2 The orchestrator

With this set up, a simple orchestrator could follow the algorithm in Listing 6.1. The operations in blue are those which need to be taken into account for creating the benchmark. As we have seen, a scan operation cannot be performed using Hazelcast, so the search needs to be performed in another fashion. A simple way is to maintain in the main memory a list of the servers and CPUs, which should be possible. Thus, we can search for a free pid and a free server with just one read each. This is not a bad approach, as keeping this list in memory should be fast and not very heavy in terms of space. For example, if we had 1000 servers, each of them with 32 cores, we would need to maintain an array of boolean of length 32000. That is a huge server set up, and the cost of the list in memory is little.

Thus, the workload should, for each process, perform:

- 5 read operations.
- 5 update operations.
- 2 insert opreations.

This means that our custom workload would be defined with 42% reads, 42% updates and 16% inserts.

```
orchestrator(KVstore db, Servers S, ProcessGen pGen, int coresPerServer):
    init(db, S, coresPerServer)
    while(pGen.generateProcess proc):
        search db to find a server with freecores>0 -> s
        core = db.read(s, freecores) # obtain core in which execute the process
        db.update(s, freecores = core-1) # reduce free cores
        search db to find first free processID -> pid
        db.insert(proc.id, localpid=pid) # associate real id to local pid
        db.update(pid, server=s, core=core, realpid=proc.id) # associate server s and core
     to process pid

        if(process_ended(Server s, Process pid, int core)):
            result = db.read(s, core)
            real_pid = db.read(pid, realpid)
            db.insert(real_pid, result=result) # Keep track of past processes outcomes
            db.update(pid, 'free') # Free localpid in db
            db.update(s, freecores = freecores + 1) # Free one core in S
            db.update(s, core, null) # Set result to null
            return (pid, result) to pGen

init(KVstore db, Servers S, int coresPerServer):
    for(s in S):
        db.put(s, freecores = coresPerServer)
        for(core in range(1, coresPerServer):
            db.put(s, core, null) # In this record we can introduce the result

    for(processID in range(1,coresPerServer*S):
        db.put(processID, 'free')
```

Listing 6.1: Simple process orchestrator.

6.1.3 The sizes

For workload sizes, we simulate different system scales:

1. 10 servers, 32 cores each: 320 initial records.
2. 100 servers, 32 cores each: 3200 initial records.
3. 500 servers, 32 cores each: 16000 initial records.
4. 1000 servers, 32 cores each: 32000 initial records.

For each of them, we run a simulation of 1000, 10000, 50000, 100000 processes, in this order.

Chapter 7

Results and analysis

In this chapter we are going to demonstrate the results and provide an analysis of them. Overall, the results will be provided for the Load and Run phases of the Workloads A, E and the Customized one we made of YSCB benchmark performed on *etcd* and *Hazelcast*. First, we will state how the tests were performed and the characteristics of the machine over which they were performed. Then, we will show the obtained results and try to understand why they were obtained.

7.1 Set up

7.1.1 Test structure

The test have been performed, for each database, with the following steps:

1. Initialize the database.
2. If the database is not empty, empty it.
3. Run the load phase for the desired number of records. This leads to the measures:
    - Load time: elapsed time (in seconds) to complete the load phase.
    - Throughput: number of operations per second. In this case it should be equivalent to

$$Throughput = \frac{N_{records}}{Load\ time}.$$

    - Insert latency: average time (in microseconds) to finish an insert operation.
4. Run the 4 run phases for the four amounts of operations. This step provides us with the following measures:
    - Run time: elapsed time (in seconds) to complete the run phase.
    - Overall throughput: number of operations per second.
    - For each type of operation in the workload, we also obtain the average throughput of this type of operation and the average latency.

7.1.2 Machine characteristics

All tests have been performed in the same machine. It has the following characteristics:

- Manufacturer: ASUSTeK COMPUTER INC.
- Product Name: ROG Strix G513QM_G513QM
- Version: 1.0
- Memory: 16 GiB
- Processor: AMD® Ryzen 9 5900hx with radeon graphics × 16
- OS: Ubuntu 22.04.1 LTS
- OS type: 64-bit

7.2 Load phase results

This section is dedicated to the load phase results. There are three measures described: time, throughput and insert latency for three workloads (A, Custom and E).

Figure 7.1 demonstrates how much time was spent on the loading for different number of rows for each of the databases according to the chosen workload. Times are expressed in logarithmic scale.

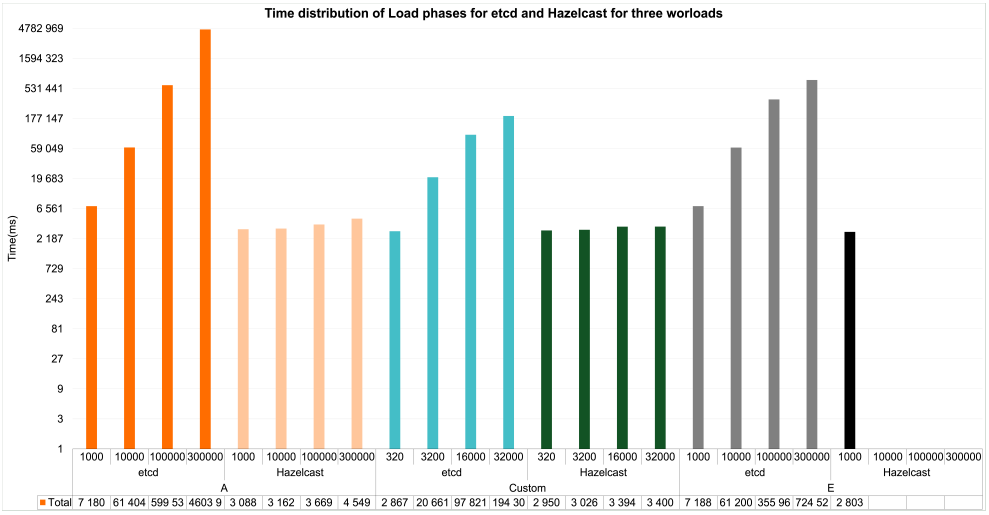


Figure 7.1: Load phase time distribution chart

Overall, *etcd* load phase time shows significant growth with an increase in the number of rows loaded for all three workloads. Furthermore, the indicator climbed in proportion to the extend of loading a large number of rows. As can be observed, the log scale graph increases linearly, which means that the insertion time is increasing exponentially, as well as the number of records inserted. The **reason** for these observations is simple: *etcd* maintains a fairly stable insertion rate, which translates into loading time being proportional to the number of records. This could be the explanation behind the *etcd* developers to advice to use it for infrequently updated data: data modification is slow.

On the other hand, Hazelcast loading time slightly rose with an increase in the number of rows loaded

for Custom and A workloads. It can also be seen how these times are lower than those of *etcd*, even if we compare the longest of *Hazelcast* with the shortest of *etcd*. This very slight increase in time, against an increasing number of insertions is not as easily explained. We will delve into this question after taking a look at the throughput measures for the load phase.

The following bar graph in Figure 7.2 compares the throughput indicator for the Workloads A, Custom and E load phases of YSCB benchmark performed on *etcd* and *Hazelcast*.

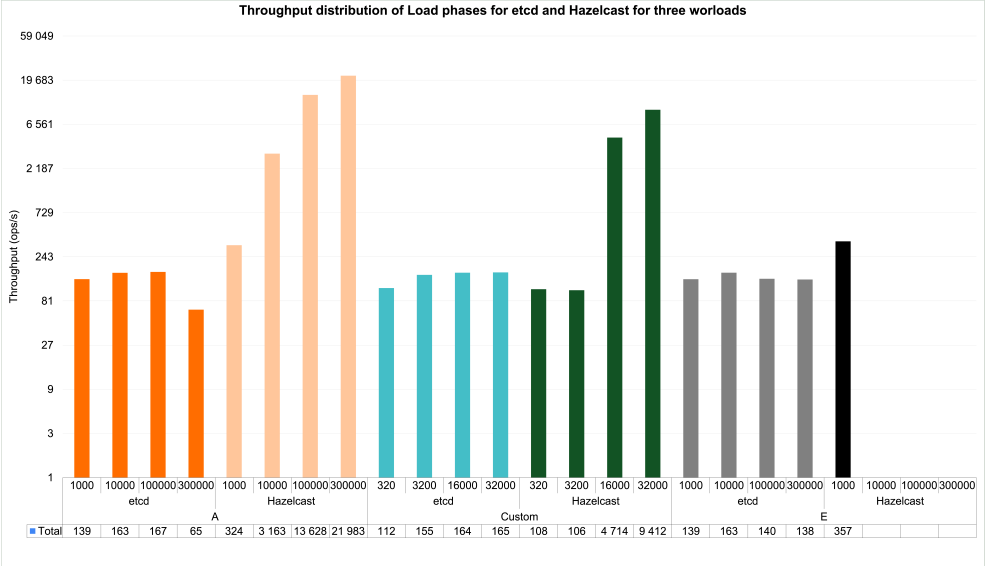


Figure 7.2: Load phase throughput distribution

In general, *etcd* load phase throughput for the workload A slightly increases up to 167 operations per second for 100.000 rows database and significantly declines down to 65 operations per second for 300.000 rows database. *Hazelcast* throughput for the worload A demonstrates constant growth with the database size changes.

For the Custom workload *etcd* throughput steadily went up through 4 scales of the database. While *Hazelcast* throughput fluctuated for two initial scales and subsequently, skyrocketed for 16.000 rows and 32.000 database scales.

etcd throughput in the workload E improved for the 10.000 database scale and decreased for the next two scales. For the first *Hazelcast* scale in workload E throughput significantly exceeded *etcd* throughput parameters.

As we can observe, *etcd* maintains a fairly constant throughput indicator, which aligns with our past explanation for the exponential increasing loading times. On the other hand, we see how *Hazelcast* leverages the increasing number of records and manages to increase the throughput when the number of records increases. This observation explains how loading times for *Hazelcast* are almost constant, even with an increasing number of insertion.

The last diagram presented for the load phase is depicted in 7.3 and shows the insert latency indicator for the Workloads A, Custom and E load phases of YSCB benchmark performed on *etcd* and *Hazelcast*.

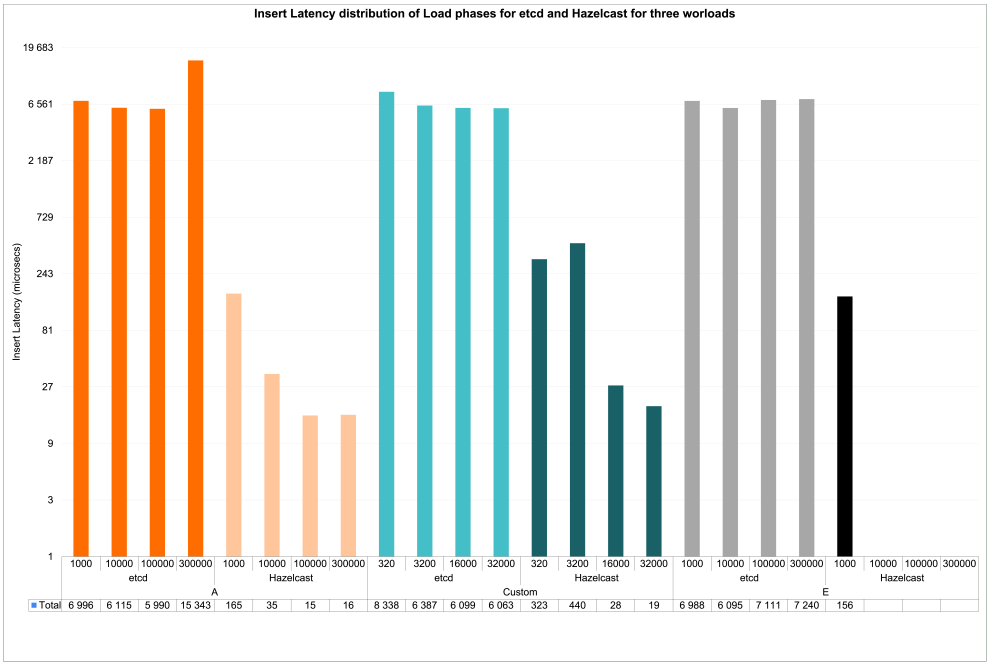


Figure 7.3: Load phase insert latency distribution

As it can be seen on the chart, the insert latency in all scales for *Hazelcast* database for workload A, Custom and E considerably lower than for *etcd*. The insert latency of *Hazelcast* workload A in the first scale (1.000 rows) is 40 times lower. For the Custom workload the same parameter for *etcd* is 18 times higher than *Hazelcast* insert latency. Thus, inserting data into *Hazelcast* is significantly faster than into *etcd* and this cause lower insert latency on the *Hazelcast* side. As expected, this is providing us with the same information as the throughput plot, but with inverted values.

To summarise, it this section were analysed three charts related to the load phase parameters, specifically time, throughput and insert latency measured on *etcd* and *Hazelcast* for three workloads (A, Custom and E). It is clear that the performance of the load phase and it's parameters for *Hazelcast* greatly exceeds the capabilities of *etcd*, which is expected, as *etcd* is thought for static databases, while *Hazelcast* is the contrary. The hashing of keys performed by *Hazelcast* is another characteristic that makes it faster for insertion operations.

7.3 Workload A results

The section is devoted to the run phase results of the Workload A. Specifically, time, throughput, read and update latency were measured. Each of the parameters was analysed on 4 scales (1.000/10.000/100.000/300.000 records) with different numbers of operations.

The following bar graph compares the run time indicator (in logarithmic scale) for the Workloads A of the YSCB benchmark run phase on our two target databases.

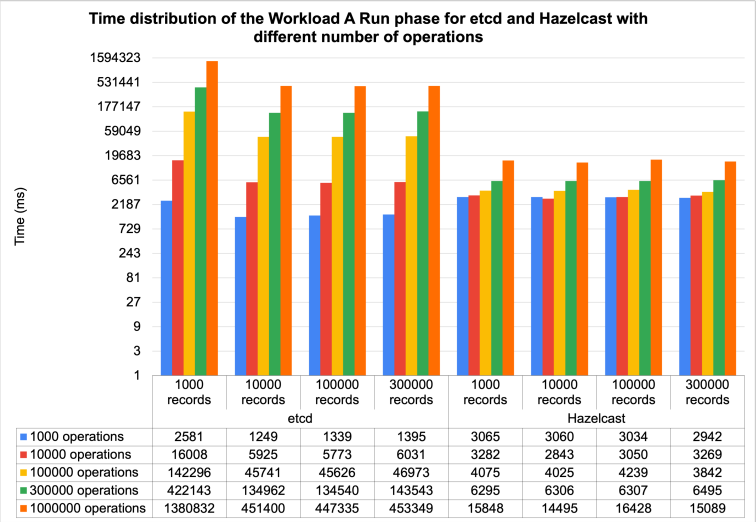


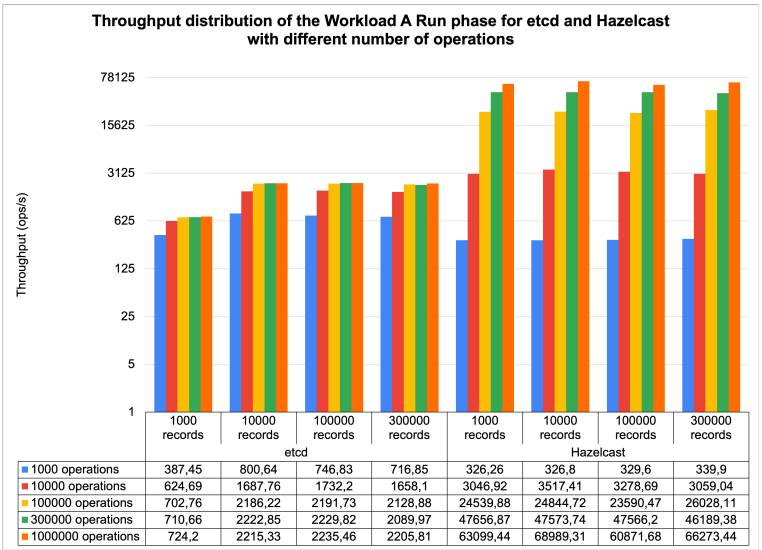
Figure 7.4: Workload A Run phase time distribution

To begin with, *etcd* run phase overall time for the Workload A rapidly growths with the increase of number of operations. The highest *etcd* time parameter reaches 1.380.832 ms. (around 23 min.) for the first DB scale of 1.000 records with 1.000.000 operations. Overall, *etcd* run phase time demonstrates a downward trend for 10.000 and 100.000 records DB scales and subsequently, slightly rises again for 300.000 records scale.

Alternatively, *Hazelcast* run phase time for the Workload A slightly rises within increasing scales and operations except running with 300.000 and 1.000.000 operations. For those options time changes rapidly. Additional point to mention is a slight decrease within run time in 10.000 records *Hazelcast* DB scale between 1.000 and 10.000 operations.

Overall, execution time of the Workload A run phase for *etcd* exceeds *Hazelcast* run time for all scales and operations, except the first combination specifically 1.000 operations with all DB scales. We observe how the measured times are more directly proportional to the number of operations in the case of *etcd* than in the case of *Hazelcast*, meaning this, again, that *etcd* seems to take a fairly constant time for each operation performed, while *Hazelcast* is able to increase its throughput when the number of operations to be done increases. We will assess this sentence in the following paragraphs.

The diagram in Figure 7.5 describes throughput distribution of the Workload A run phase for different *etcd* and *Hazelcast* scales with 4 quantities of operations.



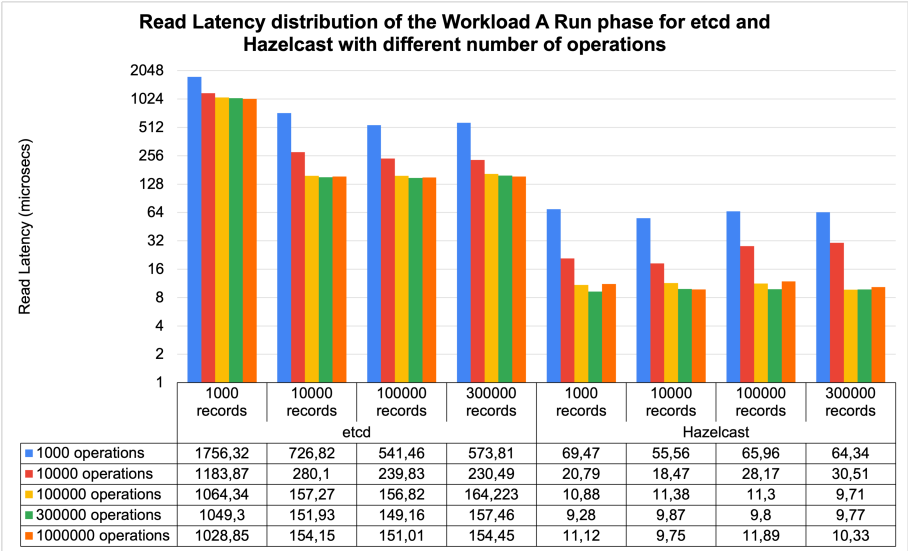
	etcd				Hazelcast			
	1000 records	10000 records	100000 records	300000 records	1000 records	10000 records	100000 records	300000 records
1000 operations	387,45	800,64	746,83	716,85	326,26	326,8	329,6	339,9
10000 operations	624,69	1687,76	1732,2	1658,1	3046,92	3517,41	3278,69	3059,04
100000 operations	702,76	2186,22	2191,73	2128,88	24539,88	24844,72	23590,47	26028,11
300000 operations	710,66	2222,85	2229,82	2089,97	47656,87	47573,74	47566,2	46189,38
1000000 operations	724,2	2215,33	2235,46	2205,81	63099,44	68989,31	60871,68	66273,44

Figure 7.5: Workload A Run phase throughput distribution

On the whole, throughput parameter with 1.000 operations considerable higher for *etcd* within three DB scales namely 10.000/100.000/300.000 rows. The difference for 1.000 rows scale between *etcd* and *Hazelcast* equals to 61.19 operations per second.

Moreover, *etcd* throughput parameter demonstrates gentle growth for all possible combinations of database size and number of operations. While, *Hazelcast* throughput parameter rises sharply and fluctuates above 60,000 operations per second. This confirms our previous statement about this measures, and explains why *Hazelcast* is able to execute an exponentially increasing amount of operations with a less-than-exponential increase in time.

The chart in Figure 7.6 compares read latency distribution of the Workload A run phase for 4 scales on *etcd* and *Hazelcast* for different number of operations.



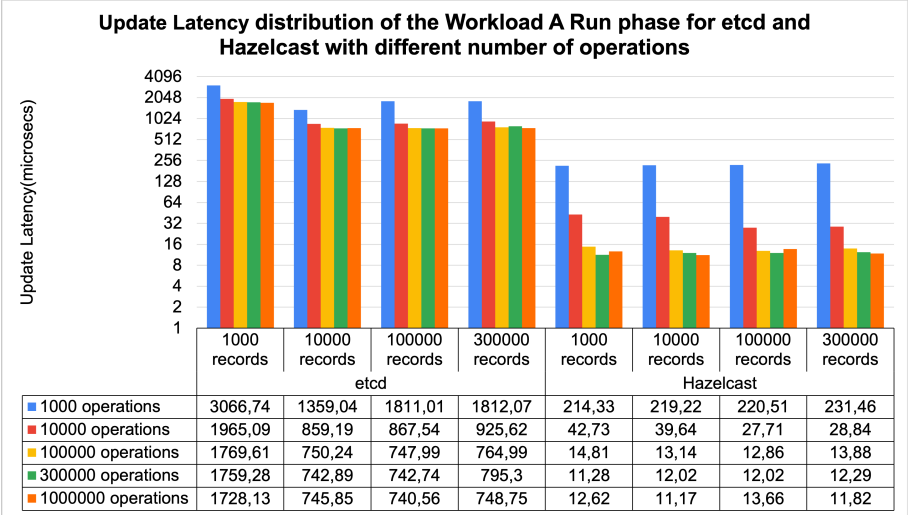
	etcd				Hazelcast			
	1000 records	10000 records	100000 records	300000 records	1000 records	10000 records	100000 records	300000 records
1000 operations	1756,32	726,82	541,46	573,81	69,47	55,56	65,96	64,34
10000 operations	1183,87	280,1	239,83	230,49	20,79	18,47	28,17	30,51
100000 operations	1064,34	157,27	156,82	164,223	10,88	11,38	11,3	9,71
300000 operations	1049,3	151,93	149,16	157,46	9,28	9,87	9,8	9,77
1000000 operations	1028,85	154,15	151,01	154,45	11,12	9,75	11,89	10,33

Figure 7.6: Workload A Run phase read latency distribution

With respect to the chart, the highest read latency is reached for both tools with 1000 operations within all scales. This seemingly counter-intuitive fact can be explained because when database are less populated, it is less able to leverage the optimizations developed to improve performance with increasing number of records. The maximum rate for *etcd* is equal to 1756.32 microseconds and 69.47 microseconds for *Hazelcast*. Furthermore, read latency for *etcd* remains stable for all scales with 100.000, 300.000 and 1.000.000 operations. While, for *Hazelcast* the read latency parameter fluctuates within all parameters. The reason behind this observation can be that when the map gets more populated, it could stop fitting in memory, so swap memory or caches start being used, decreasing a little bit the read speed.

In brief, it can be noted that there is a steady downward trend in the read latency rate with the increase of operations and the size of database for the analyzed tools with the specified parameters. What is more, *Hazelcast* latency is significantly lower than that of *etcd* which is expected because of the difference between both models.

The next diagram represents the information about update latency of the Workload A run phase for *etcd* and *Hazelcast* with progressive scale of operation quantity.



	etcd				Hazelcast			
	1000 records	10000 records	100000 records	300000 records	1000 records	10000 records	100000 records	300000 records
1000 operations	3066,74	1359,04	1811,01	1812,07	214,33	219,22	220,51	231,46
10000 operations	1965,09	859,19	867,54	925,62	42,73	39,64	27,71	28,84
100000 operations	1769,61	750,24	747,99	764,99	14,81	13,14	12,86	13,88
300000 operations	1759,28	742,89	742,74	795,3	11,28	12,02	12,02	12,29
1000000 operations	1728,13	745,85	740,56	748,75	12,62	11,17	13,66	11,82

Figure 7.7: Workload A Run phase update latency distribution

The update latency level remains stable above 700 microseconds and shows a decrease with the increase of operations and the size of database for *etcd*. The lowest rate that is equal to 740.56 microseconds occurs with 100.000 rows scale and with 1.000.000 operations. While the highest rate that is equal to 3066.74 microseconds reaches with 1000 rows scale and with 1000 operations. For *Hazelcast*, update latency demonstrates variability depending on the size of the database and the number of operations performed. The lowest rate that is equal to 11.17 microseconds occurs with 10.000 rows scale and with 1.000.000 operations. While the highest rate that is equal to 231.46 microseconds reaches with 300000 rows scale and with 1.000 operations. Additionally, the indicator for *etcd* is considerably higher than for *Hazelcast*. It means that data update transactions are faster on the *Hazelcast* side with less delay.

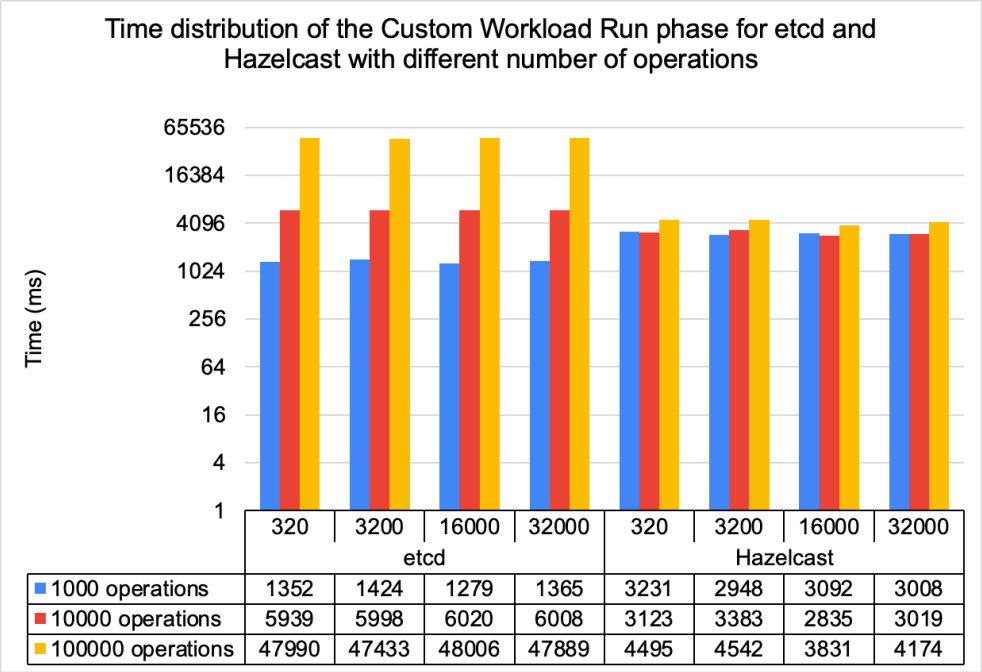
In conclusion, current section was devoted to the analysis of Workload A results. There were presented four charts related to the run phase parameters, specifically time, throughput, read and update latencies measured on *etcd* and *Hazelcast* with different numbers of operations. Overall, the performance of the Workload A run phase and it's parameters for *Hazelcast* significantly exceeds results of *etcd*. So, with these results at hand, for applications requiring common changes in the data, *Hazelcast* should

be chosen in most cases above *etcd*. Specially when dealing with a greater number of records and operations.

7.4 Custom Workload results

This section describes the custom workload in terms of the same parameters as in the workload A part, and also adds insert latency results. This part shows the results for 1000, 10.000 and 100.000 operations and for 4 factors: 320, 3200, 16.000 and 32.000 rows.

The diagram below shows the time distribution on the run phase.



	etcd				Hazelcast			
	320	3200	16000	32000	320	3200	16000	32000
1000 operations	1352	1424	1279	1365	3231	2948	3092	3008
10000 operations	5939	5998	6020	6008	3123	3383	2835	3019
100000 operations	47990	47433	48006	47889	4495	4542	3831	4174

Figure 7.8: Custom workload Run phase time distribution

For both *etcd* and *Hazelcast*, the times don't change much as the number of records changes. However, the difference in time depending on the number of transactions is much more noticeable for *etcd*, with an exponential growth in time, which is compatible with all observations before: *etcd* throughput is fairly stable. For *Hazelcast*, again, we see how it leverages an increased number of operations to maintain a very constant overall execution time.

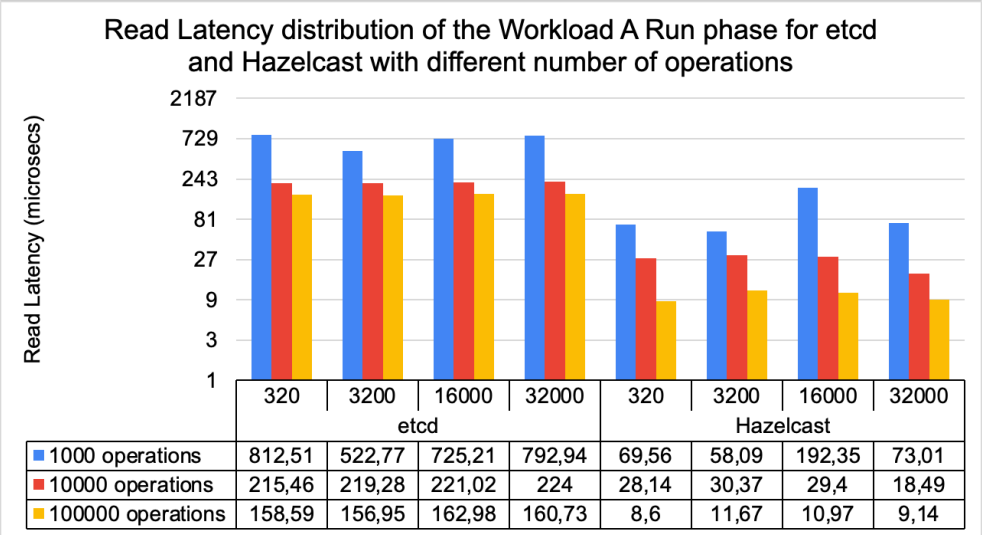
The opposite result can be noticed in the diagram of the throughput distribution of Figure 7.9.

	etcd				Hazelcast			
	320	3200	16000	32000	320	3200	16000	32000
1000 operations	739,64	702,25	781,86	732,6	309,5	339,21	323,42	332,45
10000 operations	1683,79	1667,22	1661,13	1664,45	3202,05	2955,96	3527,34	3312,36
100000 operations	2083,77	2108,24	2083,07	2088,16	22246,94	22016,73	26102,85	23957,83

Figure 7.9: Custom workload Run phase throughput distribution

Although there is a fairly noticeable throughput difference for 1000 and 10.000 operations for *etcd*, in general, the throughput distribution increases slightly compared to *Hazelcast*. In each case, for *etcd*, the throughput difference between 1000 and 10.000 operations increases by a little more than 2 times and slightly grows between 10.000 and 100.000 operations (approx. 400 ops/s). At the same time, for *Hazelcast*, the number of ops/s increases on average 10 times from 1000 to 10.000 operations and 7 times from 10.000 to 100.000 operations.

In Figure 7.10, the measured latencies are shown for both databases. As usual, *Hazelcast* outperforms *etcd* because of the different data models.



	etcd				Hazelcast			
	320	3200	16000	32000	320	3200	16000	32000
1000 operations	812,51	522,77	725,21	792,94	69,56	58,09	192,35	73,01
10000 operations	215,46	219,28	221,02	224	28,14	30,37	29,4	18,49
100000 operations	158,59	156,95	162,98	160,73	8,6	11,67	10,97	9,14

Figure 7.10: Custom workload Run phase read-latency distribution

The next diagram represents Update Latency distribution. Overall, values for both tools vary slightly

for different scales, with *Hazelcast* being able to improve the average performance when dealing with an increased number of operations to perform.

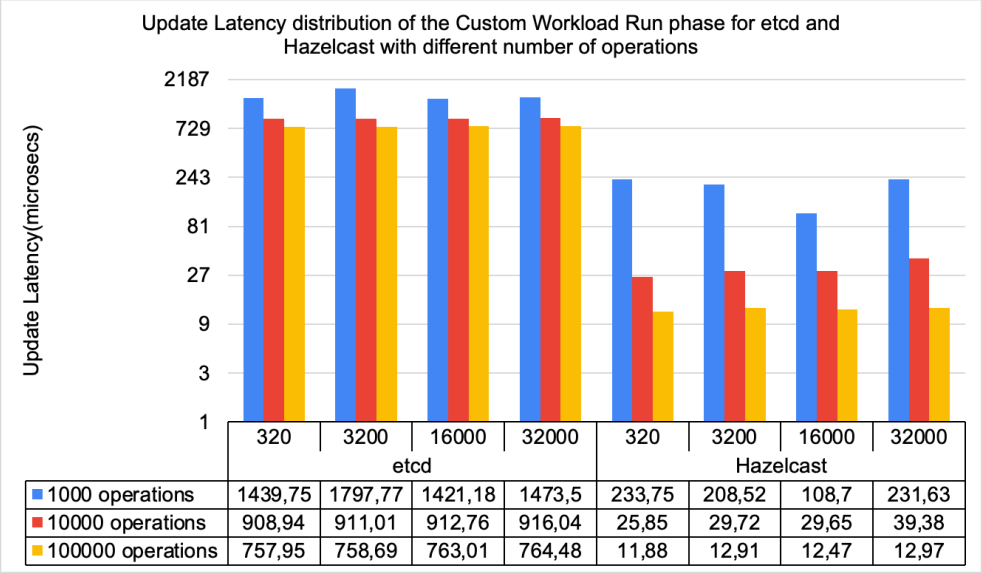


Figure 7.11: Custom workload Run phase update-latency distribution

Finally, we show the insertion latency measures for both databases in 7.12. The results are similar to what we have already seen.

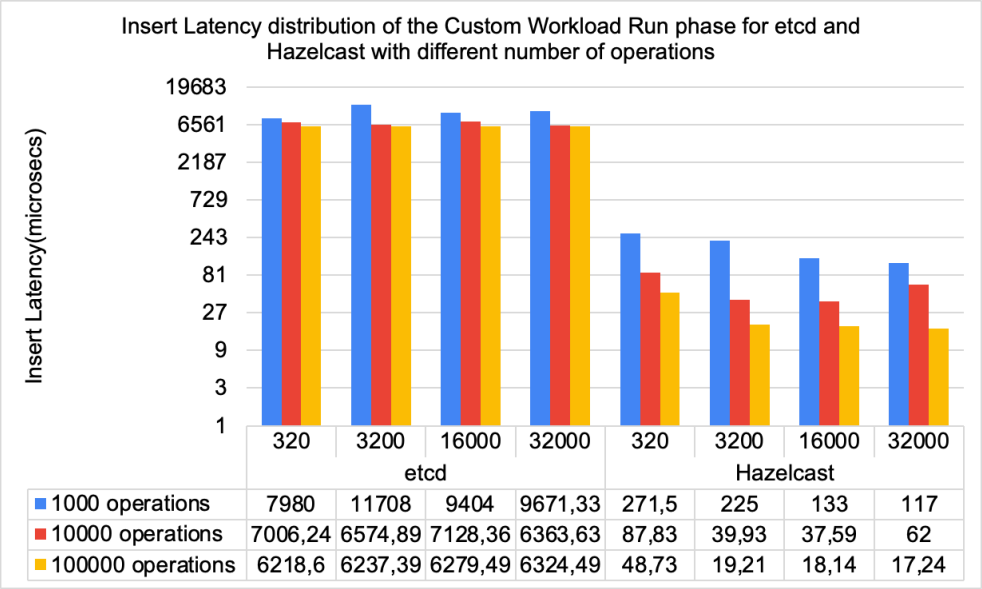


Figure 7.12: Custom workload Run phase insert-latency distribution

In summary, we see how *Hazelcast* outperforms *etcd* in this particular use case. If we reflect about how both of them are developed and the objectives of these technologies, this is the logical result to obtain: *etcd* is thought to cope with persistent non-changing data, while *Hazelcast IMDG* is the contrary, an in-memory key-value store thought for fast access and modification capabilities. Nonetheless, it could be possible to improve the theoretical performance of the orchestrator by making use of the scan

operation to search for free servers and to be able to cope with processes with multiple arguments and/or returned values. The problem is that *Hazelcast* would be highly inneficient in this case, so a different solution would need to be found.

7.5 Workload E results

This section is devoted to the run phase results of the Workload E, Time, throughput, scan and insert latency parameters were measured for 4 scales (1.000/10.000/50.000/100.000 rows) with different numbers of operations. Results for *Hazelcast* are presented only for the first scale (1.000 rows) with 4 quantities of operations (1.000/10.000/100.000/300.000 operations), because for bigger scales the inefficient scan operation[1] made it last too long, as we will see now.

The following bar graph compares the run time indicator for the Workloads E of the YSCB benchmark run phase on both our databases, for different scales with increasing operation quantities. We can observe how, as should be expected in this point, *etcd* run times increases exponentially (linearly in the log-scale). On the other side, we start noticing a strange event: run time for *Hazelcast* is increasing more than linearly (in the log-scale). This is the reason we don't have the run time measure for *Hazelcast* with 1 million operations. The reason for this increase is the scan operator, as we will see, added up to the fact that as insertions are performed in the workload, the database size increases at each run.

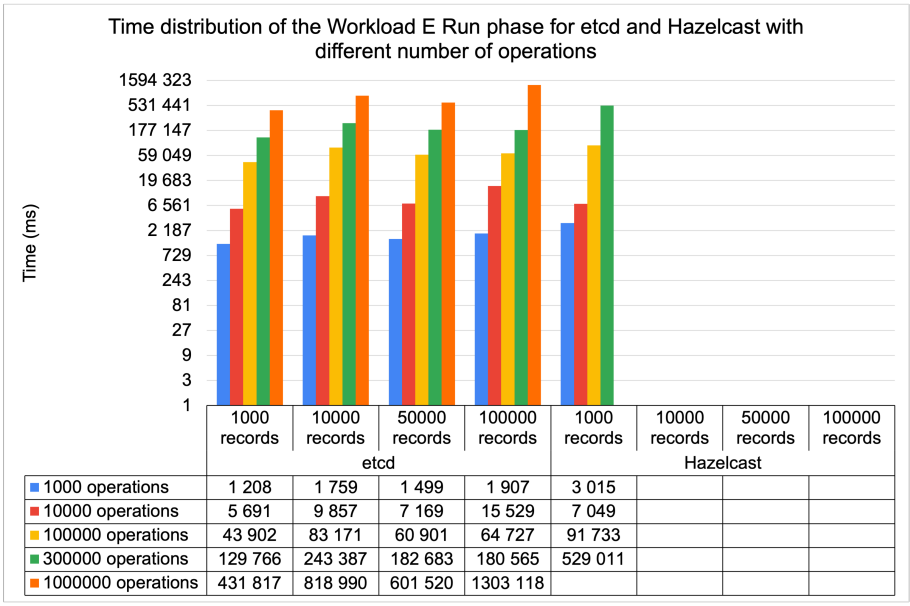
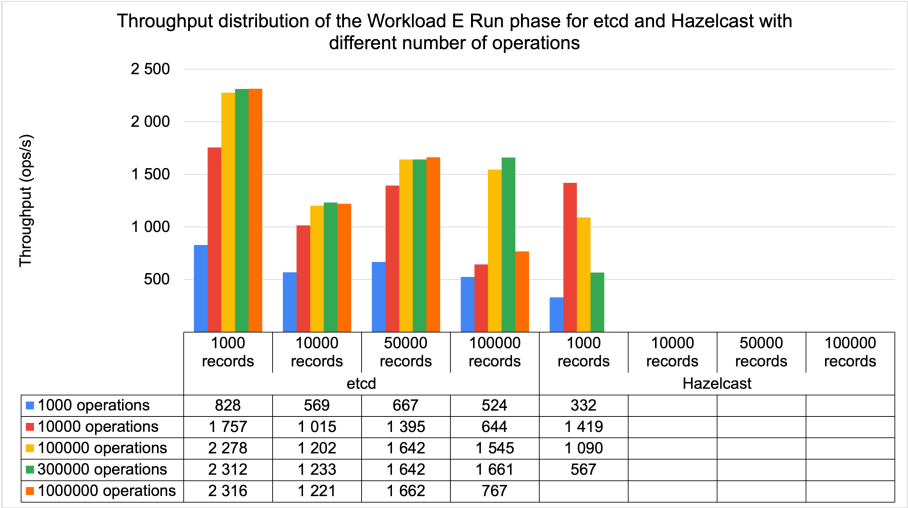


Figure 7.13: Workload E Run phase time distribution

The diagram in Figure 7.14 describes throughput distribution of the Workload E run phase. Here we see more clearly what is happening: the throughput for *Hazelcast* is being severily affected by the amount of operations performed. Also, we observed a huge decrease for the last measure of *etcd*, but we will see that the reasons for these facts are of different nature.

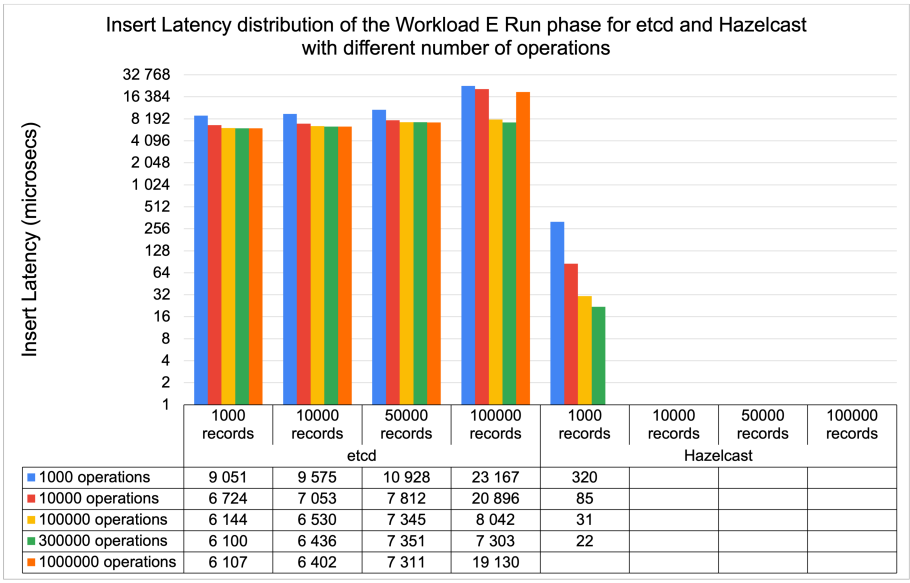
[1] See 3.4 and 5.3.2 for further explanation.



	etcd 1000 records	etcd 10000 records	etcd 50000 records	etcd 100000 records	Hazelcast 1000 records	Hazelcast 10000 records	Hazelcast 50000 records	Hazelcast 100000 records
1000 operations	828	569	667	524	332			
10000 operations	1 757	1 015	1 395	644	1 419			
100000 operations	2 278	1 202	1 642	1 545	1 090			
300000 operations	2 312	1 233	1 642	1 661	567			
1000000 operations	2 316	1 221	1 662	767				

Figure 7.14: Workload E Run phase throughput distribution

The explanation for these will be found in the next two plots, shown in Figures 7.15 and 7.16.



	etcd 1000 records	etcd 10000 records	etcd 50000 records	etcd 100000 records	Hazelcast 1000 records	Hazelcast 10000 records	Hazelcast 50000 records	Hazelcast 100000 records
1000 operations	9 051	9 575	10 928	23 167	320			
10000 operations	6 724	7 053	7 812	20 896	85			
100000 operations	6 144	6 530	7 345	8 042	31			
300000 operations	6 100	6 436	7 351	7 303	22			
1000000 operations	6 107	6 402	7 311	19 130				

Figure 7.15: Workload E Run phase insert latency distribution

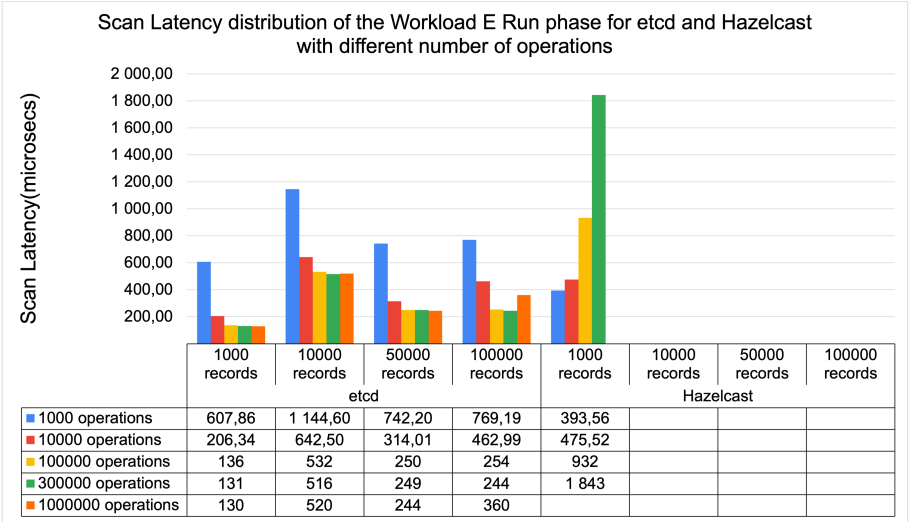


Figure 7.16: Workload E Run phase scan latency distribution

We notice how the insert latency for *etcd* is increasing with the number of records in the database, and how it increases even more for the highest amount of operation and the highest scale factor. For *Hazelcast*, as for the other workloads, better latencies are observed when increasing the amount of operations.

As for the latter graph, it is noticeable that the scan operation for *Hazelcast* is very inefficient, with an increased related to the size of the database, not to the number of operations (but this affects it because among the operations there are new insertions).

In conclusion, we have confirmed our suspicions about the scan operation with *Hazelcast*: it is advisable not to use it, specially with big databases.

Chapter 8

Conclusion

A summary of the characteristics and differences of *etcd* and *Hazelcast* is presented in Table 8.1.

Characteristic	etcd	Hazelcast IMDG
Data model	Key-Value store	More general object store
Data persistency	Yes	Optional
CAP	CP and fairly good A	AP and configurable to CP
Physical storage	On disk B+Tree	In memory structures: IMap for key-value stores
Index	Lexically ordered index on keys	Implicit Hash index on keys
Insertion	Inefficient	Efficient
Read	Efficient	Very efficient
Update	Inefficient	Efficient
Delete	Inefficient	Efficient
Scan	Efficient	Impracticable
Targeted applications	Distributed non-changing values: - Distributed configurations - Discovery services	Distributed highly changing values with high availability: - Passing messages through the cluster - Storage for session data - Distributed data streaming applications

Table 8.1: Characteristics and differences of *etcd* and *Hazelcast IMDG*.

As for the purposes of the project, we have demonstrated the basic behavior of both tools, comparing it with the expected behavior according to the documentation of each tool. Our conclusion in this respect is that, in fact, both tools present the expected behavior and should be use in the scope defined by the corresponding docs.

We have learnt the importance of understanding deeply the structure and model of a tool, because even though we have presented two key-value store, we have seen how they serve totally different purposes and should be carefully chosen when dealing with one problem or another.

Regarding the YCSB benchmark, we believe that even though it is good tool for benchmarking this kind of technologies, it is highly outdated, which makes it more difficult to use, as well as making comparisons with new ideas and trends on the market. For example, even though we tried to execute the test also with PostgreSQL, which is already implemented, we were not able to do so because so

many incompatibilities were detected due to the versions of the tools. In addition, it is not really insightful to compare our current *etcd* implementation with the version 9 of PostgreSQL, which dates from 2015.

In addition, the delete operation is not implemented as a workload operation, even though it is a common operation in a range of use cases.

Thus, we think that YCSB needs both a revision and an update, to cope with new trends, ideas and technologies, and to update those technologies that have evolved since they were implemented into the benchmark.

Bibliography

[Aut21a] etcd Authors. *v3.5 docs*. 2021. URL: https://etcd.io/docs/v3.5/.

[Aut21b] Hazelcast Authors. *Hazelcast IMDG Reference Manual 4.2.6*. 2021. URL: https://docs.hazelcast.com/imdg/4.2/.

[Bre00] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Vol. 7. 10.1145. Portland, OR. 2000, pp. 343477–343502.

[Coo+10] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. ACM Press, 2010. DOI: 10.1145/1807128.1807152.

[Haz22] Inc. Hazelcast. *Hazelcast Manual version 5.2. Installing Hazelcast Open Source*. Palo Alto, Mar. 2022. URL: https://docs.hazelcast.com/hazelcast/latest/getting-started/install-hazelcast.

[JA13] Garima Jaiswal and Arun Prakash Agrawal. "Comparative analysis of Relational and Graph databases". In: *IOSR Journal of Engineering (IOSRJEN)* 3.8 (2013), pp. 25–27.

[Liu+15] Si Liu et al. "Quantitative Analysis of Consistency in NoSQL Key-Value Stores". In: *Quantitative Evaluation of Systems*. Ed. by Javier Campos and Boudewijn R. Haverkort. Cham: Springer International Publishing, 2015, pp. 228–243. ISBN: 978-3-319-22264-6.

[OO14] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm (Extended Version)*. 2014. URL: https://raft.github.io/raft.pdf.

[SU09] Marc Seeger and S Ultra-Large-Sites. "Key-value stores: a practical overview". In: *Computer Science and Media, Stuttgart* (2009).