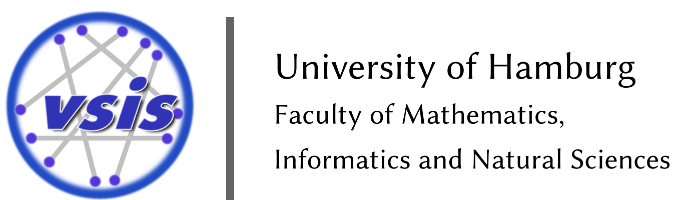

Bachelor Thesis:

Simulation of Replication and Sharding with the SickStore Database System

Markus Fasselt

Matriculation number: 6310552
1fasselt@informatik.uni-hamburg.de
Bachelor of Science Informatics

1. Reviewer:	Prof. Dr.-Ing. Norbert Ritter
2. Reviewer:	Wolfram Wingerath
Supervisors:	Wolfram Wingerath and Steffen Friedrich

Abstract

With the increasing amount of available NoSQL database system, a proper evaluation is necessary to find the one that suits the application context best. For that decision, the performance of a system is an important factor and needs to be analyzed with benchmark tests.

Because of the distributed nature of many NoSQL systems, the development of appropriate benchmark tools is difficult and error-prone. Therefore, the prototypical key-value store SickStore was introduced. It simulates a distributed database system and can be used to validate the accuracy of benchmark tools. The objective of this work is to extend SickStore with a simulation of replication and sharding, that emulates MongoDB's behavior.

The simulation of replication considers two parameters of MongoDB's write concern: the number of replica acknowledgments and whether a journal commit is required. The effects of different write concerns on the write latency are estimated and validated. Subsequently, the implementation into SickStore is described.

In case of the sharding simulation, the nature of MongoDB and the implications on the performance are not clearly identifiable. Therefore, the simulation does not consider MongoDB's behavior and only a basic implementation of hash- and range-based sharding is given.

Abstract in German (Zusammenfassung)

Mit der zunehmenden Menge an verfügbaren NoSQL Datenbanksystemen, ist eine genau Evaluierung notwendig, um das System zu finden, dass am besten in den Anwendungskontext passt. Die Performanz des Systems ist hierbei ein wichtiger Faktor für die Entscheidung und muss daher mit Benchmarktests analysiert werden.

Aufgrund der verteilten Architektur vieler NoSQL Systeme ist die Entwicklung von geeigneten Benchmarktests schwierig und fehleranfällig. Aus diesem Grund wurde das prototypische Datenbanksystem SickStore vorgestellt. Es simuliert ein verteiltes Datenbanksystem und kann verwendet werden, um die Genauigkeit von Benchmark-Tools zu validieren. Das Ziel dieser Arbeit ist es, SickStore mit einer Simulation von Replikation und Sharding zu erweitern, die an MongoDB's Verhalten angelehnt ist.

Die Simulation der Replikation berücksichtigt zwei Parameter des so genannten *Write Concerns* von MongoDB: die Anzahl der notwendigen Bestätigung von Replikas und ob ein Schreibvorgang in das so genannte *Journal* geschrieben sein muss. Die Effekte verschiedener Write Concerns auf die Schreiblatenz werden abgeschätzt und validiert. Anschließend wird die Implementierung in den SickStore beschrieben.

Im Falle von Sharding, sind das Verhalten von MongoDB und die Auswirkungen auf die Performance nicht eindeutig feststellbar. Aus diesem Grund berücksichtigt die Simulation das Verhalten von MongoDB nicht und es wird ausschließlich eine simple Implementierung von *Hash*- und Bereichsbasiertem Sharding vorgenommen.

Table of contents

List of Tables IX

List of Figures IX

List of Listings IX

List of Abbreviations XIII

List of Symbols XIII

1. Introduction 1

1.1. Motivation . . . 1
1.2. State of Research . . . 2
1.3. Goals . . . 3
1.4. Chapter Outline . . . 4

2. Foundations of Distributed Database Systems 5

2.1. Types of Distributed Systems . . . 5
2.2. BASE . . . 6
2.3. CAP Theorem . . . 7
2.4. Data Partitioning . . . 8
2.4.1. Vertical Partitioning . . . 8
2.4.2. Horizontal Partitioning & Sharding . . . 9
2.4.3. Sharding Algorithms . . . 10
2.5. Replication . . . 11
2.5.1. Tasks . . . 12
2.5.2. Classification of Replication Strategies . . . 12
2.6. Combining Sharding and Replication . . . 13

3. Architecture of SickStore 15

3.1. SickStore Server . . . 15
3.1.1. Virtual Nodes . . . 16
3.1.2. Data Storage . . . 16
3.1.3. Replication . . . 17
3.1.4. Anomalies . . . 17
3.2. SickStore Client . . . 17
3.3. Technologies . . . 18

4. Simulating Replication with SickStore **19**
4.1. Replication in MongoDB . . . 19
4.1.1. Write Concerns . . . 20
4.2. Estimation of MongoDB's Write Latencies with Different Write Concerns . . . 21
4.2.1. Unacknowledged and Acknowledged . . . 23
4.2.2. Replica Acknowledged with a Specific Number of Nodes . . . 23
4.2.3. Replica Acknowledged with a Tag Set . . . 24
4.2.4. Journal Commit . . . 25
4.2.5. Journal Commit and Replica Acknowledgments . . . 25
4.3. Test Setup for Validation . . . 26
4.3.1. Simulation of Network Delays . . . 28
4.4. Validation of Write Latencies . . . 30
4.4.1. Replica Acknowledged with a Specific Number of Nodes . . . 30
4.4.2. Replica Acknowledged with a Tag Set . . . 31
4.4.3. Journal Commit . . . 31
4.4.4. Journal Commit and Replica Acknowledgments . . . 32
4.5. Data Staleness . . . 34
4.6. Implementation into SickStore . . . 35
4.6.1. Write Concerns . . . 35
4.6.2. Anomaly Generator . . . 36
4.6.3. Client Delay Calculation . . . 36
4.6.4. Calculation of the Observable Replication Delay . . . 36
4.6.5. Calculation of the Journaling Delay . . . 38
4.6.6. Staleness Generator . . . 38
4.6.7. Validation with YCSB . . . 39
4.7. Evaluation . . . 40

5. Simulating Sharding with SickStore **43**
5.1. Sharding in MongoDB . . . 43
5.1.1. Range-based sharding . . . 44
5.1.2. Tag-aware sharding . . . 44
5.1.3. Hash-based sharding . . . 45
5.2. Implementation into SickStore . . . 45
5.2.1. Considerations on MongoDB's Sharding . . . 45
5.2.2. Sharding with SickStore . . . 46
5.2.3. Sharding Router . . . 47
5.2.4. Hash-based sharding strategy . . . 48
5.2.5. Range-based sharding strategy . . . 48
5.3. Evaluation . . . 49

6. Conclusion **51**
6.1. Summary 51
6.2. Future Prospects 52

A. Listings **53**

Bibliography **61**

Declaration **65**

List of Tables

4.2. Write Concern Levels in MongoDB [Mon15]. 21
4.3. An example of the replication latency calculation. 24
4.4. List of nodes with their tags. 25
4.5. Delays used to validate replication latencies. 30
4.6. Expected and measured replication latencies. 31
4.7. Used delays for the validation with YCSB. 39

List of Figures

2.1. Vertical Partitioning 9
2.2. Horizontal Partitioning / Sharding 9

3.1. Basic architecture of the SickStore server. 15

4.1. Replication with MongoDB as illustrated in [Mon15] 19
4.2. Write operation with two acknowledgments. [Mon15] 22
4.3. Replica set with n secondaries 23
4.4. Illustration of journal commit intervals. 25
4.5. Test setup with four nodes to validate MongoDB’s latencies. 26
4.6. Illustration of netem delays 28

5.1. Basic architecture of a sharded MongoDB cluster [Mon15]. 43
5.2. Ranged-based sharding in MongoDB [Mon15]. 44
5.3. Hash-based sharding in MongoDB [Mon15]. 45
5.4. Architecture of the SickStore server with three shards. 47

List of Listings

4.1. Command to start a new Docker container. 27
4.2. Command to start the Mongo-client on the primary. 27
4.3. Command to retrieve an IP address of a Docker container. 28
4.4. Command to start the replica set and to register its members. 28
4.5. Setting a delay on all outgoing packages of a specific network interface. 29
4.6. Output of `ip link` on the host machine with four running containers. 29
4.7. Output of `ip link` inside a Docker container. 30
4.8. Command to perform a write operation and to measure its time. 30
4.9. Starting a MongoDB container with a specific journal commit interval. 31
4.10. Script to validate the journal commit interval. 32
4.11. Results of the script from Listing 4.10. 32
4.12. Script to validate the combination of journaling and replication. 33
4.13. Latencies of a write operation with a higher replication than journaling delay. . 34
4.14. Latencies of a write operation with a higher replication than journaling delay. . 34
4.15. Calculation of the write delay, based on the replication and journaling delay. . . 36
4.16. Calculation of the replication delay. 37
4.17. Calculation of the delay that emerges from a required journal commit. 38
4.18. Commands to execute YCSB benchmark. 39

5.1. Determination of the responsible shard within a hash-based sharding. 48
5.2. Determination of the responsible shard within a range-based sharding. 49

A.1. Script to automate the replica set setup. 53
A.2. Script to detect Docker’s virtual network interface names. 54
A.3. Selection of the delays that are relevant for the current operation. 55
A.4. Calculation of the observable replication delay that arises for the given number of acknowledgments and custom delays. 56
A.5. Calculation of the observable replication delay that arises from a write operation with tag set acknowledgment. 57
A.6. Selection of all custom delays of nodes that have the requested tag. 58
A.7. Generation of the staleness map. 58
A.8. The `doScanRequests()` method of the hash-based strategy executes the scan request on all shards and combines the results. 59
A.9. The `doScanRequests()` method of the range-based strategy begins with the responsible shard for the incoming key and queries all successors if necessary. . . 60

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability

BASE Basically Available, Soft state, Eventually consistent

CAP Consistency, Availability, Partition tolerance

CPU Central Processing Unit

DNS Domain Name System

I/O Input / Output

IP Internet protocol

ms milliseconds

SSH Secure Shell

SSHFS Secure Shell File System

YCSB Yahoo! Cloud Serving Benchmark

List of Symbols

Figure

Table

</> Listing

1. Introduction

1.1. Motivation

With the rapid growth of many application systems, it becomes an important task to scale-up these systems. One important aspect when it comes to scaling is the increasing amount of data which needs to be processed, especially in the context of Big Data. Classical relational database systems, which follow the ACID[1] properties, store their data in tuples and relations. At some point these systems do not scale any longer and other solutions have to be found.

Given the demand on data scaling, new approaches were developed which differ from classical database systems and are often described as **NoSQL**. These databases do not follow the traditional relational data model and use a data structure which fits exactly into the use case of the database. They often drop the ACID properties in favor of scalability. As a result, many NoSQL database systems were developed to solve a specific problem.

Based on the increasing amount of available NoSQL databases, it is difficult to choose the right system for a particular use case. There are many parameters which need to be considered to choose the right system, such as the data model and the availability characteristics. Another important parameter is the performance of the selected database system. It describes how a system reacts under load, such as the effect on the response time, the throughput or (especially for distributed databases) the time to synchronize the nodes. The performance cannot be defined or configured – it needs to be measured in a benchmark, as it depends on many uncontrollable factors.

While there are already many benchmark tools for relational database systems (compare [DPCC13]), the development of benchmark tools for NoSQL databases has just started. First benchmarks measure the read-write-throughput[2] or the staleness[3] of data in distributed systems. However, there are many other aspects which could not be measured at the moment and need separate benchmarks in the future. One main advantage of most NoSQL databases is the scalability in distributed environments to manage the increasing amount of data. This sets high expectations into benchmarks for distributed NoSQL database systems, as these need to create a load which challenges all nodes under test. Furthermore, the measurement of some performance characteristics, such as the staleness window between write and read on different nodes, requires extra communication between the test clients, which must not influence the test results.

The requirements on benchmarks for distributed systems make the development of those difficult and error-prone. The authors in [PPR+11] tried to measure the staleness window of two distributed database systems with their benchmark tool *YCSB++* and concluded that the staleness

[1] Atomicity, Consistency, Isolation, Durability

[2] The read-write-throughput describes how many read and write request could be processed in a specific amount of time.

[3] The *staleness* states whether and how long a data item is out of date.

window in the tested systems increases with an increasing workload. However, [WFGR15] discovered that the measured staleness window is actually quite imprecise. The results of YCSB++ were distorted by an erroneous implementation.

In order to avoid and detect errors in benchmarks, there have to be tools which can be used to test benchmark systems and to validate their results. Those tools can give an indication of the accuracy of the results of a benchmark. As there is a demand on such tools, I will focus this work on extending the prototypical NoSQL database system **SickStore** to simulate more realistic scenarios. SickStore helps to validate the results of benchmark systems by simulating configurable data anomalies, which can be compared with the actual measured results.

1.2. State of Research

The first widely used benchmark tool for NoSQL databases – **YCSB**[4] – was published in 2010 by *Yahoo!* in [CST+10]. It measures the read-write-throughput of a NoSQL database system by stressing the database with a specific workload and measuring the time needed to complete all operations. In the original paper, the database systems *Cassandra*, *HBase*, *PNUTS* and sharded *MySQL* were considered. Due to the extensibility of YCSB it is easily possible to embed and test other database systems. By now, many other popular database systems are embedded into YCSB such as MongoDB, DynamoDB and Elasticsearch.

Based on YCSB, there were other attempts to make more aspects of NoSQL databases measurable. [PPR+11] describes **YCSB++**, a tool which measures amongst other things the staleness of data in distributed database systems. The staleness is measured by writing data with one client and measuring the time interval until another client can successfully read the data. Since extra communication is needed between the clients to coordinate the measurement, [WFGR15] assumed and confirmed that the implementation is erroneous and distorted the results, as YCSB++ often checks the same data items in direct succession and does not request other items for tens of seconds.

Classical NoSQL databases, which follow the BASE[5] principle, normally do not have transaction support. Nevertheless, first NoSQL databases emerged which offer a limited transaction support. For example, there are single-row transactions in HBase and document-based transactions in MongoDB. There are first attempts to provide full transaction support within these two systems by implementing the transaction logic in the client application or by using existing libraries. For HBase, there are for instance the libraries Tephra[6] and Haeinsa[7], which provide cross-table transactions. MongoDB provides a detailed example on how to perform multi-document transactions in its documentation[8]. To measure the transactional overhead, [DFNR14] presents the

[4] Yahoo! Cloud Serving Benchmark
[5] Basically Available, Soft state, Eventually consistent
[6] `http://tephra.io/`
[7] `https://github.com/VCNC/haeinsa`
[8] `http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/`

YCSB extension YCSB+T. It measures the latency of start-, abort- and commit-methods and the performance of each individual database operation when it is executed within or outside of a transaction. Furthermore, it adds a validation phase to ensure data integrity. As [FWGR14] observed, the validation in YCSB+T is quite rudimentary and many data anomalies could not be detected.

To recognize errors in different benchmark tools, [WFGR15] introduced the prototypical NoSQL database system **SickStore**, a "single-node key-value store which is able to simulate the behavior of a distributed key-value store". Currently, it is able to simulate data staleness with a fixed time interval.

1.3. Goals

SickStore is currently able to simulate data staleness with a fixed value, which in return can be measured with a benchmark. To simulate more realistic behaviors, the main goal of this work is to extend SickStore with a simulation of replication and sharding. To make this possible, the delays that are produced through replication and sharding should be configurable for each node[9] to simulate different distribution strategies with the SickStore. The delay calculation for this simulation will be oriented on the behavior of the MongoDB database system.

Replication makes it possible to distribute the load and to handle a possible node failure by copying the data across multiple nodes. According to [GHOS96], there are two parameters to classify replication strategies: *when* and *where*.

The parameter *where* describes whether it is possible to write on each replicated node[10] (master-master replication) or whether there is only one master that accepts write operations (master-slave replication).

The parameter *when* describes whether data is propagated directly or delayed to the replica. With *eager replication*, written data is immediately copied to and persisted on all replica. A write request succeeds only if each replica has confirmed the request. Therefore, there will never be stale data on any node, but a write operation will need extra time to complete. On the other hand, with *lazy (or asynchronous) replication*, written data is propagated delayed to the replicas. A write request does not need extra time to wait for the propagation and distribution of the data to all replica, but there is no guarantee when data will be visible on another node. Due to that, it is possible to read stale data by reading from a replica.

Additionally, sharding will be implemented into SickStore. With sharding it is possible to manage huge amounts of data by partitioning the data across multiple nodes. Therefore, a single data item will only be available on one node. As a data item is only stored on one node, sharding can be combined with replication to prevent data loss on the failure of a node. In this case, each partition has its own replicated nodes.

[9] In the context of SickStore, the term "node" means a simulated node and not a dedicated node
[10] In the following, a replicated node will be named a *replica*

1.4. Chapter Outline

In the first chapter I will describe the motivation of my work, give an overview about the current state of research, define the goals and give a chapter outline. In order to fulfill the goals of this work, we will first investigate the foundations of replication and sharding in distributed database systems in Chapter 2. To describe the implementation in SickStore, I will at first describe its architecture in Chapter 3. Afterwards, in Chapter 4, the replication system of MongoDB will be described and an estimation of the appearing delays given. Furthermore, the estimation will be validated and the implementation into SickStore described. Chapter 5 will describe the sharding functionality of MongoDB and how it was implemented into SickStore. Finally, the results of this work will be summarized and a prospect for further works given in Chapter 6.

2. Foundations of Distributed Database Systems

There are various ways to build distributed database systems. Basically, different combinations of replication and sharding algorithms are used to solve various failure and consistency scenarios. To explain the simulation with SickStore later, we will at first investigate the foundations of distributed database systems in this chapter.
Therefore, we will examine different types of distributed systems in Section 2.1 and clarify the term BASE and the CAP theorem in Sections 2.2 and 2.3. We will then explore the basics of data partitioning across multiple nodes in Section 2.4. Afterwards, the foundations of data replication will be explained in Section 2.5. Finally, the combination of sharding with replication will be described in Section 2.6.

2.1. Types of Distributed Systems

To scale a database system, different parameters need to be considered. At first, the type of the database system has to be determined. [Sto86] describes three main system types: *shared-memory*, *shared-disk* and *shared-nothing*.

A database system following the **shared-memory** approach runs on a single server and can only be scaled up by hardware upgrades, which is called *vertical scaling* or *scale-up*. The server could get a faster or an extra CPU[1], additional main memory or a bigger hard drive, depending on the piece of hardware that cannot handle the load anymore. The scaling of this approach is limited by the performance of currently available hardware and can become very expensive. In addition, the whole database system is no longer available if the server fails. Overall, this approach does not assist in scaling a database system in terms of huge amounts of data and to provide higher reliability.

The **shared-disk** approach depends on a single shared disk between several independent nodes. Each node has its own main memory but the disk and data are shared between all nodes. This has the advantages that even if one server fails, the database system is still able to operate. However, this architecture introduces the problem that the same data can be held in different buffer pools. This leads to a high coordination overhead between the nodes to ensure that all buffer pools are refreshed after an update. Thus, this approach is restricted to a small number of nodes, typically fewer than 10 [SC11].

In contrast to the previous approaches, a **shared-nothing** architecture runs across completely independent nodes. Each node has its own processor, main memory and hard disk. The nodes are only connected among each other via a network connection. Therefore, a shared-nothing system

[1] Central Processing Unit

can be scaled by adding further nodes into the cluster. However, this system type leads to high communication costs between the nodes and increases with the number of nodes. This type of scaling is called *horizontal scaling* or *scale-out* and is the desired architecture of most distributed NoSQL systems. For this reason this work will only consider the *shared-nothing* approach and neglects the other ones.

A shared-nothing architecture can be established by distributing or, to be more precise, partitioning the data across multiple nodes, which will be explained in Section 2.4. In Section 2.5, the foundations of replication will be explained, which is useful to distribute the load and to handle a node failure.

2.2. BASE

Many traditional relational database systems follow the strict **ACID** (Atomicity, Consistency, Isolation, Durability) properties which guarantee a strong consistency in highly concurrent environments. ACID systems are best suited for transactional systems which need a high level of data integrity, such as bank transactions, billing systems and order processing. Nevertheless, fulfilling the ACID properties has a huge impact on performance and availability of a system. A distributed ACID system cannot be highly available, as the consistency cannot be ensured any longer and the system refuses to operate in the case of a node failure or network partition. However, many services do not necessarily need the high level of consistency an ACID system provides. In many current application systems the availability of the service is more important than data consistency, since users expect services to be always available. All in all, the ACID properties do not fit to describe systems which should be highly-available.

For that reason, the acronym **BASE** (Basically Available, Soft state, Eventually consistent) was introduced in [FGC+97] as a counterpart to ACID. BASE systems focus on the availability of the database and accept stale data and inconsistencies if the database can eventually be in a consistent state. Furthermore, it handles failures with less complexity and gives opportunities for better performance in comparison to ACID, as the system becomes more simplistic by weakening the consistency. Most of the available NoSQL databases follow the BASE semantics.

The acronym BASE consists of three attributes:

- **Basically available**: The database system is able to answer requests, even if the answer contains stale data.
- **Soft state**: The state of the system (the data) could change over time, even without input, because of the eventual consistency.
- **Eventual consistency** indicates that the system will become consistent at one point in the future if no more inputs arrive.

2.3. CAP Theorem

The CAP[2] theorem was first published in 1999 by Fox and Brewer [FB99]. One year later, Brewer presented the theorem at the keynote of the Annual ACM Symposium on Principles of Distributed Computing in 2000 [Bre00]. The authors made the conjecture that only two of the following three guarantees are fulfillable for a distributed system:

- **Consistency** (*C*): A system is consistent if each node sees the same data at the same time.
- **Availability** (*A*): A system is available if all read and write requests can be answered in a reasonable amount of time, even if the answer indicates an error.
- **Partition Tolerance** (*P*): A system is partition tolerant if it continues to work even if the cluster is separated into multiple partitions.

In distributed systems it is desirable to fulfill them all, which is actually not possible; at most, two guarantees are achievable at a time (which was formally proved in [GL02]). Therefore, only the following three combinations are possible:

- Consistency & Availability (**CA**)
- Availability & Partition Tolerance (**AP**)
- Consistency & Partition Tolerance (**CP**).

CA systems strive after a high availability and consistency. For that reason, such systems are unable to handle network partitions. Classical relational database systems follow these properties, since they often run on a single server or in small high-available networks, in which the case of a network partition is very unlikely.

In distributed systems which are connected across wide-area networks, network partitioning cannot be prevented. For that reason, such systems have to be partition tolerant and it is only possible to choose between availability (AP) and consistency (CP). **AP** systems sacrifice the consistency of a system in favor of availability. Therefore, such systems send outdated data items rather than sending no answer at all. It is expected, that in such environments, inconsistent data does not lead to bigger problems and that a user is allowed to see outdated data. In contrast, **CP** systems sacrifice the availability in favor of the consistency. This has the effect that the whole system will be unavailable or refuses to answer requests in case of a node failure or network partition, as the system cannot be know for sure whether there might be a recent data item.

The CAP theorem was often criticized since its publication (examples[3][4][5]). For that reason, Brewer published an update in 2012 [Bre12] in which he evaluated the impact and the interpretations of the original CAP theorem. He states that CAP fulfilled its purpose and opened the minds of system designers. It further led to a wide range of novel distributed systems. However, he admits that the “two of three” formulation was misleading, as it oversimplifies the problem. As

[2] Consistency, Availability, Partition tolerance
[3] `http://codahale.com/you-cant-sacrifice-partition-tolerance/`
[4] `http://voltdb.com/blog/clarifications-cap-theorem-and-data-related-errors`
[5] `http://dbmsmusings.blogspot.de/2010/04/problems-with-cap-and-yahoos-little.html`

partitions are rare, there is no reason to forfeit consistency or availability when there are no partitions. In case of a partition, the choice for consistency and availability can be made on various granularities. On the one hand, different subsystems can make different choices. On the other hand, the choice can be based on the operation, the specific data item or the involved user. Moreover, the choice between consistency and availability is not binary; there are different levels of both. If the system should stay consistent, it is necessary to limit some operations, which reduces the availability. In contrast, if the system should stay available, it is necessary to record data changes for later recovery.

2.4. Data Partitioning

When the amount of data becomes too large to be managed by a single computer, the data needs to be partitioned across multiple nodes. There are different partitioning[6] strategies, but all aim for one main goal: distributing the data *evenly* across all nodes. However, there are different meanings of an "even distribution": on the one hand, the data can be clustered into similar sized partitions, so that each node holds an (almost) identical amount of data. On the other hand, the data can be clustered by the number of users accessing the data, so that each node processes an equal amount of requests. [SF12]

[LS13] defines three general requirements on data partitioning:

- **Completeness:** All tuples or columns belong to at least one partition.
- **Reconstructability**: The original relations can be recreated from the partitions.
- **Disjoint**: Data should never be stored in more than one partition.

There are two general ways of data segmentation: *vertical* and *horizontal partitioning*, which will be explained in the following.

2.4.1. Vertical Partitioning

One data distribution strategy is vertical partitioning. According to [LS13], the data is split by its attributes (or columns) into different partitions. As a result, each partition always contains all data items but with a limited set of attributes. For example, customer information can be split into two partitions: one partition holds the information which is needed for order processing and another partition contains the marketing-related attributes. Figure 2.1 illustrates how vertical partitioning might look like: the primary key pk is contained in each partition and the attributes a_i $(1 \leq i \leq 4)$ are split into two partitions.

Vertical partitions are specified by a projection of the original data into different partitions. Due to the disjoint requirement on data partitioning, each projection must hold different attributes with only one exception: the primary key is allowed to and must be contained in each partition. Otherwise it would be impossible to match data items in different partitions, that belong

[6] Often, partitioning is also described as "fragmentation".

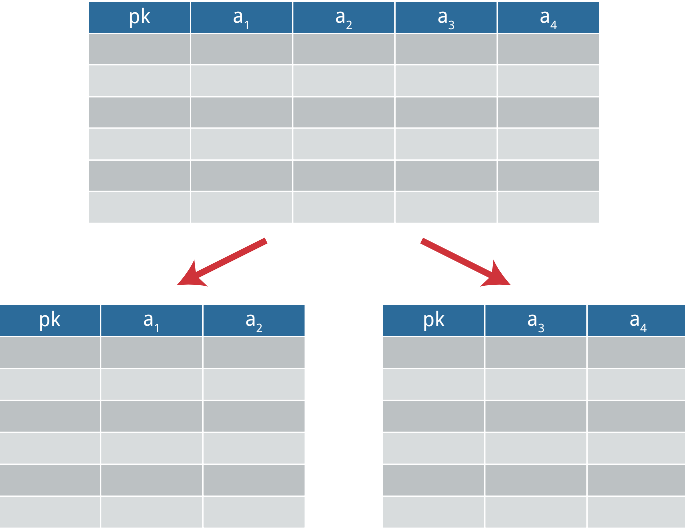

Figure 2.1.: Vertical Partitioning

together. Therefore, the disjoint requirement must be weakened for vertical partitioning to allow the primary key to be redundant in each partition. [LS13]

However, as vertical partitioning is not relevant in the context of this work, this strategy will not be explored any further.

2.4.2. Horizontal Partitioning & Sharding

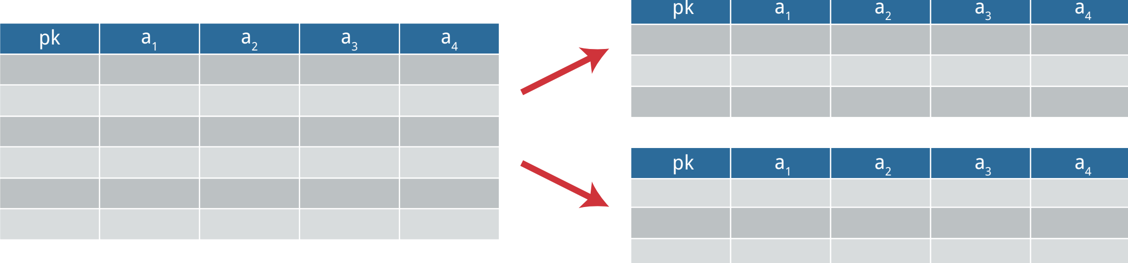

Figure 2.2.: Horizontal Partitioning / Sharding

In contrast to the vertical partitioning strategy, the horizontal partitioning strategy does not split up a single data item into multiple items with different attributes. Instead, [LS13] defines horizontal partitioning as a strategy which divides the whole data set into multiple subsets, which only contain complete data items (tuples). Each partition contains only a subset of all items, as illustrated in Figure 2.2. This strategy is called **sharding**, if the partitions are placed on multiple independent nodes in a shared-nothing environment (see Section 2.1). In this case, the partitions are also called *shards*. Sharding is the most commonly used strategy in large-scale systems.

In the past, sharding was implemented as part of the application logic. However, this has the crucial disadvantage that nodes cannot be added easily. One the one hand, the application logic has to be adjusted to consider newly added nodes. On the other hand, the existing data has to be

rebalanced manually, so that the data is again distributed evenly across all available nodes. For that reason, most NoSQL systems offer automatic sharding [SF12].

Sharding helps in several ways to increase the database performance and reliability: [Hof09, SF12]

- The data is kept in small chunks which allows a faster access and is easier to manage.
- Sharding allows a horizontal scaling of writes, as ingest operations can be spread across all available nodes.
- More work can be done in parallel, as each shard runs on an independent node.
- If one shard fails, only the data on that single node becomes unavailable, but the database can still continue to operate.

[SF12] defines some general demands to ensure optimal sharding. Basically it is essential to find an ideal granularity for each partition. Within a perfect distribution, each node would process an even amount of users, requests and data. It is further desirable to store data that is commonly accessed together or in sequence on the same node. In globally distributed systems it is also preferable that the data is stored geographically near to the requesting user in order to reduce request latencies. However, it is almost impossible to find a perfect distribution; there are always trade-offs which need to be considered. Therefore, the decision of which partitioning algorithm to use and the granularity of the different shards is highly domain specific. These parameters must be evaluated carefully for each use-case.

2.4.3. Sharding Algorithms

There are different methods to split data into single partitions. On the one hand, it is possible to build the partitions based on values of specific attributes of a data item. On the other hand, it is also possible to distribute data items randomly. There are different algorithms that can be used to partition the data. Based on [LS13], the range-based and hash-based partition will be described in the following.

Range-Based

With the range-based sharding, data items are assigned to a specific partition based on the value of one or more attributes. The partitions can have an equal or an unequal size and contain the data items which have values within a certain interval. For example, partitions can be built by the first letter of the customer's name or by the year of the order date. Range-based sharding is often used, as in the latter example, if there is some sort of data aging. In this case, older entries which are not accessed that often anymore, can be stored on partitions with less redundancy, less availability and slower hardware to reduce costs. However, range-based sharding requires knowledge of the application domain to properly adjust the ranges. Therefore, this method does not fit for every use case.

Hash-Based

Hash-based sharding provides a quick and easy way of sharding for any application size. This method applies a hash function to the values of one or more attributes and uses the resulting hash to determine the target partition. In so doing, all data items will be distributed evenly and randomly across the available nodes. It also allows a quick read access to a data item, as the hash defines exactly where the item is located. Hash-based sharding is further used if no information about the application domain is available. For that reason, this method is well suited for generic database systems of any scale.

Sharding can be combined with replication to improve the failure tolerance, reliability and availability of a shard. Therefore, we are going to explore the foundations of replication in the next section.

2.5. Replication

Replication is the process of mirroring data from one node to other nodes during the runtime of a database system. This is necessary, as computer systems can fail at any time, which happens for example when the power supply is interrupted or a hardware component breaks. For that reason, it is important to consider this case and to make arrangements to prevent a downtime or data loss. The risk of the latter can be reduced by making regular backups. Nevertheless, a downtime after a failure is unavoidable if the whole database system is running on a single node. To avoid this scenario, data can be replicated to other nodes which take over if the primary node fails [LS13].

[LS13] describes more benefits which come with replication:

- **Reliability**: The database system becomes more reliable, as it does not depend on the availability of a single node.
- **Durability**: Replication can improve the durability of data, as replicated data does not get lost in case of a disk failure on one node. (Nevertheless, it does not replace an actual backup.)
- **Load balancing**: Replication can improve the performance of the whole database system, especially the read performance, by spreading the requests across all available nodes. The performance can be increased further by using regional replicas which have a lower distance, and therefore a lower request latency, to the requesting user.

Apart from the benefits of replication, some new problems are introduced. In practice it is a difficult task to create a reliable replication system: a higher effort for updates is required and it is necessary to make a compromise between consistency and availability, which is reasoned by the CAP theorem (compare Section 2.3). Furthermore, maintaining replicas requires additional storage space. [LS13]

Therefore, we will examine the various strategies and trade-offs in the following sections.

2.5.1. Tasks

According to [LS13], the main task of all replication strategies is to keep the replicas consistent. This includes the following subtasks:

- The **update propagation** is responsible for making all replicas aware of data updates.
- The **concurrency control** task tries to ensure an always consistent state for the database system, even in highly concurrent environments with parallel transactions on different replicas.
- The **failure detection and recovery** task detects node failures and network partitions and tries afterwards to bring all nodes back into a consistent state.
- The **handling of read transactions** task is responsible for the selection of an *optimal* replica or set of replicas to answer a read request. Typically, there is trade-off between fast access and the freshness of data.

However, there is no general way to accomplish these tasks, due to various trade-offs between them. Each replication strategy has a different focus and tries to solve another problem. It always depends on the application and the objective which strategy should be used.

2.5.2. Classification of Replication Strategies

According to [GHOS96], there are basically four different replication strategies, which could be classified by two parameters: the *data ownership* (*where?*) and the *propagation strategy* (*when?*). The data ownership describes which node owns a data item and is therefore allowed to update that item. The propagation strategy defines when data updates will be disseminated to the replicating nodes.

Data Ownership

Then data ownership parameter describes *where* data can be updated, i.e., on which node.. Based on [GHOS96], there are two possible solutions: *Primary Copy* and *Update Anywhere*.

With the **Primary Copy**[7] approach, each data item has one owning node[8] which manages the *primary copy* of the data. It is only allowed to update the primary copy; all replicas are read-only and receive data changes only from the master after an update. [WSP+00] noticed that such a system introduces a single point of failure and a bottleneck. With the failure of the primary, it becomes impossible to write new data. A single point of failure can be prevented by introducing an election protocol that elects a new primary after the current one failed. Furthermore, a bottleneck can be avoided by assigning different data items to different master nodes.

The **Update Anywhere**[9] approach pursues a different method: a data item does not belong to any particular node and can be updated anywhere. This provides higher flexibility than the

[7] The Primary Copy approach is often also describes as Master-Slave replication.
[8] The owning node is also called master or primary.
[9] The Update Anywhere approach is often also described as Master-Master replication.

primary copy approach, but requires in reverse a higher communication and coordination effort for the concurrency control.

Propagation Strategy

The propagation strategy parameters describes *when* data will be disseminated to the replicating nodes. [GHOS96] describes two solutions: *eager* and *lazy* replication.

An **eager replication**[10] protocol propagates updates to the replicas directly as part of the original transaction. An update succeeds only if all replicas confirmed and processed the operation. Therefore, it becomes certain that there are no inconsistencies between the nodes, as each node always sees the same data. However, there is a reduced write performance in an eager replication system. [GHOS96] observed that eager replication does not scale and a "ten-fold increase in nodes and traffic gives a thousand-fold increase in failed transactions (deadlocks)".

In contrast, **lazy replication**[11] propagates data updates asynchronously to all replicas. A write request can be confirmed once the request was processed on the first node. Eventually the transaction is then propagated to the replicas. By this, inconsistencies can arise and it is possible to read stale data from an outdated replica.

2.6. Combining Sharding and Replication

Sharding by default is not very fault tolerant. If one shard fails, the data on that shard becomes unavailable and can get lost. Nevertheless, the overall system is still able to operate, even if some parts of the data are missing or unavailable. As some application contexts require higher failure tolerances, it is possible to combine sharding with replication.

By replicating single shards, those become more failure tolerant. If one replica fails, another one can take over the work. In the overall system, each shard can have a different replication configuration, depending on the business case and the requirements on data availability.

So far, the previous chapter outlined the foundations of distributed database systems and two strategies were introduced, that allow to scale up a database system. In the following chapter, the prototypical database system SickStore and its architecture will be introduced.

[10] Eager replication is often also described as synchronous replication.
[11] Lazy replication is often also described as asynchronous replication.

3. Architecture of SickStore

This chapter will give an introduction into the architecture and functionality of SickStore. This is necessary to understand the implementation and simulation of replication and sharding in the following chapters. Therefore, the client and server application will be explained in Section 3.1 and Section 3.2. Finally, a short overview about the used technologies is given in Section 3.3.

SickStore (**s**ingle-node **inc**onsistent **k**ey-value **store**) is basically a key-value store that runs on a single computer, but simulates the behavior of a distributed database system. It is able to simulate the anomalies of distributed database systems, but has in fact only a single consistent state. It was first published in [WFGR15].

3.1. SickStore Server

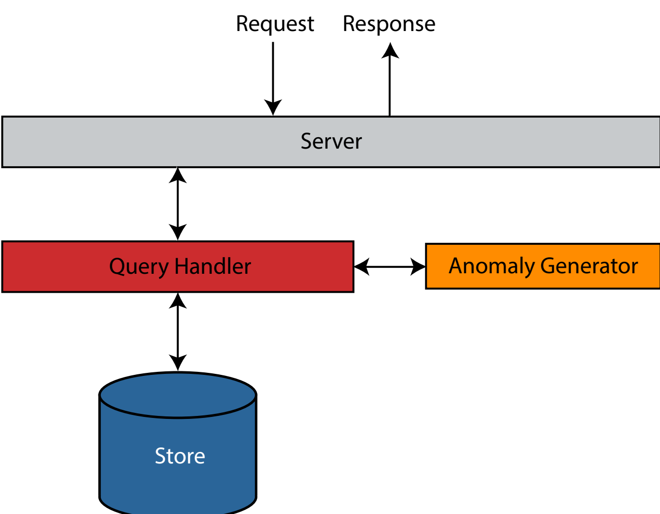

Figure 3.1.: Basic architecture of the SickStore server.

The core of the SickStore database system is the server application. It basically receives requests from clients and passes them over to the query handler which is responsible for its processing. The query handler parses the request and fetches or stores the data from and into the data storage. Furthermore, it applies anomalies generated by an *anomaly generator* and constructs the response object. Afterwards, the response is sent back by the server to the client. An illustration of the server's structure can be found in Figure 3.1.

As SickStore simulates a distributed database system, it consists of multiple virtual nodes that run inside a single process. It is possible to open a connection to a specific node[1] or to the server in general. If no *destination node* is specified, the server automatically selects the primary. As the query handler is aware of all available nodes, it is able to handle requests to specific nodes

[1] In the context of SickStore, the term "node" always means a virtual node.

differently and to simulate anomalies only for some nodes. The query handler uses only a single data store that contains all data items and is shared by all nodes. Consequently, there can be no inconsistencies between the nodes, unless not explicitly simulated.

3.1.1. Virtual Nodes

From a client perspective, a virtual node of SickStore behaves similar to a node from a distributed database system. A client can open a connection to a specific node to read and write data. Nevertheless, a node in SickStore is actually just an object that represents the distributed node and has no further functionality. It only contains the node's metadata that is used by the query handler and anomaly generator to create a sophisticated simulation. Currently, it consists of the following three attributes:

- **Name**: The name that a client can use to address a specific node.
- **Primary**: Defines whether the node acts as the primary that will be selected if no explicit node was defined in the request.
- **Tags**: A set of tags that are relevant for replication with tag-based write concerns (that will be introduced in Chapter 4).

This node object is further relevant to identify the owner of a data item, i.e. the node that received the write operation. This is relevant for the staleness calculations which will be explained in Section 3.1.4.

3.1.2. Data Storage

SickStore's data storage is a single component that is accessed by the query handler. It contains all data items of all nodes of all time. Internally, a data item is called a *version*. A version basically consists of the following elements:

- **Key**: The key of the data item.
- **Owner**: The node that received the write operation.
- **Timestamp of the write**: This timestamp is necessary to determine whether a node is allowed to see this version yet.
- **Staleness Map**: The staleness map contains the staleness values for all available nodes.
- **Data values**: The key-value pairs of the version.

The store saves all versions in a hash map that maps the item's key to a version set. A version set contains all data items of all time that have the same key. Therefore, the server can select a version that is currently visible[2] on the requested node for the connected client. By doing this, staleness simulations become possible.

[2] The term "visible" means that a version can be read with a read request.

3.1.3. Replication

SickStore allows write operations on any virtual node. Therefore, the replication model of SickStore is basically an update anywhere approach. However, by writing only to a specific node it is possible to simulate primary copy replication. If no destination node is defined the primary will be selected automatically.

As SickStore uses only a single data storage for all nodes, data updates become immediately visible on other nodes. However, by simulating staleness, data updates can only be observed on another node after a specific amount of time. Therefore, the propagation strategy of SickStore is lazy replication, as inconsistencies and stale data between the nodes can arise.

3.1.4. Anomalies

The anomaly generator is responsible for the generation of anomalies on stored data and incoming requests. The generated anomalies will be applied by the query handler. Currently, two types of anomaly generators are supported:

- The **staleness generator** calculates a staleness map that define at which time written data becomes visible on other nodes, i.e. that the data can be observed with a read operation. This allows the behavior of distributed database systems to be simulated, in which data updates need some time until they are propagated to all replicas because of the communication delay between the nodes.
- The **client delay generator** calculates an artificial delay (in milliseconds) that would be produced by performing the request. Thus, this is the time the client has to wait for an answer from the server.

This work extends the anomaly generator with a client delay generator that simulates latencies that are caused by a MongoDB-like replication and a staleness generator that considers the communication delays between nodes in a MongoDB-cluster. (compare Chapter 4).

3.2. SickStore Client

The SickStore client is the front-end library that allows communication with the server. The client can modify the stored data by inserting, updating and deleting items. Furthermore, saved data can be queried and it is possible to scan items within a specific key range.

As there is only one server application, the client always connects to the same server. If necessary, a destination node that processes all sent requests can be configured by an additional parameter for the whole connection.

SickStore is able to simulate communication delays and write latencies. However, as the server runs in a single thread to ensure serializability, it is not able to let the client wait. Consequently, the calculated delay will only be returned to the client which in return waits for the corresponding time.

3.3. Technologies

SickStore is implemented in the *Java* programming language and uses the *KyroNet*[3] library for the network communication between the client and the server. During the runtime of the application it collects metrics with the *Dropwizard Metrics*[4] library that can be compared with the benchmark results.

This chapter investigated the architecture and components of SickStore. With this knowledge, a simulation of a MongoDB-like replication behavior will be implemented in the next chapter.

[3] `https://github.com/EsotericSoftware/kryonet`
[4] `https://dropwizard.github.io/metrics/3.1.0/`

4. Simulating Replication with SickStore

One goal of this work was to simulate the characteristics of a replicated database system with SickStore. To do so, SickStore was extended with a replication component that is inspired by MongoDB's replica sets. In order to explain the implementation, this chapter examines MongoDB's replication system in different steps. At first, the main functionality of MongoDB's replication will be explained in Section 4.1. Then, an estimation of the occurring write latencies will be given in Section 4.2. After describing the test setup in Section 4.3, that was used to validate the estimation, the estimation will be validated in Section 4.4. Additionally, the impact on data staleness will be described in Section 4.5. Afterwards, the implementation into SickStore will be described in Section 4.6. In the end, the results will be evaluated in Section 4.7.

4.1. Replication in MongoDB

MongoDB[1,2] is a document-oriented NoSQL database system and was the most popular NoSQL database system in 2014 [AG15]. Replication in MongoDB is offered through so called *replica sets* that are based on master-slave replication (compare Section 2.5). Replica sets extend the default master-slave replication with an automatic failover that specifies a new primary if the current one becomes unavailable.

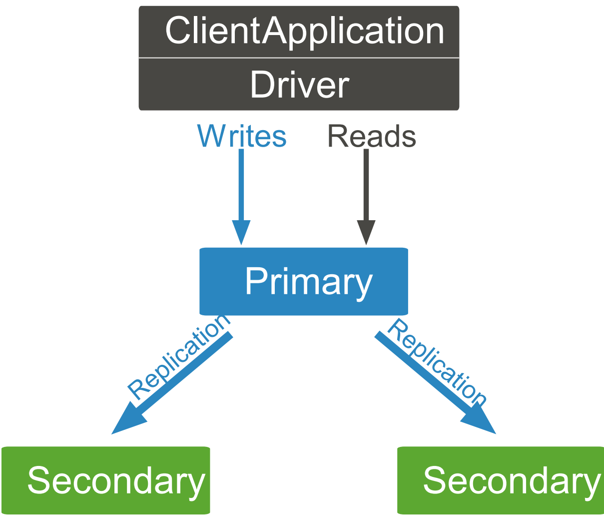

Figure 4.1.: Replication with MongoDB as illustrated in [Mon15]

A replica set consists of a single primary and multiple secondary[3] nodes. A client is only allowed to perform write operations on the primary. The operation is then propagated asynchronously

[1] This work focuses on Version 3.0.3.
[2] In the following, all information regarding MongoDB are based on [Mon15].
[3] MongoDB supports replica sets with up to 50 secondary nodes. If more secondaries are needed, replica sets cannot be used and normal master-slave replication has to be used.

to the secondaries (see Figure 4.1). By default it is further only allowed to read data from the primary. However, it is possible to configure MongoDB to allow data to be read from a secondary, too. This feature is disabled by default, as it becomes thereby possible to read stale data, since an update operation might not have been propagated to the secondary at the time of the read.

In case of a failure of the primary, MongoDB tries to elect a new primary out of the available[4] secondaries to get back to normal operations. This happens fully automatically and without manual intervention. During the election phase, the set has no primary and does therefore not accept any write operations in the meantime; though reading data is still possible. As the election of a new primary is not part of this work, we are not going to explore its details any further. More information on this topic can be found in the MongoDB documentation[5].

4.1.1. Write Concerns

According to [Mon15], *write concerns* in MongoDB describe a level of assurance about the durability of a data update when a write request is confirmed. The weakest write concern does not provide any guarantee at all, since it does not even wait for a response from the primary. Stronger write concerns require that write operations were replicated to a specific number of secondaries and were written into the journal. The write concern of an update operation has a direct impact on the write latency; write operations with higher write concerns need more time to complete.

A write concern consists basically of two parameters. One is the `w` parameter. It specifies the number of nodes or a tag set that must have acknowledged the write operation before it can be confirmed to the client. Its default value is *Acknowledged* ($w = 1$) which means, that only the primary needs to confirm the operation. Table 4.2 gives an overview of the available write concern levels in MongoDB.

The second parameter `j` specifies whether a write operation has to be written into the on-disk *journal*[7] of the primary before a request is acknowledged. Normally, a data update will be done only in-memory and eventually flushed to the on-disk database. By requiring a journal commit, MongoDB guarantees durability.

Additionally, a write concern can be refined by a third parameter `wtimeout` which defines a timeout in milliseconds after which the operation is aborted and an error is returned to the user. This can happen if some secondaries are unavailable or take too much time to process the operation. Nevertheless, MongoDB does not undo successful data modifications.

[4] Actually, only a subset of the available secondaries – the voting members – can be elected to become primary. MongoDB allows at most 7 voting members in a single replica set.

[5] `http://docs.mongodb.org/manual/core/replica-set-elections/`

[7] The journal in MongoDB is the transaction log.

Category	`w`	Description
Unacknowledged	0	A write operation with this write concern will not be acknowledged by the cluster at all. After submitting the write operation, the application will continue its work without waiting for the result. Only network and socket errors that are directly detected will be passed to the client.
Acknowledged *(default)*	1	Only the accessed node (the primary) has to acknowledge the request. This is the default write concern in MongoDB.
Replica Acknowledged	"majority"	A write operation will only be acknowledged after a majority of all voting members[6] in the replica set confirmed the operation.
Replica Acknowledged	$n > 1$	Only after n members of the replica set have acknowledged the write operation, the client will be informed. If n is greater than the number of registered nodes, the operation will fail. If less than the n nodes are reachable, the write will never be acknowledged at all (unless a timeout is specified, but in this case an error will be returned).
Replica Acknowledged	a tag set	Nodes can be grouped by tags (e.g. different data centers). By defining a tag set on the write concern it is possible to require acknowledgments from a certain number of nodes with specific tags.

Table 4.2.: Write Concern Levels in MongoDB [Mon15].

4.2. Estimation of MongoDB's Write Latencies with Different Write Concerns

To simulate the behavior of MongoDB's replica sets, the time needed for the replication of a write operation has to be estimated initially. The estimation is necessary as the MongoDB documentation does not give any information on how a specific write concern might influence the write latency. In fact, the documentation describes almost only how to *use* write concerns. The only attempt to describe the replication process with write concerns is given by the sequence diagram in Figure 4.2. The diagram indicates that a write operation is propagated to each secondary in sequence and not concurrently, nevertheless this would be very inefficient. However, as we will see below, the operation is in fact disseminated concurrently to all replicas. For that reason, an estimation of the write latency for specific write concerns will be given in the following.

To give a precise estimation, it is important to define at first what exactly should be estimated and which parameters need to be considered. We are interested in the behavior and delays produced by MongoDB's write concerns. As stated above, a write concern consists of the number of acknowledgments and whether the operation should be committed into the journal. Both need to be examined individually and in combination.

To estimate the effects of different acknowledgments the *observable replication latency*[8] will be

[8] *Observable replication latency* means the amount of time that is observable by the client in the write delay.

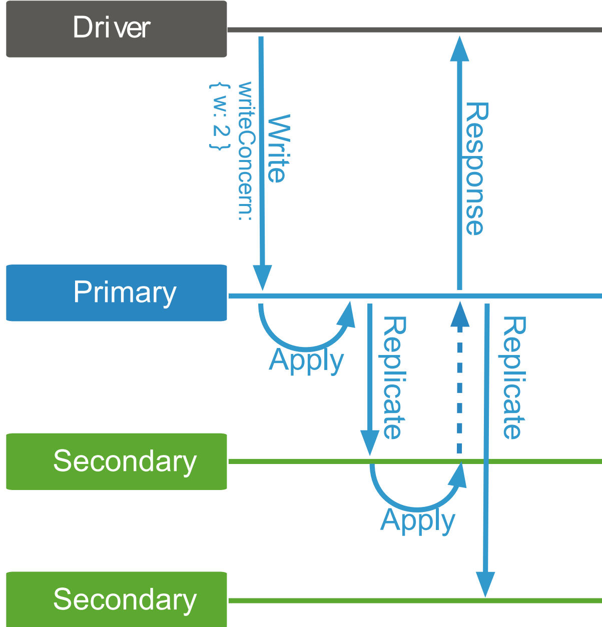

Figure 4.2.: Write operation with two acknowledgments. [Mon15]

taken into account. It describes how much time it takes to replicate a write operation to the appropriate number of replica until the request can be confirmed. The estimation of the observable replication latency should only consider the duration from the time the write request was received at the primary until the write concern is fulfilled and the request can be acknowledged. The communication delay between the client and the primary will be ignored; it will be assumed that the request was triggered directly on the primary. Besides, the I/O[9] time that is needed to handle an incoming write operation at a replica is ignored, as the I/O time can be expressed as part of the communication delay. By doing so, only the communication time between the primary and the replica will be considered and any other parameter ignored.

A similar estimation is made, to estimate the latency that is produced by requiring a journal commit. As MongoDB commits the journal periodically, only the commit interval and the time until the next commit are relevant. This write concern does not have any effect on the secondaries and the replication latency, as only the primary is required to commit the operation into the journal. Additionally, the I/O time is ignored again.

At first, we are going to estimate the latencies of write concerns that only specify the number of acknowledgments. Afterwards, the effects of a journaled write concern on the write latency will be examined. In the end, both concepts will be combined and the overall write latency determined.

[9] Input / Output

4.2.1. Unacknowledged and Acknowledged

In case of the write concerns *Unacknowledged* ($w = 0$) and *Acknowledged* ($w = 1$), the estimation is simple. The unacknowledged write concern does not cause any delay, as the write request is finished directly after it was sent to the server without waiting for a response. The acknowledged write concern behaves similarly: only the primary, that received the write operation, has to confirm it. Consequently, there will be no delay observable by the client which is caused by the propagation to the replicas; the primary acknowledges the request directly after processing it. All in all, there is no replication latency observable in the write delay for these two write concerns and the operation lasts 0 ms[10].

4.2.2. Replica Acknowledged with a Specific Number of Nodes

The estimation of the observable replication latency of a replica acknowledged write concern is more complex. To give an estimation, we imagine a replica set with one primary node P and n secondary nodes R_i $(1 \leq i \leq n)$ as illustrated in Figure 4.3. Propagating a write operation from the primary to replica R_i takes a time of d_i until the confirmation is received at the primary.

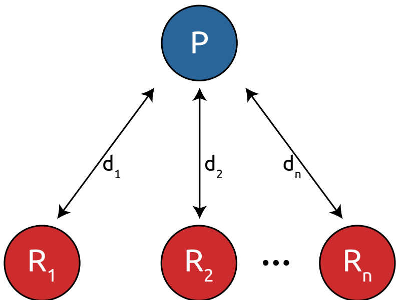

Figure 4.3.: Replica set with n secondaries

We assume, that a write operation is propagated concurrently to all replica, as this would be the most efficient way. For that reason, the operation can be confirmed after as many replica acknowledged the operation as required by the write concern. Formally, the observable replication latency l_r for a write concern with w acknowledgments is described in (4.1).

$$\begin{aligned} e &\mathrel{\hat{=}} \text{sorted tuple of all delays } d_i \quad (e_i \leq e_j,\ 1 \leq i < j \leq n) \\ l_r &= max(e_1, e_2, ..., e_{w-1}) \end{aligned} \tag{4.1}$$

In other words, the operation lasts until $(w-1)^{th}$ replica confirmed the operation and takes exactly that much time. Table 4.3 gives an example with four nodes. In Table 4.3a, the communication delays between the primary and each replica are given. The observable replication latencies for the specific acknowledgments are given in Table 4.3b.

[10] milliseconds

Replica	Delay d_i
1	10 ms
2	20 ms
3	5 ms
4	10 ms

(a) Communication times from the primary to each replica R_n

Write concern w	Replication latency l_r
0	0 ms
1	0 ms
2	5 ms
3	10 ms
4	10 ms
5	20 ms

(b) Possible acknowledgments with their corresponding replication latency.

Table 4.3.: An example of the replication latency calculation.

4.2.3. Replica Acknowledged with a Tag Set

MongoDB allows different nodes to be clustered by tags. Those can be defined by adding a set of tags to the node's replication configuration. It is possible to add tags to any node, including the primary. *Tag sets* can be used in write concerns, to define that a specific number of nodes with certain tags need to acknowledge a write operation, before it can be confirmed to the client. For instance, a write operation can be required to be acknowledged by at least two nodes in all available data centers, before it can be confirmed to the client.

To give an estimation, we imagine multiple tags $T_s \subseteq R \cup \{P\}$), that are subsets of all available nodes. A write concern with a tag set consists of multiple sub-concerns that define a replica acknowledgment for all available tags: $W(T_s) = k$ $(0 \leq k \leq |T_s|)$. In other words, only after at least k nodes with tag T_s confirmed the operation, the sub-concern is fulfilled.

The sub-concerns $W(T_s)$ are expected to behave in the same way as the replica acknowledgment for a specific number of nodes, which was estimated above (compare Section 4.2.2). Therefore, the operation lasts until all sub-concerns for all tags are fulfilled. Equation (4.2) gives a formal estimation of the observable replication latency l_r in this case. As the primary can also be a member of a tag set, its delay must be defined first for the estimation. Obviously, as the operation is executed on the primary, its delay is $d_P = 0$.

$$\begin{aligned} e(T_s) &\hat{=} \text{sorted tuple of all delays } d_i \text{ of the nodes in } T_S \quad (e_i \leq e_j,\ 1 \leq i < j \leq n) \\ l(T_s) &= max(e_1(T_s), e_2(T_s), ..., e_{W(T_s)}(T_s)) \\ l_r &= max(\{l(T_s) \mid \text{for all available } T_s\}) \end{aligned} \tag{4.2}$$

In the following, an example will be given. We consider the same nodes and communication delays given in Table 4.3a. Table 4.4 lists the nodes with their tags. A write concern which needs one confirmation of nodes with the tag *A* and two confirmations of nodes with the tag *B* is expected to have an observable replication latency of 10 ms.

Node	Tags
P	A
R_1	A, B
R_2	B
R_3	B
R_4	- (no tag)

Table 4.4.: List of nodes with their tags.

4.2.4. Journal Commit

MongoDB's journal is flushed periodically between a configurable interval of 2 and 300 ms to disk. Thus, writing something that needs to be committed into the journal requires waiting for the next commit iteration. To improve the insert performance in case an operation is waiting for a journal flush, MongoDB reduces the commit interval to a third of the set value. Figure 4.4 illustrates this behavior with a configured commit interval of 300 ms. At $t = 750$, a write operation that requires a journal commit is performed. The commit interval is reduced to 100 ms, so that the next journal commit occurs at at $t = 800$. Afterwards, the journal is committed at an interval of 300 ms again.

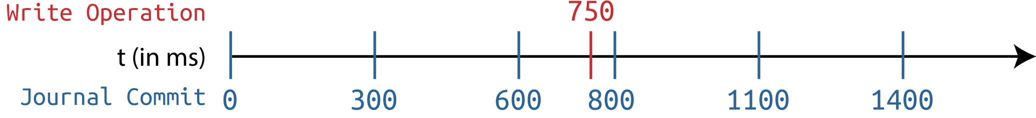

Figure 4.4.: Illustration of journal commit intervals.

We estimate the impact on the write latency by using the following model: the journal is flushed periodically at an interval of I_f with a value between 2 and 300 ms. A write operation is received by the server at t_w $(0 \leq t_w < I_f)$ after the last journal commit. Then, MongoDB reduces the commit interval to $\frac{I_f}{3}$. As the write operation has to wait for the next journal commit, a write latency of l_j as described in (4.3) is expected.

$$l_j = \frac{I_f}{3} - \left(t_w \bmod \frac{I_f}{3}\right) \tag{4.3}$$

Example: The journal is flushed every $t_I = 150$ ms. A write request arrives 80 ms after the last journal commit and must therefore wait 20 ms for the next flush.

4.2.5. Journal Commit and Replica Acknowledgments

As it is a usual scenario to require a specific number of acknowledgments (or a tag set) in combination with a journal commit, above estimations can be combined into an overall estimation. In this case, it is expected that the journal commit happens concurrently to the replica propagation. Therefore, the overall write latency l_w takes as much time as the longest one of both operations.

Formally, l_w can be defined as described in (4.4).

$$l_w = max(l_j, l_r); \tag{4.4}$$

4.3. Test Setup for Validation

MongoDB does not provide any tools to measure those latencies. Therefore, we have to build an environment which is able to produce delays that can be observed in the overall write latency. To do that, a MongoDB cluster with several nodes will be simulated that runs on a single computer. This allows the addition of constant latencies to the connections between the nodes. At first, the basic setup of the cluster will be explained. Subsequently, the simulation of replication latencies will be described.

The cluster will be built with the help of *Docker*[11], a tool that provides lightweight-virtualization on Linux operation systems and runs applications in isolated *containers*. Docker containers are started from pre-built images, that are based on instructions in the so called `Dockerfile`. For this simulation, the official MongoDB image[12] is sufficient and will be used.

To allow communication between containers, Docker creates its own virtual network. On the one hand, it creates a network bridge named `docker0` which is used to route packages between the containers, the host computer and remote networks. On the other hand, Docker creates a virtual network interface for every container on the host computer. Those interfaces are connected to the `docker0` bridge and receive their own IP[13] address [Doc15]. Consequently, each container can be reached by an individual IP address.

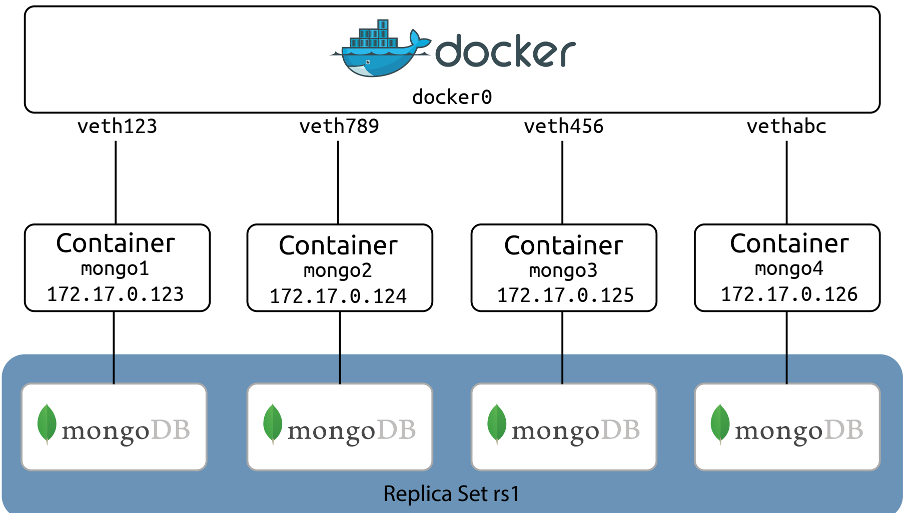

Figure 4.5.: Test setup with four nodes to validate MongoDB's latencies.

[11] `https://www.docker.com/`, Version 1.6.2
[12] `https://registry.hub.docker.com/_/mongo/`
[13] Internet protocol

In the following, a setup as illustrated in Figure 4.5 is built. It consists of four Docker containers, each running a single MongoDB instance. Those MongoDB instances will be connected to a single replica set `rs1`.

A new Docker container can be created and started with the `docker run` command (Listing 4.1). The first parameter specifies a name that is given to the container[14]. The name of the first instance can be, for example, `mongo1`. The `-d` switch defines that Docker should run the container in the background. The following argument (`mongo`) is the name of the base image that is used to start the container. If the image is not available on the local computer yet, it will be downloaded automatically. The following arguments are directly passed to the MongoDB binary. As we want to validate the replication behavior, we create a new replica set with the name `rs1` with the `--replSet rs1` argument.

```
docker run --name $name -d mongo --replSet rs1
```

</> Listing 4.1: Command to start a new Docker container.

By executing this command several times, Docker creates the appropriate number of MongoDB instances. Now, it is possible to open a connection to one MongoDB instance with the command from Listing 4.2 by providing the instance's name. This starts the client application directly in the corresponding container, which opens a local connection to MongoDB. By doing this, the client will not be distorted by later added delays to the container's network interface. The `docker exec` command can further be used to run any application inside a container, not just the Mongo-client.

```
docker exec -t -i $name /usr/bin/mongo
```

</> Listing 4.2: Command to start the Mongo-client on the primary.

To initiate the replica set, the IP addresses of all containers have to be obtained first. This can be done with the command from Listing 4.3. Afterwards, we can use the IP addresses to initiate the replica set with a command similar to Listing 4.4[15,16]. Please note, that all hosts are passed directly as a parameter to the `rs.initiate()` function and not added via `rs.add()`. This is done because `rs.add()` would try to resolve the IP addresses to host names, which have not been defined here. Furthermore, using host names would require an extra DNS[17] server to resolve those names. Therefore, and to keep the setup simple, host names were not used here. Finally, the cluster is ready to be used. All in all, the previous steps can be automated by using the Shell given in Listing A.1 (page 53).

[14] Specifying a name for the container is not necessary, as Docker would assign a random name otherwise. However, it is simpler to reference a container with a specific name.
[15] MongoDB uses JavaScript as scripting language.
[16] In this example, some random IP addresses were used which have to be replaced by the actual ones.
[17] Domain Name System

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' $name
```

</> Listing 4.3: Command to retrieve an IP address of a Docker container.

```
rs.initiate({
    _id: "rs1",
    members: [
        { _id: 1, host: "172.17.0.123:27017" },
        { _id: 1, host: "172.17.0.124:27017" },
        { _id: 1, host: "172.17.0.125:27017" },
        { _id: 1, host: "172.17.0.126:27017" },
    ]
});
```

</> Listing 4.4: Command to start the replica set and to register its members.

4.3.1. Simulation of Network Delays

To validate the estimated replication latencies, the actual latency has to be measured and compared with the expected one. However, as there is no real network connection between the MongoDB instances that produce latencies, such delays have to be generated artificially. This can be achieved with the *netem* tool that is part of the Linux kernel and is present in most current Linux distributions. It provides functionality to emulate networking delays, package loss, duplication and re-ordering. However, in the context of this work, only the delay emulation is of interest [Fou09].

Netem adds its delays per default to all outgoing packages. Therefore, the time for sending a package until an answer is received is a combination of the delays of both nodes (and the normal communication time). This happens because the answer is also an outgoing package on the receiver and will be delayed. Figure 4.6 portrays this behavior: a data package is sent from one node to another and an answer is sent back. The data package is delayed at the sender and the answer is delayed at the receiver, which is illustrated by the dashed line with the hourglass. Thus, the overall artificial delay is $d_1 + d_2$.

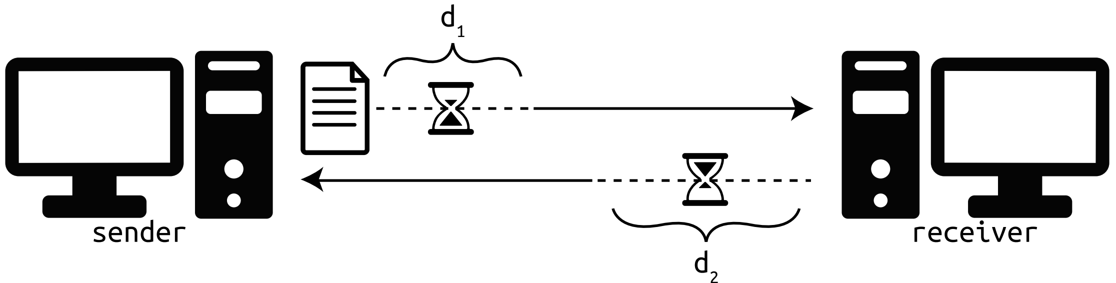

Figure 4.6.: Illustration of netem delays

Netem delays can be set by executing the command from Listing 4.5 as root user. The variable `$interface` must be replaced with the name of the network interface to which the delay should

be added. Moreover, `$delay` must be replaced with the appropriate delay, for example `100ms`.

```
1 tc qdisc replace dev $interface root netem delay $delay
```

</> Listing 4.5: Setting a delay on all outgoing packages of a specific network interface.

In case of this work, the delays are added to the container's virtual network interfaces. As Docker does not provide the container's interface name directly, it must be obtained manually at first. This can be done by mapping the container's internal network interface id to the virtual interface's id on the host computer. The ids can be obtained by running the `ip link` command, which lists the available network interfaces, on the host computer and in the Docker containers with `docker exec -t -i $name ip link`. The output on the host computer will look similar to the example in Listing 4.6; the output of the command inside a Docker container will look similar to Listing 4.7. The number in the beginning indicates the network interface's id. Subsequently, the name of the interface is given. The other information is negligible for this work and will not be explained any further.

To determine the interface name on the host computer, we have to take the id of the internal `eth0` interface (here: 5) and look for the interface with the same id plus one (here: 6), as both interfaces are created sequentially. In this case, the corresponding interface name of `mongo1` is `vethcef5006`. The shell script in Listing A.2 (page 54) can be used to automate this process; it displays all container names with their corresponding interface names.

```
1  1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
   ↪  group default
2      link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3  4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
   ↪  DEFAULT group default
4      link/ether 16:8f:7d:2c:99:ad brd ff:ff:ff:ff:ff:ff
5  6: vethcef5006: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state
   ↪  UP mode DEFAULT group default
6      link/ether 16:8f:7d:2c:99:ad brd ff:ff:ff:ff:ff:ff link-netnsid 0
7  8: veth574bfb2: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state
   ↪  UP mode DEFAULT group default
8      link/ether 1a:4a:19:88:15:ea brd ff:ff:ff:ff:ff:ff link-netnsid 1
9  10: veth85c9d25: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state
   ↪  UP mode DEFAULT group default
10     link/ether 8a:6a:a2:7c:e4:52 brd ff:ff:ff:ff:ff:ff link-netnsid 2
11 12: veth080daae: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state
   ↪  UP mode DEFAULT group default
12     link/ether 62:29:3c:9b:b5:04 brd ff:ff:ff:ff:ff:ff link-netnsid 3
```

</> Listing 4.6: Output of `ip link` on the host machine with four running containers.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
5: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT
    link/ether 02:42:ac:11:00:01 brd ff:ff:ff:ff:ff:ff
```

</> Listing 4.7: Output of `ip link` inside a Docker container.

4.4. Validation of Write Latencies

As the MongoDB documentation does not give any information about the effects of write concerns on the write performance, an estimation thereof was given in Section 4.2. To be certain, that this estimation is reasonable it will be validated in the following. Similar to the estimation, the replication latency will be validated at first and the journal commit latency subsequently. In the end, the combination of both will be validated.

4.4.1. Replica Acknowledged with a Specific Number of Nodes

With the help of Docker and netem, it is now simple to validate the replication latencies. To validate them, a MongoDB cluster with four nodes (as illustrated above in Figure 4.5) will be started. `Mongo1` will become the primary. The delays from Table 4.5 will be added with netem to the appropriate virtual network interfaces:

Node	`mongo1`	mongo2	`mongo3`	`mongo4`
Delay (ms)	100ms	500ms	1000ms	1000ms

Table 4.5.: Delays used to validate replication latencies.

After adding the delays to the network interfaces, the actual write latency can be measured by writing something into MongoDB and measuring the time needed until the operation is completed. This can be achieved by executing the code from Listing 4.8 as a single instruction[18], which needs to be executed with the Mongo-client running in the primary's container. The variable `$w` needs to be replaced with the desired level of acknowledgment.

```
var before = new Date();
print(db.sample.insert({"thesis": "Bachelorthesis"}, {writeConcern: { w: $w }}));
print("duration: " + new Date() - before);
```

</> Listing 4.8: Command to perform a write operation and to measure its time.

In Table 4.6, the expected and measured write latencies are given. As we can see, the measured latency differs only slightly from the expected ones. This can be reasoned by additional I/O of MongoDB which was ignored in the estimation. Moreover, there was actual network latency in addition to the artificial delays, even though those last less than a millisecond and occur only on

[18] To execute the code as a single instruction, it can be combined into one line or loaded from a external script with `load('file.js')`

the host computer. However, as long artificial delays were used to confirm the estimation, the measured latencies match the expectation.

Write concern w	Expected latency	Measured latency
0	0 ms	1 ms
1	0 ms	1 ms
2	600 ms	604 ms
3	1100 ms	1103 ms
4	1100 ms	1104 ms

Table 4.6.: Expected and measured replication latencies.

4.4.2. Replica Acknowledged with a Tag Set

The validation of the replication latencies with a tag set is similar to the validation with a specific number of nodes (see Section 4.4.1). The estimation for write concerns with tag sets can be and was confirmed, by configuring tag sets and repeating the previous steps. Therefore, the necessary steps will not be explained in detail. The documentation on how to configure tag sets can be found in the MongoDB documentation[19].

4.4.3. Journal Commit

Previously, it was assumed, that a write operation that needs to be flushed into the journal has to wait for the next commit. To validate this behavior, a new MongoDB instance will be started with a specific commit interval. Then, multiple write requests will be sent in a loop to the server. The time of each operation will be measured.

To keep it simple, MongoDB will be started in a Docker container like above, although this would not be necessary. For this validation, a single MongoDB instance is sufficient and no replica set needs to be initiated. Instead, the journal commit interval will be configured. The command in Listing 4.9 specifies, how to start a new instance with a journal commit interval of 300 ms. This time, a fix name (`mongo`) was used, as multiple instances are not necessary.

```
1 docker run --name mongo -d mongo --journalCommitInterval 300
```

</> Listing 4.9: Starting a MongoDB container with a specific journal commit interval.

After the instance is started, the commit interval can be tested with the script in Listing 4.10. As a commit interval of 300 ms is configured, the interval will be reduced to 100 ms when a journaling operation is waiting. The script executes ten write operations and outputs the needed time of each operation. Every second operation, the script sleeps for 40 milliseconds. This should reduce the latency of those write operations, since the write is triggered closer to the next commit.

Listing 4.11 shows the results of this validation. The first operation took 77 ms, which does not match the expectation. However, as the script was started at an arbitrary time, the duration

[19] http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets/

```
for (var i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        sleep(40)
    }

    var before = new Date();
    db.sample.insert({"i": i}, {writeConcern: { j: true }});
    var after = new Date();
    var diff = after - before;

    print("i=" + i + "\tdur=" + diff);
}
```

</> Listing 4.10: Script to validate the journal commit interval.

does not state anything in this case; this value can be ignored. The second operation took 59 ms which matches approximately the expected value, as we waited 40 ms before performing the write. The measured value is a bit lower than the expected time of 60 ms, which can be reasoned by additional I/O of the previous operation which delayed the response. Therefore, the second operation was not triggered directly (including the 40 ms sleep) after the previous journal commit. The third operation was executed directly afterwards without a previous sleep and took 102 ms. This matches roughly the expected value of 100 ms. As we can see, all further durations alternate more or less between 100 ms and 60 ms, which matches the expectation.

```
i=1     dur=77
i=2     dur=59
i=3     dur=102
i=4     dur=63
i=5     dur=102
i=6     dur=59
i=7     dur=102
i=8     dur=61
i=9     dur=104
i=10    dur=58
```

</> Listing 4.11: Results of the script from Listing 4.10.

In conclusion, it can be confirmed that the journal is committed periodically to disk and that a write operation has to wait for the next journal commit. Moreover, it can be confirmed that the commit interval is reduced to a third of the configured value if a write operation is waiting that should be committed.

4.4.4. Journal Commit and Replica Acknowledgments

The combination of requiring a journal commit and replica acknowledgments is expected to take at most that much time that the longest one of both operation needs. To confirm this behavior,

the previous steps of the journal validation will be repeated within a replica set and with different replication delays. At first, a replication delay will be used that is higher than the maximum possible journal commit latency and the journaling. By doing so, a write operation should need as much time as the observable replication delay and journaling should not be observable. Afterwards, the opposite effect will be validated: when the observable replication delay is lower than the journal commit delay, a write operation should take as long as the waiting time for the next journal commit and the replication will not be observable.

For this validation, the setup given in Section 4.3 will be used again, but with one change: each MongoDB instance will be started with a journal commit interval of 300 ms (with the `--journalCommitInterval` parameter).

The validation will be conducted with a write concern that requires acknowledgments from all four replias and a journal commit. To do this, a slightly modified version of the script in Listing 4.10 will be used, that can be found in Listing 4.12. The only difference is the modified write concern, which now requires a replica acknowledgment from four nodes.

```
for (var i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        sleep(40)
    }

    var before = new Date();
    db.sample.insert({"i": i}, {writeConcern: { w: 4, j: true }});
    var after = new Date();
    var diff = after - before;

    print("i=" + i + "\tdur=" + diff);
}
```

</> Listing 4.12: Script to validate the combination of journaling and replication.

Observable Replication Latency higher than Journaling Latency

For the first validation, a replication delay of 500 ms was generated by adding a delay of 250 ms to each virtual network interface with netem. It was expected, that a write operation will take about 500 ms to complete, as the journal commit interval endures at most 100 ms and is therefore absorbed by the replication delay. The results of this validation, that match the expectation, can be found in Listing 4.13.

Journaling Latency higher than Observable Replication latency

This time of the observable replication delay should be less than the commit interval, which is at most 100 ms long in case that a write operation is waiting for a commit. Therefore, a replication latency of 50 ms will be configured by adding a delay of 25 ms to each virtual network interface. Thus, a write operation endures at most 100 ms because of the waiting time for the journal

```
i=1     dur=504
i=2     dur=503
i=3     dur=504
i=4     dur=503
i=5     dur=504
i=6     dur=504
i=7     dur=505
i=8     dur=504
i=9     dur=504
i=10    dur=503
```

</> Listing 4.13: Latencies of a write operation with a higher replication than journaling delay.

commit. As the same script is used that was used to validate the commit interval, it also sleeps 40 ms before each second write operation. Due to that, the duration is expected to alternate between roughly 60 and 100 ms.

The results of this validation can be found in Listing 4.14. Basically, those results match the expectation. The result of the first operation can be ignored again, as the validation was started at an arbitrary time.

```
i=1     dur=53
i=2     dur=67
i=3     dur=99
i=4     dur=68
i=5     dur=100
i=6     dur=60
i=7     dur=108
i=8     dur=59
i=9     dur=100
i=10    dur=60
```

</> Listing 4.14: Latencies of a write operation with a higher replication than journaling delay.

4.5. Data Staleness

Basically, MongoDB recommends to perform all operations on the primary to ensure strong consistency. Additionally, there is no data staleness if operations are only performed on the primary. Nevertheless, it is also possible to perform read operations on a secondary.

By reading data from a secondary it becomes possible to read stale data, since an update operation might not have been propagated yet. By defining a strong write concern, MongoDB only ensures that with the acknowledgment of the write request the operation has been propagated to the specified number of nodes. This does not necessarily mean that the operation has been propagated to all nodes, unless a confirmation of all available nodes is required. Therefore, stale

data can be read from a secondary even if the operation was previously confirmed by the primary. Additionally, it is possible to read recent data from a secondary before the operation was confirmed by the primary.

Data staleness arises because of the communication delay from the primary to the secondaries. It is pretty clear, that the secondary cannot be aware of an update operation before it arrived. Therefore, a simulation of data staleness with SickStore needs to consider the communication time from the primary to the secondary. Only after that amount of time has passed, written data is allowed to be observed on a secondary.

For example, a data item is written at t_0. It takes 10 ms to propagate the operation to replica R_1 and 20 ms to propagate it to replica R_2. Then, the written data can only be observed on R_1 after $t_0 + 10$ and on R_2 after $t_0 + 20$. On the primary, the written data is directly available. Prior to the propagation, a read operation will either return the old data item or it will fail if the item did not exist before.

In this work will be assumed that MongoDB follows the described behavior and no additional validation will be performed. Additionally, I/O time that comes on top of the network communication time is ignored again.

4.6. Implementation into SickStore

The simulation of MongoDB's replication behavior is implemented in the `MongoDbAnomalies` class into SickStore. It is on the one hand a *client delay generator*, as the configured write concern affects only the write latency the client can observe. On the other hand, it is also a *staleness generator* that is able to produce a staleness map according to the configured delays. Nevertheless, it is also possible to use only the delay generator or the staleness generator, depending on the scenario that should be simulated.

In the following, the implementations of various parts of the class will be described.

4.6.1. Write Concerns

According to Section 4.1.1, a write concern describes the level of assurance about the durability of a write operation. It is specified by the client and sent as part of the request to the server.

As each write concern has different implications on the performance, the client needs to be able to specify the write concern for a write operation. Consequently, a `WriteConcern` class was implemented into SickStore, that allows to specify the write concern as part of the request. SickStore's write concern accepts the same parameters as MongoDB[20].

[20] However, SickStore's implementation uses more descriptive names.

4.6.2. Anomaly Generator

In the following, the methods that are responsible for the particular delay and staleness calculations will be explained. This anomaly generator depends on the following parameters that have to be passed as constructor arguments:

- The **default delay** defines the communication delay between two nodes that have no custom delay configured.
- **Custom delays** define individual communication delays between two nodes.
- The **Journal commit interval** defines the interval in milliseconds at which the journal is committed.
- **Tag sets** define the required number of acknowledgments for a various tags and can be used in a write concern.

4.6.3. Client Delay Calculation

The entry point of the client delay generator, is the `calculateDelay()` method, which expects a request object and a set with all available nodes to be passed as parameters. Basically, the client delay generator is applied on any incoming request. However, as only write operations (insert, update and delete) are replicated, this delay affects only those operations and ignores read operations. Therefore, the following steps are only applied if the incoming request is a write request.

The `calculateWriteDelay()` method (given in Listing 4.15) is responsible for the actual delay calculation for write requests. It calculates the observable replication and journaling latencies and returns the greater one. In the following sections, the calculation of the replication latency (Section 4.6.4) and the journal commit latency (Section 4.6.5) will be explained in detail.

```
1 private long calculateWriteDelay(ClientWriteRequest request, Set<Node> nodes) {
2     long replicationDelay = calculateReplicationDelay(request, nodes);
3     long journalingDelay = calculateJournalingDelay(request);
4 
5     return Math.max(replicationDelay, journalingDelay);
6 }
```

</> Listing 4.15: Calculation of the write delay, based on the replication and journaling delay.

4.6.4. Calculation of the Observable Replication Delay

The observable replication delay calculation needs to be distinguished into replica acknowledgment with a number of nodes and with a tag set, as both have different requirements on the calculation. The basic delay calculation can be found in Listing 4.16. If a tag set acknowledgment is required, the `calculateTaggedReplicationDelay()` method calculates the delay. Otherwise, the calculation is made in place.

```java
private long calculateReplicationDelay(ClientWriteRequest request, Set<Node> nodes) {
    WriteConcern writeConcern = request.getWriteConcern();
    if (writeConcern.getReplicaAcknowledgementTagSet() != null) {
        return calculateTaggedReplicationDelay(request, nodes);
    }

    if ((nodes.size() == 1 || writeConcern.getReplicaAcknowledgement() <= 1)) {
        // if there is only one node or if no replica acknowledgment is required
        // the delay is zero
        return 0;
    }

    // look for custom delays
    TreeSet<Long> delays = findCustomDelays(request.getReceivedBy(), nodes);

    // the primary is subtracted (-1), as it has no delay
    return calculateObservableReplicationDelay(
        writeConcern.getReplicaAcknowledgement() - 1,
        delays
    );
}
```

</> Listing 4.16: Calculation of the replication delay.

Replica Acknowledgment with a Number of Nodes

The calculation of the replication delay with a number of nodes follows basically the estimation from Section 4.2.2. If there is only one node in the cluster or less than two acknowledgments are required, there is no replication delay observable at the client and `0` is returned.

Otherwise, the `findCustomDelays()` method (see Listing A.3 on page 55) selects the delays that occur from propagating the request to the various nodes. Similar to the simulated network delays for the validation (see Section 4.4), the overall delay is based on the communication time from the primary to the replica and the way back. Therefore, the overall delay is based on two configured custom delays or the default delay, if no custom delays are configured.

The selected delays are brought into an ascending order and passed into the `calculateObservableReplicationDelay()` method (see Listing A.4 on page 56). It looks for the lowest delay in the list that can fulfill the required replica acknowledgment (see example in Table 4.3).

Replica Acknowledgment with Tag Sets

The calculation of the replication delay with a tag set acknowledgment is a bit more complicated. The calculation is made in the `calculateTaggedReplicationDelay()` method (see Listing A.5 on page 57).

Basically, it selects the number of replica acknowledgments for each tag and calculates the observable delay for each tag, similar to the calculation above for a number of nodes. This time,

however, only those nodes are considered by the `findCustomDelaysWithTag()` method (see Listing A.6 on page 58) which have the required tag assigned. In the end, the largest delay of all tag acknowledgments is the expected observable replication delay and is returned.

4.6.5. Calculation of the Journaling Delay

The calculation of the journal commit delay is made in the `calculateJournalingDelay()` method, which can be found in Listing 4.17. If no journal commit is required by the write concern, a delay of `0` is returned.

If a journal commit is required, only one third of the configured commit interval is relevant for this calculation. The simulation can use the time since the server's startup for the calculation instead of the time of the last write, which was used in the estimation in Formula 4.3. This is allowed, as I/O and other factors that might defer commits were ignored. Therefore, and because of the modulo calculation, it has the same result. Apart from that, the delay calculation follows the estimation, which was made in Section 4.2.4.

```
1  private long calculateJournalingDelay(ClientWriteRequest request) {
2      if (!request.getWriteConcern().isJournaling()) {
3          return 0;
4      }
5
6      long timeSinceStartup = timeHandler.getCurrentTime() - startedAt;
7      long oneThird = journalCommitInterval / 3;
8
9      return oneThird - (timeSinceStartup % oneThird);
10 }
```

</> Listing 4.17: Calculation of the delay that emerges from a required journal commit.

4.6.6. Staleness Generator

A staleness map defines after which amount of time a written data item becomes visible for other nodes in the cluster. In the case of this simulation, it considers the delays from the primary to all replicas. If a custom delay is configured that one will be used, otherwise the default delay will be considered. The staleness value for the primary is always zero.

The staleness map is generated by the `generateStalenessMap()` method, which can be found in Listing A.7 (page 58). It needs the current request and a set of all available nodes to be passed as parameters. The method iterates over all available nodes and determines the appropriate staleness value. If the currently considered node is the receiving node (the primary), the staleness value is zero. Otherwise, it looks for a custom delay from the primary to the currently considered node. If no custom delay is found, the default delay is used. All staleness values are inserted into a map that in the end contains a staleness value for each node.

4.6.7. Validation with YCSB

As there is a YCSB client for SickStore, the YCSB client was adjusted to support write concerns. In the following, SickStore is benchmarked with YCSB. In this case, a cluster with four nodes will be simulated. The replication delays from the primary to the three replicas are given in Table 4.7. As no communication delays are given for the reverse direction, a default delay of 10 ms is used.

Replica	Delay
replica1	100 ms
replica2	200 ms
replica3	50 ms

Table 4.7.: Used delays for the validation with YCSB.

For this validation, the *Workload A* of YCSB is used which inserts 1000 records into the database in the *load* phase. Afterwards, in the *transaction* phase, 1000 operations are executed, that are evenly split into read and update operations. The commands in Listing 4.18 are used to execute the load and transaction phase.

```
1 ycsb load sickstore -P workloads/workloada -p sickstore.write_concern.ack=2 -p
  ↪ measurementtype=timeseries
2 ycsb run sickstore -P workloads/workloada -p sickstore.write_concern.ack=2 -p
  ↪ measurementtype=timeseries
```

</> Listing 4.18: Commands to execute YCSB benchmark.

With a replica acknowledgment of 2, an observable replication latency of 60 ms is generated, which should be roughly observable with YCSB. In fact, the measured results are expected to be a little bit higher, as SickStore still needs some additional time to process the request. The exact amount of time is difficult to predict, as it depends on the system and hardware. However, as only the replication delay is of interest here, the additional time is irrelevant here and will not be examined further.

After executing the benchmark, the measured average latency of an insert operation was 60.875 ms. In case of an update operation during the transaction phase, the average latency was 60.655 ms. However, as both delays are based on the same replication configuration, the latencies are very similar.

By using different write concerns, similar results occur that match the expected ones. However, as the same steps are necessary to validate other configurations, no further examples will be given.

4.7. Evaluation

In the previous sections, the behavior of MongoDB's replication system was examined and the estimation validated. In the following, the results will be evaluated.

This chapter considers mainly the write latency a client can observe. There are basically two parameters that influence the write latency directly and can be controlled by the client by defining a write concern: the replica acknowledgment and whether a journal commit is required. For that reason, the effects of both parameters on the latency were examined separately.

The estimation of the observable replication latency was basically validated, though the results matched the expectations only approximately; there were minor variances. These point out that there are other factors that influence the latency which were not considered yet. Some possibilities, for instance, are the time that is needed to process the request, the hardware performance and the number of concurrent requests. However, the effects of those parameters are difficult to measure and were therefore not considered in this work. All in all, it can be confirmed that the network delay is part of the entire write latency, even though there are more factors that influence the latency.

The impact of the journal commit interval onto the write latency was in principle validated, but with similar limitations like with the replication latency: there were also slight variations. Those variations can, similar to the replication behavior, be based on the hardware of the system or the number of concurrent requests. In addition, it takes some time until the journal was written to disk, which might also depend on the number of items. Overall, the commit interval was basically observable in the write latency, even though there are other factors that influence the latency and were not considered in the estimation.

As the estimations were based on assumptions and some parameters were not considered, the validations confirmed the estimations only approximately. However, the investigated parameters had a huge impact on the latencies and were responsible for a majority of the measured values to identify them clearly. Even if the replication latencies are not that long in a real-world system, they can have a huge impact in globally distributed systems and might make other factors negligible. Therefore, it was important to identity these parameters. The consequence of the journaling delay is similar. As the waiting time for the next commit can be quite long, other factors might be negligible. Nevertheless, it is also important to consider the missing parameters to make the simulation more sophisticated.

In addition, the staleness that can arise by reading data from a secondary was considered. The expected behavior was not examined into deep, as the main focus of this chapter was the write latency that is caused by different write concerns. Further research might validate the expected staleness. Nevertheless, the implemented staleness simulation can be used for benchmark validations, even though it might not exactly match MongoDB's behavior.

In further research, the so far neglected parameters can be considered. For instance, the behavior of MongoDB under load with many concurrent requests and the effects of a journal commit with lots of items can be examined. Furthermore, the effects of slow hardware and especially of a slow hard disk drive can be further investigated.

In conclusion, the simulation of replication with SickStore matches roughly MongoDB's behavior. However, only latencies that are caused by replication delays and the journal commit interval were considered, but no latencies that are caused by I/O and other factors. Nevertheless, it gives a sufficient foundation for future benchmarks and research. By continuing to examine the behavior of MongoDB, such as by investigating the source code, these simulations can be improved further.

As this chapter described the extension of SickStore with replication based on MongoDB's behavior, the next chapter will cover the principles of sharding in MongoDB and the simulation with SickStore.

5. Simulating Sharding with SickStore

Besides simulating replication, the second goal of this work is to add sharding into SickStore. The implementation is planned to be oriented on MongoDB's sharding functionality, which will be explored in Section 5.1. Afterwards, in Section 5.2, the implementation into SickStore will be described. Finally, the results will be evaluated in Section 5.3.

5.1. Sharding in MongoDB

According to [Mon15][1], a sharded MongoDB cluster consists of three basic components: *query routers*, *config servers* and *shards*. A **query router** is the interface for the client application and redirects requests to the corresponding shards. **Config servers** store the cluster's metadata that contains a mapping of the data sets to their shards. The **shards** contain the actual data and are *normal* MongoDB instances. Figure 5.1 illustrates the basic setup of a MongoDB cluster.

It is recommended (but not required) to have at least two query routers to have some failure tolerance and to distribute the load. With an increasing amount of requests the cluster has to handle, it may become necessary to add more query routers. Furthermore, it is required to have *exactly* three config servers to ensure redundancy and durability of the metadata in production[2]. Otherwise, the cluster will be inoperable in case of a failure. A shard can be a single MongoDB instance, but it is recommended to create a replica set for each shard to provide high availability (see Section 4.1).

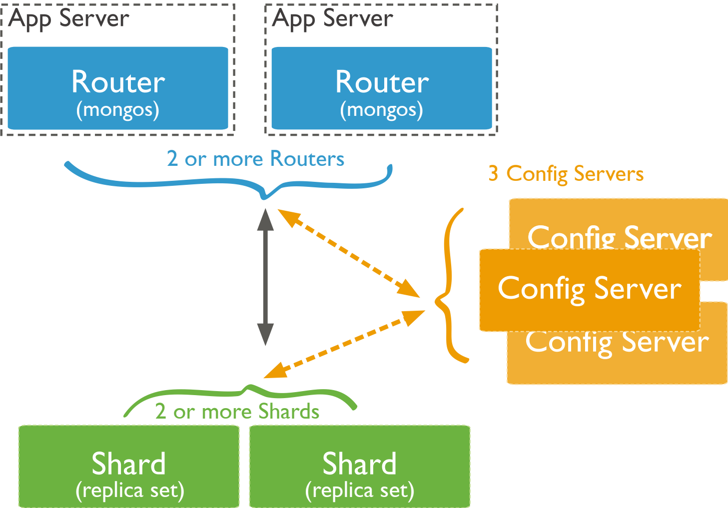

Figure 5.1.: Basic architecture of a sharded MongoDB cluster [Mon15].

The shard's data is further separated into *chunks* that are small, non-overlapping ranges of data items. A chunk can hold per default up to 64 megabyte of data, if not configured differently.

[1] In the following, all information regarding MongoDB are based on its documentation [Mon15].
[2] For developing and testing purposes it is allowed to have only one config server.

When a chunk grows, it can be split in half. This can happen at any time, but at the latest when it grows beyond the configured size. If there is an uneven amount of chunks on a shard in comparison to the other shards, MongoDB will re-balance the shards in order to have a similar number of chunks on each shard.

MongoDB shards data on the collection[3] level. Data items are assigned to chunks according to their *shard key*, an indexed field that must exist in any document (if it does not exist, MongoDB denies the insertion of the document). Additionally, the sharding strategy and the chunk's ranges are relevant to determine the target shard. MongoDB offers three different strategies to decide how a data item is assigned to a shard. Those will be explored in the following.

5.1.1. Range-based sharding

Ranged-based sharding is the default sharding strategy of MongoDB, that is automatically used if no other strategy is configured. MongoDB detects the chunk's ranges automatically from the inserted data and tries to distribute the data evenly in order to have a similar amount of data on each shard. In case that some chunks (or ranges) contain too much data, the data is re-balanced.

Figure 5.2 from the MongoDB documentation illustrates range-based sharding. The entire key range, which is stretched from a minimum key to a maximum key, is divided into chunks. The ranges do not have to be equal sized.

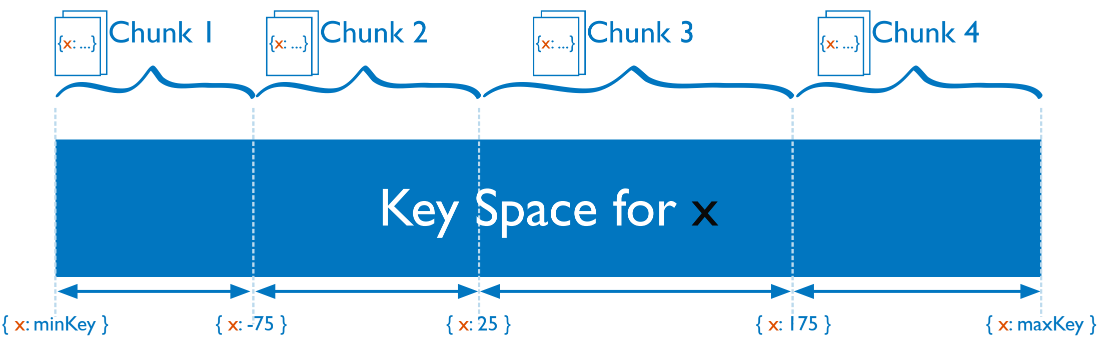

Figure 5.2.: Ranged-based sharding in MongoDB [Mon15].

5.1.2. Tag-aware sharding

The second sharding strategy of MongoDB is tag-aware sharding. It allows the mapping of data items to shards by a mapping of specific values to a tag. A shard can have multiple tags assigned and a tag can be assigned to multiple shards. This allows a manual assignment of data items to shards and it is even possible to assign data items to more than one shard.

To configure tag-aware sharding, the desired tags have to be assigned to the shards at first. Additionally, a range of values must be assigned to a specific tag.

[3] Collections in MongoDB are comparable to tables in relational database systems and contain data items of the same type, e.g. users. A single database can contain several collections.

5.1.3. Hash-based sharding

MongoDB's third sharding strategy is hash-based sharding. Hereby, a hash will be calculated from the shard key which will be used to determine the target chunk. Figure 5.3 illustrates the behavior. Three documents are inserted into the database, which are distributed into the chunks according to the calculated hash. In case of hash-based sharding, MongoDB allows only a single field as the shard key.

MongoDB uses the *Message-Digest Algorithm 5 (MD5)* hash function to determine an *MD5-hash* from the shard key. The resulting hash has a length of 128 bit, but MongoDB selects only the first 64 bit and converts it into a value of the type *long*. The resulting long is the actual value, that MongoDB uses to determine the target chunk, which is determined in a similar manner to range-based sharding. However, in the case of hash-based sharding the key space is split into equal-sized ranges.

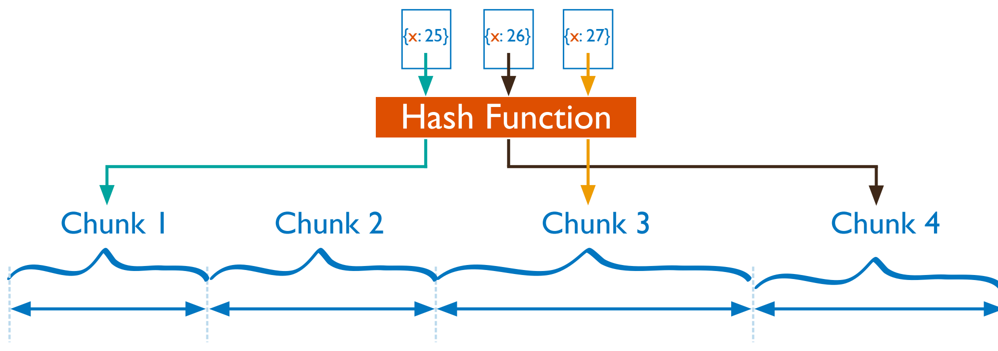

Figure 5.3.: Hash-based sharding in MongoDB [Mon15].

5.2. Implementation into SickStore

5.2.1. Considerations on MongoDB's Sharding

The actual implementation of sharding into SickStore is different from MongoDB's behavior, even though it was planned to resemble it. However, during the exploration of MongoDB's sharding strategy, it became pretty clear, that this behavior is not directly transferable to SickStore.

MongoDB accomplishes its sharding primarily on values in the data item. This is possible, as MongoDB has a complex query engine and index mechanism to store and query data objects. SickStore does not have something like this; it is just a key-value store. The only access to a data item is based on the item's key and it is not planned to soften this behavior.

For that reason, tag-aware sharding cannot be implemented with SickStore. It would require a query engine that is able to query for specific values on a data item, but SickStore supports only operations based on the key. Furthermore, there is no point in applying this behavior to keys, as those have to be unique. Assigning a tag for specific keys would correspond to a manual assignment of all data items to their shard. With range-based tags, this strategy would correspond

to range-based sharding, which is considered below. Therefore, this strategy is ineligible for the implementation.

The simulation of MongoDB's range-based sharding with SickStore is problematic, too. The internals and behavior of MongoDB are not very well documented to simulate an accurate behavior. Basically, the ranges are adapted during the insertion of documents. At startup, there are no chunks and a first chunk will be created with the first insertion of a new item. By inserting more items, the ranges are adapted and the data is re-balanced. However, the MongoDB documentation does not give very much information on the behavior of re-balancing and splitting of chunks. A proper simulation would only be possible with a deep understanding of the internals of MongoDB, which for example can be gained by examining the source code. Even the implication on the performance, which would be of interest in the context of benchmark tests, are very unclear. Therefore, this strategy is also ineligible to be implemented, as it would lead to a completely different behavior.

MongoDB handles hash-based sharding similar to range-based sharding. Even though a hash is calculated from the shard key, it is assigned to a shard in a range-based manner. It also depends on chunks with splitting and re-balancing processes. Therefore, the actual implementation cannot be inspired by MongoDB's hash-based sharding, too.

For those reasons, the actual implementation of sharding into SickStore will differ from MongoDB's and is only an elementary implementation of sharding.

5.2.2. Sharding with SickStore

Because of the problems discussed in the previous section, the actual implementation of sharding into SickStore does not consider MongoDB's behavior. It is only a rudimentary implementation that distributes the data and requests across the available shards, without chunks, re-balancing and splitting processes. A simulation of those features would behave completely different and was therefore not done.

SickStore runs per definition as a single application on a single server. This basic principle still applies after sharding was implemented. Therefore, it is implemented as part of the server application and operates transparently for the client.

The sharding functionality is implemented as a new layer between the server and the query handler. The new *sharding router* is responsible for the redirection of requests to the proper shards, similar to MongoDB's router. Figure 5.4 illustrates the new architecture. Each shard is an instance of the query handler, as it has the same functionality. This is similar to MongoDB's architecture, where a shard is also a normal MongoDB instance. Therefore, each shard[4] has its own data storage and anomaly generator. Depending on the desired number of shards, the corresponding number of query handlers has to be instantiated.

[4] A "shard" in the context of SickStore always means an instance of the query handler.

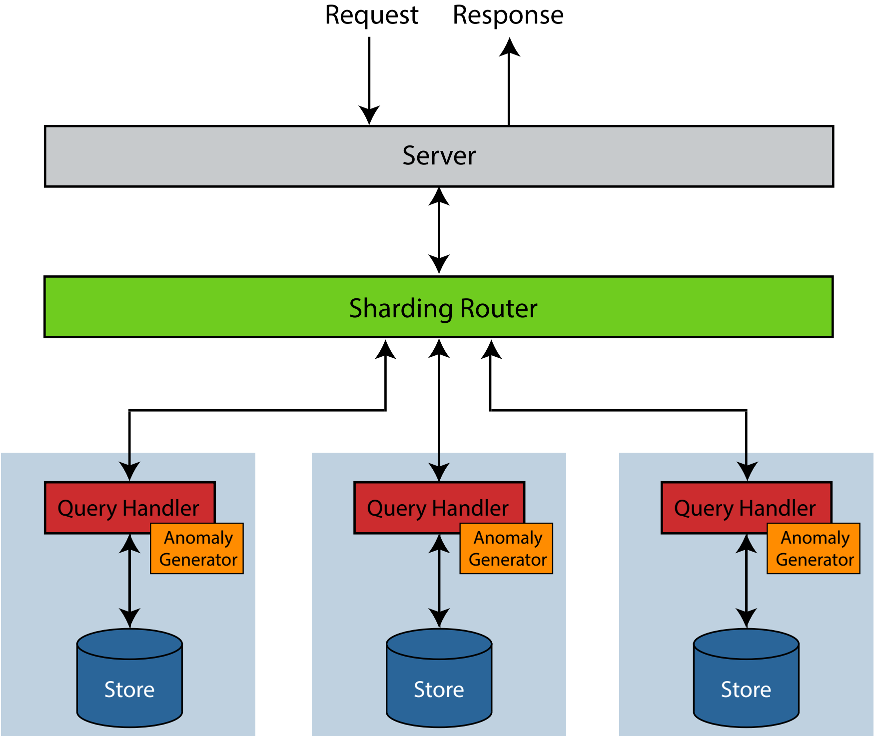

Figure 5.4.: Architecture of the SickStore server with three shards.

5.2.3. Sharding Router

The sharding router redirects incoming request to the *responsible shard*, according to the sharding strategy and the key of the item. Therefore, the router needs to be aware of all available shards and the strategy to use for segregating the data. Both pieces of information are passed as parameters to the router to instantiate it.

The responsible shard is determined by a *sharding strategy* in accordance with the request parameters. In this work, hash-based and range-based sharding strategies were implemented. Both will be explained in the following sections.

Read and write requests (insert, update and delete) are directly redirected to the responsible shard. Scan requests need to be processed differently, as multiple shards might contain parts of the queried data. Therefore, the sharding strategy is also responsible for the execution of scan requests. It sends the requests to the applicable shards and combines the result afterwards.

Previously, without sharding, it was possible to define a destination node that receives all client requests. The router respects this information and redirects those requests directly to the specified node. However, it is not recommended to specify a destination node and read or write directly to that node. This subverts the sharding strategy, so that items might not be found without specifying the node explicitly.

5.2.4. Hash-based sharding strategy

The hash-based sharding strategy was selected to be implemented in the context of this work, as it provides a simple sharding model that does not need any knowledge about the data that will be saved.

The hash-calculation is based on the *cyclic redundancy check (CRC)* algorithm. This algorithm is commonly used for error-detection and to detect changes in data. Additionally, this algorithm can be used as a hash function, even though it is not cryptographic safe. However, in this context cryptographic safety is not relevant; it still provides a random distribution of the data. The resulting hash will be divided by the number of nodes and the remainder[5] will state the responsible shard.

The responsible shard is determined by the `getTargetShard()` method. Its code can be found in Listing 5.1.

```java
public QueryHandlerInterface getTargetShard(ClientRequest request,
                                            List<QueryHandlerInterface> shards) {
    CRC32 crc = new CRC32();
    crc.update(request.getKey().getBytes());

    long hash = crc.getValue();
    long shardIndex = hash % shards.size();

    return shards.get((int) shardIndex);
}
```

</> Listing 5.1: Determination of the responsible shard within a hash-based sharding.

In the event of a scan request, this strategy needs to send the query to every shard, due to the random distribution of the data; the strategy and the router do not know on which shards the possible data items are stored. After all shards were queried, the results are united and sent back to the client. The implementation can be found in Listing A.8 (page 59).

5.2.5. Range-based sharding strategy

As a second strategy, range-based sharding was implemented into SickStore. It offers a manual assignment of key ranges to shards. Similar to MongoDB's range-based sharding, it divides the key space from a minimum to a maximum key into parts. However, in contrast to MongoDB, the ranges are not determined automatically, but needs to be configured manually.

The ranges can be configured by defining *separators* that divide the entire key space into the shard's ranges (similar to Figure 5.2). If there are n shards, $n-1$ separators need to be defined. Each separator defines the upper limit of a shard is therefore the *highest* key that is included

[5] This is a modulo calculation.

in the range. Per default, the comparison with the separator is case-sensitive. However, this strategy can be configured to be case-insensitive.

The comparison of keys with the separators, to identify the responsible shard, is based on a lexicographical comparison of both values. In the implementation, the method `compareTo()` of the `String` class in Java is used to compare both values. The implementation of the `determineShardIndex()` method is given in Listing 5.2.

```
public QueryHandlerInterface getTargetShard(ClientRequest request,
                                             List<QueryHandlerInterface> shards) {
    if (shards.size() != rangeSeparators.length + 1) {
        throw new RuntimeException("The number of separators does not correspond to "
        ↪   +
            "the number of shards (separators + 1 = shards)");
    }

    String key = request.getKey();
    if (!caseSensitive) {
        key = key.toLowerCase();
    }

    int shard = 0;
    for (String separator : rangeSeparators) {
        if (key.compareTo(separator) <= 0) {
            break;
        }
        shard++;
    }

    return shards.get(shard);
}
```

</> Listing 5.2: Determination of the responsible shard within a range-based sharding.

In case of scan requests, all shards need to be queried that might contain data in the requested range. To do so, it begins with the responsible shard of the specified key in the request and scans this shard. If it does not contain the requested number of items, the following shards need to be queried until enough records are found. The implementation can be found in Listing A.9 (page 60).

5.3. Evaluation

To finish the chapter about sharding with SickStore, the results will be evaluated in the following.

The initial goal of implementing a MongoDB-like sharding into SickStore was not reached. This was reasoned in Section 5.2.1 with different data models and the automatic splitting and re-

balancing of chunks. A simulation of those features would not be very accurate, as it would have nothing to do with the actual behavior of MongoDB, since it is not documented very well. For a proper simulation, it would be necessary to explore the internals of MongoDB, for example by examining the source code.

Nevertheless, it was still possible to implement sharding into SickStore, even though it is a much simpler implementation and does not consider MongoDB's sharding characteristics. It disseminates all requests according to a specific strategy across all shards. As exemplary strategies, hash-and range-based sharding were implemented. While the hash-based strategy distributes the data randomly across the available shards, is a configurable distribution of the data in accordance to the configured ranges possible with range-based sharding. These two strategies can be used as an example for further implementations.

Finally, it can be concluded that this implementation of sharding can be seen as the foundation for future research. By implementing additional strategies, more sophisticated behaviors can be simulated. It is even possible to simulate the sharding behavior of other database systems. For the implementation of a MongoDB-like sharding, a further investigation into the internals of MongoDB is necessary first.

With the implementation of sharding into SickStore in this chapter, the main part of this work is finished. In the following chapter, the results will be summarized and prospects for future research given.

6. Conclusion

Previously, the motivation for this work, the foundations of distributed database systems and the architecture of the prototypical database system SickStore were introduced. Subsequently, the replication behavior of MongoDB was examined and the simulation with SickStore described. In the end, MongoDB's sharding functionality was investigated and the simulation of sharding with SickStore explained. In the following, the subject and results of this work will be summarized in Section 6.1. The following Section 6.2 will finish this work with inspirations for further research.

6.1. Summary

Beginning with the **introduction**, this work was motivated with the large number of available NoSQL database systems and the difficult decision of which one to use. As the performance is an important factor for the decision, there is a demand on good benchmark tools. However, as those tools can be error-prone, the prototypical database system SickStore was introduced for benchmark validations. For that reason, the main objective of this work was to extend SickStore with a simulation of replication and sharding.

In the following chapter, the **foundations of distributed database systems** were introduced. At first, the types of distributed systems were examined and the acronyms BASE and CAP defined. Afterwards, the fundamentals of data distribution to handle huge amounts of data were introduced. In particular, sharding was explained which is used to distribute the data across multiple independent nodes. Subsequently, replication was introduced which enables data mirroring across multiple nodes to provide high-availability and failure tolerance. In the end, both concepts were combined.

To explain the later implementations of replication and sharding, the **architecture of SickStore** was outlined. On the one hand, the server simulates a distributed database system that consists of multiple nodes and is able to simulate different anomalies. On the other hand, the client library was introduced that allows communication with the server.

In the following chapter, the **simulation of replication with SickStore** introduced the main part of this work. At first, the replication system of MongoDB, which is based on replica sets, was surveyed. By defining write concerns, the client can require a specific level of assurance about the durability of a write request when it is confirmed. As the write concern has a direct impact on the actual write latency, the effects were estimated and validated in the following. Finally, the implementation into SickStore was explained and the results evaluated. The expected latencies were basically confirmed in the validation. However, there were minor variations in the measured latencies in comparison to the expected ones, which were reasoned with other factors that were not considered in the estimation.

In the end, the **simulation of sharding with SickStore** was covered. Similar to the simulation of replication, it should be inspired by MongoDB. However, during the examination of MongoDB's sharding functionality it became pretty clear that a proper simulation of this functionality would require a deep understanding of the internals of MongoDB, as the documentation describes only the principles. Nevertheless, sharding was still implemented into SickStore, but it is much more simplistic and does not consider MongoDB's behavior. In this work, hash-based and range-based sharding were introduced. By further examining MongoDB, SickStore can be extended in the future to simulate MongoDB's sharding.

6.2. Future Prospects

Based on the results of this work, some ideas for future research were already given in the evaluations. Those will be picked up and continued in the following.

In case of replication and journaling, variances between the measured and the expected write latencies were revealed. Those were reasoned with additional I/O to process the operation in MongoDB. However, the exact parameters that influence the latency need to be examined and considered in the simulation. Furthermore, this needs a deeper evaluation of MongoDB's internals.

The number of concurrent requests and the effects of parallel operations were also not considered in this work; it only dealt with one request at a time. The number of parallel operations will have an impact on the write latency, which might increase with the number of requests. Therefore, those effects need to be examined and included in the simulation.

In case of sharding, the simulation of MongoDB's behavior was not possible, since the effects were not documented and difficult to measure. Therefore, a proper simulation of MongoDB's sharding requires a deeper investigation of that, for example by examining the source code. Furthermore, the effects of the routing component can be examined. As the router is another distributed node that communicates with the shards, there will be a communication delay that can be included in the simulation.

Apart from refining the results of this work, there are many other aspects that can be considered to simulate the behaviors of distributed database systems. Those can be used to validate other aspects of benchmarks. For instance, MongoDB provides so called *read preferences* that allow reading data not only from the primary in a replica set, but also from secondaries. In this case, the client might observe other read latencies.

Moreover, the behavior of database systems other than MongoDB can be adapted and simulated with SickStore.

All in all, this work provides a good foundation for future research and student works to extend SickStore with more sophisticated replication and sharding simulations.

A. Listings

This section contains additional scripts that automate some of the described commands. Furthermore, it contains listings that describe parts of the SickStore implementations.

```bash
#!/bin/bash

nodes=4

# init replica set script
echo "rs.initiate({ _id: \"rs1\", members: [" > replicaset.js

for i in `seq 1 $nodes`; do
    name="mongo${i}"

    docker run --name $name \
        -d mongo --replSet rs1

    ip=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${name})
    echo -e "\t{ _id: ${i}, host: \"${ip}:27017\" }," >> replicaset.js
done

echo "] });" >> replicaset.js
echo "rs.status();" >> replicaset.js

echo "run the following scrpt on the primary node:"
echo
cat replicaset.js
rm replicaset.js
```

</> Listing A.1: Script to automate the replica set setup.

```
#!/bin/bash

containers=$(docker ps -q)
for id in $containers; do
    name=$(docker inspect -f "{{ .Name }}" $id | cut -d '/' -f 2)

    internal_interface=$(docker exec -t -i $name ip link show eth0)
    internal_id=$(echo "$internal_interface" | head -n 1 | cut -d ":" -f 1)

    external_id=$(($internal_id + 1))
    external_interface=$(ip link | grep "^$external_id" | cut -d ":" -f 2 | cut -c
    ↪  2-)

    echo -e "$external_interface\t$name"
done
```

</> Listing A.2: Script to detect Docker's virtual network interface names.

```java
private TreeSet<Long> findCustomDelays(Node receivedBy, Set<Node> nodes) {
    TreeSet<Long> delays = new TreeSet<>();

    // calculate the delays that occur by propagating the request to each node
    for (Node node : nodes) {
        if (node == receivedBy) {
            // there is no delay to the primary (the receiving node)
            // so, the node can be ignored
            continue;
        }

        long requestDelay = -1; // delay from primary to replica
        long responseDelay = -1; // delay from replica to primary
        for (NetworkDelay customDelay : customDelays) {
            if (customDelay.getFrom() == receivedBy && customDelay.getTo() == node) {
                requestDelay = customDelay.getDelay();
            }
            if (customDelay.getTo() == receivedBy && customDelay.getFrom() == node) {
                responseDelay = customDelay.getDelay();
            }
        }

        if (requestDelay == -1) {
            requestDelay = defaultDelay;
        }
        if (responseDelay == -1) {
            responseDelay = defaultDelay;
        }

        long delay = requestDelay + responseDelay;
        delays.add(delay);
    }

    return delays;
}
```

</> Listing A.3: Selection of the delays that are relevant for the current operation.

```
1  private long calculateObservableReplicationDelay(int acknowledgments, TreeSet<Long>
   ↪  delays) {
2      Iterator<Long> it = delays.iterator();
3      long delay = 0;
4      int acknowledgmentsLeft = acknowledgments;
5      while (it.hasNext() && acknowledgmentsLeft > 0) {
6          Long tmpDelay = it.next();
7          if (tmpDelay > delay) {
8              delay = tmpDelay;
9          }
10
11         acknowledgmentsLeft--;
12     }
13
14     return delay;
15 }
```

</> Listing A.4: Calculation of the observable replication delay that arises for the given number of acknowledgments and custom delays.

```java
private long calculateTaggedReplicationDelay(ClientWriteRequest request, Set<Node>
↪  nodes) {
    WriteConcern writeConcern = request.getWriteConcern();

    if (!tagSets.containsKey(writeConcern.getReplicaAcknowledgementTagSet())) {
        throw new IndexOutOfBoundsException("There is no tag-concern with name " +
            writeConcern.getReplicaAcknowledgementTagSet());
    }

    Map<String, Integer> tagSet =
    ↪  tagSets.get(writeConcern.getReplicaAcknowledgementTagSet());

    long delay = 0;
    for (HashMap.Entry<String, Integer> tagsetEntry : tagSet.entrySet()) {
        String tag = tagsetEntry.getKey();

        int acknowledgment = tagsetEntry.getValue();
        if (request.getReceivedBy().getTags().contains(tag)) {
            // if the receiving node has the current tag, reduce the number of
            ↪  acknowledgments
            acknowledgment--;
        }

        // find custom delays of nodes with the current tag
        TreeSet<Long> relevantDelays = findCustomDelaysWithTag(
            tag,
            request.getReceivedBy(),
            nodes
        );

        // calculate delay for this tag
        long tagDelay = calculateObservableReplicationDelay(acknowledgment,
        ↪  relevantDelays);
        if (tagDelay > delay) {
            delay = tagDelay;
        }
    }

    return delay;
}
```

</> Listing A.5: Calculation of the observable replication delay that arises from a write operation with tag set acknowledgment.

```java
private TreeSet<Long> findCustomDelaysWithTag(String tag,
                                              Node receivedBy,
                                              Set<Node> nodes) {
    // Find all nodes with that tag
    Set<Node> foundNodes = new HashSet<>();
    for (Node node : nodes) {
        if (node.getTags().contains(tag) && node != receivedBy) {
            foundNodes.add(node);
        }
    }

    return findCustomDelays(receivedBy, foundNodes);
}
```

</> Listing A.6: Selection of all custom delays of nodes that have the requested tag.

```java
public StalenessMap generateStalenessMap(Set<Node> nodes, ClientRequest request) {
    StalenessMap stalenessMap = new StalenessMap();

    for (Node node : nodes) {
        long staleness = -1;
        // find custom delay to replica (which is also the staleness value)

        if (node == request.getReceivedBy()) {
            // request arrived already on the primary, so there is no staleness
            staleness = 0;
        } else {
            // find the delay that occurs until the request arrives at the replica
            for (NetworkDelay delay : customDelays) {
                if (delay.getFrom() == request.getReceivedBy() && node ==
                ↪   delay.getTo()) {
                    staleness = delay.getDelay();

                    break;
                }
            }
        }

        // no custom delay found, use the default one
        if (staleness == -1) {
            staleness = defaultDelay;
        }

        stalenessMap.put(node, staleness);
    }

    return stalenessMap;
}
```

</> Listing A.7: Generation of the staleness map.

```java
public List<Version> doScanRequest(ClientRequestScan request,
↪   List<QueryHandlerInterface> shards) {
    // scan all shards for this range and put the Versions into the map
    TreeMap<String, Version> orderedVersions = new TreeMap<>();
    for (QueryHandlerInterface shard : shards) {
        ServerResponseScan response = (ServerResponseScan)
        ↪   shard.processQuery(request);

        for (Version version : response.getEntries()) {
            orderedVersions.put(version.getKey(), version);
        }
    }

    // afterwards select only the required number of versions
    int range = request.getRecordcount();
    boolean asc = request.isAscending();

    List<Version> versions = new ArrayList<>(range);
    String nextKey = request.getKey();

    do {
        versions.add(orderedVersions.get(nextKey));
    } while (range > versions.size() && (
        (asc && (nextKey = orderedVersions.higherKey(nextKey)) != null) ||
            (!asc && (nextKey = orderedVersions.lowerKey(nextKey)) != null)
    ));

    return versions;
}
```

</> Listing A.8: The `doScanRequests()` method of the hash-based strategy executes the scan request on all shards and combines the results.

```java
public List<Version> doScanRequest(ClientRequestScan request,
                                   List<QueryHandlerInterface> shards) {
    QueryHandlerInterface firstShard = getTargetShard(request, shards);

    int range = request.getRecordcount();
    List<Version> versions = new ArrayList<>(range);
    int i = shards.indexOf(firstShard);
    while (i < shards.size() && request.getRecordcount() > 0) {
        QueryHandlerInterface shard = shards.get(i);
        ServerResponseScan response = (ServerResponseScan)
        ↪   shard.processQuery(request);

        versions.addAll(response.getEntries());

        // reduce the number of needed records
        request.setRecordcount(request.getRecordcount() -
        ↪   response.getEntries().size());

        i++;
    }

    return versions;
}
```

</> Listing A.9: The `doScanRequests()` method of the range-based strategy begins with the responsible shard for the incoming key and queries all successors if necessary.

Bibliography

[AG15] ANDLINGER, Paul ; GELBMANN, Matthias: *MongoDB is the DBMS of the year, defending the title from last year.* `http://db-engines.com/en/blog_post/41` (Accessed on 29.05.2015), January 2015

[Bre00] BREWER, Eric A.: Towards robust distributed systems (abstract). In: [Nei00], 7

[Bre12] BREWER, Eric A.: CAP twelve years later: How the "rules" have changed. In: *Computer* 45 (2012), Feb, Nr. 2, S. 23–29. `http://dx.doi.org/10.1109/MC.2012.37`. – DOI 10.1109/MC.2012.37. – ISSN 0018–9162

[CST+10] COOPER, Brian F. ; SILBERSTEIN, Adam ; TAM, Erwin ; RAMAKRISHNAN, Raghu ; SEARS, Russell: Benchmarking cloud serving systems with YCSB. In: HELLERSTEIN, Joseph M. (Hrsg.) ; CHAUDHURI, Surajit (Hrsg.) ; ROSENBLUM, Mendel (Hrsg.): *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, ACM, 2010. – ISBN 978–1–4503–0036–0, S. 143–154

[DFNR14] DEY, Akon ; FEKETE, Alan ; NAMBIAR, Raghunath ; RÖHM, Uwe: YCSB+T: Benchmarking web-scale transactional databases. In: *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, IEEE, 2014, 223–230

[Doc15] DOCKER: *Network Configuration.* `https://docs.docker.com/articles/networking/` (Accessed on 06.06.2015, Version 1.6), June 2015

[DPCC13] DIFALLAH, Djellel E. ; PAVLO, Andrew ; CURINO, Carlo ; CUDRÉ-MAUROUX, Philippe: OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. In: *PVLDB* 7 (2013), Nr. 4, 277–288. `http://www.vldb.org/pvldb/vol7/p277-difallah.pdf`

[FB99] FOX, Armando ; BREWER, Eric A.: Harvest, Yield and Scalable Tolerant Systems. In: DRUSCHEL, Peter (Hrsg.): *Proceedings of The Seventh Workshop on Hot Topics in Operating Systems, HotOS-VII, Rio Rico, Arizona, USA, March 28-30, 1999*, IEEE Computer Society, 1999, 174–178

[FGC+97] FOX, Armando ; GRIBBLE, Steven D. ; CHAWATHE, Yatin ; BREWER, Eric A. ; GAUTHIER, Paul: Cluster-Based Scalable Network Services. In: *SOSP*, 1997, 78–91

[Fou09] FOUNDATION, Linux: *netem.* `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem` (Accessed on 26.05.2015), November 2009

[FWGR14] FRIEDRICH, Steffen ; WINGERATH, Wolfram ; GESSERT, Felix ; RITTER, Norbert: NoSQL OLTP Benchmarking: A Survey. In: PLÖDEREDER, Erhard (Hrsg.) ; GRUNSKE, Lars (Hrsg.) ; SCHNEIDER, Eric (Hrsg.) ; ULL, Dominik (Hrsg.): *44. Jahrestagung der*

Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland Bd. 232, GI, 2014 (LNI). – ISBN 978–3–88579–626–8, S. 693–704

[GHOS96] Gray, Jim ; Helland, Pat ; O'Neil, Patrick E. ; Shasha, Dennis: The Dangers of Replication and a Solution. In: Jagadish, H. V. (Hrsg.) ; Mumick, Inderpal S. (Hrsg.): *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, ACM Press, 1996, S. 173–182

[GL02] Gilbert, Seth ; Lynch, Nancy A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *SIGACT News* 33 (2002), Nr. 2, 51–59. http://dx.doi.org/10.1145/564585.564601. – DOI 10.1145/564585.564601

[Hof09] Hoff, Todd: *An Unorthodox Approach To Database Design: The Coming Of The Shard.* http://highscalability.com/blog/2009/8/6/an-unorthodox-approach-to-database-design-the-coming-of-the.html (Accessed on 13.04.2015), August 2009

[LS13] Lehner, Wolfgang ; Sattler, Kai-Uwe: *Web-Scale Data Management for the Cloud.* Springer Publishing Company, Incorporated, 2013. – ISBN 1461468558, 9781461468554

[Mon15] MongoDB, Inc: *MongoDB Documentation.* Release 3.0.3, May 2015 2015. http://docs.mongodb.org/master/MongoDB-manual.pdf

[Nei00] Neiger, Gil (Hrsg.): *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.* ACM, 2000 . – ISBN 1–58113–183–6

[PPR+11] Patil, Swapnil ; Polte, Milo ; Ren, Kai ; Tantisiriroj, Wittawat ; Xiao, Lin ; López, Julio ; Gibson, Garth ; Fuchs, Adam ; Rinaldi, Billie: YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In: Chase, Jeffrey S. (Hrsg.) ; Abbadi, Amr E. (Hrsg.): *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, ACM, 2011. – ISBN 978–1–4503–0976–9, 9

[SC11] Stonebraker, Michael ; Cattell, Rick: 10 rules for scalable performance in 'simple operation' datastores. In: *Commun. ACM* 54 (2011), Nr. 6, 72–80. http://dx.doi.org/10.1145/1953122.1953144. – DOI 10.1145/1953122.1953144

[SF12] Sadalage, Pramod J. ; Fowler, Martin: *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* 1st. Addison-Wesley Professional, 2012. – ISBN 0321826620, 9780321826626

[Sto86] Stonebraker, Michael: The Case for Shared Nothing. In: *IEEE Database Eng. Bull.* 9 (1986), Nr. 1, 4–9. http://sites.computer.org/debull/86MAR-CD.pdf

[WFGR15] Wingerath, Wolfram ; Friedrich, Steffen ; Gessert, Felix ; Ritter, Norbert: Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking. In: Seidl, Thomas (Hrsg.) ; Ritter, Norbert (Hrsg.) ; Schöning, Harald (Hrsg.) ; Sattler, Kai-Uwe (Hrsg.) ; Härder, Theo (Hrsg.) ; Friedrich, Steffen (Hrsg.) ; Wingerath, Wolfram (Hrsg.): *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings* Bd. 241, GI, 2015 (LNI). – ISBN 978–3–88579–635–0, 351–360

[WSP+00] Wiesmann, Matthias ; Schiper, André ; Pedone, Fernando ; Kemme, Bettina ; Alonso, Gustavo: Database Replication Techniques: A Three Parameter Classification. In: *SRDS*, 2000, 206–215

Declaration

Ich versichere, dass ich die Bachelorarbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Außerdem erkläre ich, dass ich mit der Einstellung dieser Bachelorarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden bin.

Ort und Datum

Unterschrift