ROYAL INSTITUTE
OF TECHNOLOGY

Load Balancing in a Distributed Storage System for Big and Small Data

LARS KROLL

Master of Science Thesis in
Information and Communication Technology
Supervisor: Dr. Jim Dowling
Examiner: Prof. Seif Haridi
Stockholm, Sweden, June 2013

TRITA-ICT-EX-2013:180

Abstract

Distributed storage services form the backbone of modern large-scale applications and data processing solutions. In this integral role they have to provide a scalable, reliable and performant service. One of the major challenges any distributed storage system has to address is skew in the data load, which can either be in the distribution of *data items* or *data access* over the nodes in the system. One widespread approach to deal with skewed load is data assignment based on uniform consistent hashing. However, there is an opposing desire to optimise and exploit data-locality. That is to say, it is advantageous to collocate items that are typically accessed together. Often this locality property can be achieved by storing keys in an ordered fashion and using application level knowledge to construct keys in such a way that items accessed together will end up very close together in the key space. It can easily be seen, however, that this behaviour exacerbates the load skew issue. A different approach to load balancing is partitioning the data into small subsets which can be relocated independently. These subsets may be known as partitions, tablets or virtual nodes, for example.

In this thesis we present the design of *CaracalDB*, a distributed key-value store which provides automatic load-balancing and data-locality, as well as fast re-replication after node failures, while remaining flexible enough to support different consistency levels to choose from. We also evaluate an early prototype of the system, and show that the approach is viable.

Acknowledgements

I would like to thank my examiner *Prof. Seif Haridi* and my supervisor *Dr. Jim Dowling* firstly for providing me with the opportunity to carry out this research and secondly for guiding me through it, as well. Especially Dr. Dowling has been a constant presence with ideas and council, pointing out important issues or related research.

Furthermore, I wish to convey my gratefulness to *Paris Carbone*, *Alexandru "Alex" Ormenisan* and *Jakob Oesinghaus*, as well as *Dr. Cosmin Arad*, *Dr. Tallat Shafaat* and *Niklas Ekström* for providing a constant stream of valuable input and feedback on my work and ideas.

I am thankful to *my friends* and my sister *Inga* for providing encouragement and moral support even from thousands of kilometres away.

And last but certainly not least, I wish to express my deepest gratitude to *my parents* who have been with me all this way for the last 24 years. Without them nothing I have done so far would have been possible.

Contents

Chapter 1

Introduction

At a time where global data storage requirements have passed the zettabyte (i.e. 10^{21} bytes) mark, the development of ever larger storage systems is not just a necessity but also a challenge. Even a very dense storage medium like deoxyribonucleic acid (DNA) has so far been shown to hold no more than 700 terabytes per gram [1]. Considering that storing a zettabyte of information in DNA in this way would require more than a ton of genetic material, the argument can safely be made that data storage needs to be distributed for quantitative reasons alone. Of course, among the qualitative arguments for distributed storage systems are reliability and availability. This is to say that distribution, possibly at geographically remote locations, and concurrent accessibility by anything from tens to billions of users are just as important as raw storage capacity.
And as such, distributed storage systems have become widespread in basically any industry that uses storage systems at all. The largest of these systems are likely being employed by web giants like Google, Amazon, Facebook and Apple. But research organisations like CERN produce and store tens of petabytes per year, as well. And not only has all this data to be stored somewhere, it also needs to be located, accessed, processed and maybe even modified.
All these requirements imply the necessity for storage solutions that provide massive scalability, working on single rack installations just as well as on globally distributed multi-datacenter deployments with thousands of servers. This also implies that system management should be federated in some way to avoid bottlenecks or lack of availability. Furthermore, on these scales failures, be they in software or in hardware, cease to be an exception and become the rule. Even if any given component has a minuscule probability of error, the sum of all the components permanently operating for years at a time makes failures a given. And thus they need to be considered during planning of solutions and handled in implementations.
In addition to all the inherent challenges to distributed large scale storage, there is also a concern about the kind of data being stored and the way it is accessed. A typical concept here is called *big data*. And despite there being no clear definition as to what data actually qualifies, the idea to be considered is that *big data* is so huge it's

challenging to handle with the available resources or naïve solutions. When it comes to access, workloads are typically being categorised as online transaction processing (OLTP) for transaction based data entry and retrieval, and online analytical processing (OLAP) for analytical loads. These concepts are mostly used for relational database systems, though, and modern applications require more flexible solutions for processing. This trend is well exemplified by the MapReduce [2] pattern which was originally implemented by Google but today is the base for many open source systems, like Apache's Hadoop framework. The important consideration for these processing frameworks is that they deal with large datasets and long running tasks, where placement decisions for operations in relation to the data they are working on are crucial for feasibility. This is opposed to a service that only queries in the kilobyte range, but needs to minimise the latency of such requests, possibly because a (human) user is waiting for the reply and humans are not generally known for their patience. Additionally, streaming applications that require medium sized chunks of data in a specific order are common today. Consider Spotify or YouTube as examples. Either of these access patterns might deal whith huge amounts of data, but a storage solution would be hard pressed to cater efficiently to all of them while treating their data undifferentiated.

1.1 Motivation

It is clear from the elaborations above that the development of flexible, scalable and reliable distributed storage solutions is a necessity. Of course there is a large number of products already available today. Among them are relational database management systems (RDBMSs) like MySQL, which may be deployed in a distributed way with replication and sharded or as MySQL Cluster. Their limiting factor is usually scalability, and companies have to expend great effort to make their large-scale deployment feasible, as Facebook has done [3], for example.
In the *NoSQL* world there are systems like Amazon's Dynamo [4], Apache's Cassandra [5], Google's BigTable [6] or Riak. These systems scale very well and provide a range of options when it comes to consistency, availability and partition tolerance (CAP) trade-offs [7]. Generally, they don't provide transaction support like classical RDBMSs, though.
When it comes to the storage of large data blobs there are Amazon's simple storage service (S3), the Google file system (GFS) [8], Apache's Hadoop distributed file system (HDFS) [9] or the approach of Nightingale et al. [10]. These systems are basically distributed filesystems, intended for batch processing applications. Yet, projects like Apache's Hive [11] or Cloudera's Impala [12] strive to provide ad-hoc services on top of these distributed filesystems.
However, what is very rare is a system that provides support for small low-latency queries as well as long running batch processing tasks or streaming applications. One example of something similar is Yahoo!'s Walnut [13] system, which aims to provide a combined storage layer for PNUTS [14], a key-value store for small data

under 100kB, MObStor, an object store for unstructured medium sized data, and HDFS [9]. Walnut does not provide RDBMS-like transactional operations with atomicity, consistency, isolation and durability properties (ACID), though.
On the front of HDFS there is another problem which concerns metadata storage. In HDFS metadata is stored at a special node (or a group of nodes) called *namenode*. While work has been done to improve reliability of the namenode, like adding fail-over nodes, the actual problem is limited storage space of a single node, consequently limiting the capacity of the whole Hadoop cluster.
From these examples follows the need for a system that addresses their shortcomings, providing ACID low-latency access to small data, and (quasi-)unlimited storage capacity for big data as well as support for flexible processing solutions. And at the core of these requirements is a scalable and fast consistent key-value store with transaction support.

1.2 Contributions

In this thesis we present the design and an early prototype implementation of *CaracalDB*, a distributed key-value store for datacenter or cloud use with deployments of a few thousand servers. CaracalDB is designed to be flexible and cater to a variety of use-cases in the small data area, and also to serve as part of a future database system, that would allow for big and small data handling. With this in mind the contributions of this thesis are:

- The design and architecture of a distributed key-value store called CaracalDB as a base for future research.
- An evaluation of an early prototype state of the implementation of CaracalDB in Java using the *Kompics* [15] message-passing concurrency component framework.

1.3 Related Work

1.3.1 Walnut and PNUTS

Yahoo!'s *PNUTS* [14] is a geographically distributed database system by Yahoo!. It uses hashed or ordered tables for data storage and has provides a set of per-record consistency guarantees to choose from. PNUTS provides load balancing on two forms: For hashed tables it is inherent, due to the uniform random distribution of hashed keys. In addition to this PNUTS balances the number of tablets per storage unit. It does not take actual load, i.e. popularity, of a tablet into account, though.
As mentioned before *Walnut* [13] is an object store by Yahoo! that acts as a unified low-level storage layer for PNUTS, MObStor, and HDFS, providing support for a wide variety of storage needs and workloads. A Walnut *cluster* manages different types of data in a number of *collections* which are divided into *partitions* that form

the unit of replication. Walnut load balancing is done through partition placement decisions by a dedicated, centralised *partition manager* node, that has data like storage node topology, per-collection policies and per-partition, per-device and per-node statistics available to it.

1.3.2 Cassandra

Facebook's *Cassandra* [5] is a distributed storage system for structured objects indexed by a *row key*. Additionally, Cassandra's data model includes *tables*, basically distributed multi dimensional maps, which are also indexed by a key, and *column families* which are sets of columns in a fashion similar to BigTable [6]. Cassandra provides atomic per-replica operation on a single row key. To address data partitioning Cassandra uses order-preserving *consistent hashing* [16] to map keys to nodes in a fashion similar to the Chord distributed hash table (DHT) and also uses Stoica et al.'s system of moving nodes on the ring in order to facilitate load balancing [17].

1.3.3 Dynamo

Amazon's *Dynamo* [4] is a highly-available distributed key-value store that sacrifices consistency guarantees for availability and scalability. Dynamo provides eventual consistency only, which allows it to return an operation to a client and apply it to other replicas asynchonously. It uses a similar scheme for partitioning as Cassandra but does not utilise ordered hashes. Furthermore, Dynamo addresses non-uniformity of node distribution in the ring with virtual nodes (vnodes) and provides a slightly different partition-to-vnode assignment strategy which achieves better distribution of load across the vnodes and thus over the physical nodes.

1.3.4 Scatter

Scatter [18] is a distributed consistent key-value store. Scatter is highly decentralised and provides linearisable consistency in the face of (some) failures. In Scatter the DHT overlay is formed not on single nodes, but instead on node groups which also form the replication groups for a specific key range. This model is similar to *ID-Replication* [19]. In addition to the uniform key distribution through consistent hashing typical for DHTs, there are two mechanisms for load balancing in Scatter. The first policy directs a node that newly joins the system to randomly sample k groups and then join the one handling the largest number of operations. The second policy allows neighbouring groups to trade off their responsibility ranges based on load distribution.

1.3.5 CATS

CATS [20, 21, 15, 19], like Scatter, is a distributed scalable key-value store that provides linearisable consistency. It is very Chord-like [17] in its usage of periodic

stabilisation and successor list replication (sl-replication) in a ring-based key space on which nodes are randomly distributed. CATS uses an order-preserving hash function for the consistent hashing based data partitioning of Chord and stores its data in memory. To provide this high a level of consistency CATS introduces so called *consistent quorums*. A quorum is consistent if all members share the same view of the group. Arad, Shafaat and Haridi extend the ABD algorithm by Attiya et al. [22] with support for reconfiguration and consistent quorums.
CATS has also been extended to provide support for range queries [23] and persistence to stable storage [24].

1.3.6 Kompics

The only system in this list that is not designed for storage, Kompics [15] is a message-passing concurrency and component-based programming model designed to facilitate the development of distributed systems. It has implementations in multiple programming languages, but in this document we only consider the Java reference implementation. The implementation provides library and a domain specific language (DSL) for programming and testing component-based systems.
The basic building block of Kompics is the *component*, a reactive state-machine which can execute concurrently to other components. A component provides and requires a number of *ports* which specify an interface by declaring typed *events* that can be triggered on it. A port also defines the direction in which events can be triggered, thus separating its events into indications and requests. Internally a component subscribes a number of *handlers* to its ports which represent its procedures. A handler for a specific event type executes reactively when such an event is triggered on a port it is subscribed to. Externally the ports of different components are connected via *channels*, whereby a channel always connects the required port of one component to the provided port of another. A port can have an arbitrary number of channels connected to it. A channel forwards events that are triggered on one of its ports to the other, possibly filtering some out along the way.
The Kompics framework also provides support for whole-system repeatable simulations.

1.4 Report Outline

In the next chapter, chapter 2, we will provide some background for the work, formalising properties and presenting existing solutions that are relevant to our work. Chapter 3 explains the design and architecture of the CaracalDB distributed key-value store. Following that, chapter 4 will first elaborate on the current state of the CaracalDB prototype and then present an experiment and its results to validate the feasibility of the system. Finally, chapter 5 will have some concluding remarks as well as future work to be done.

Chapter 2

Background

2.1 Distributed Systems Properties

2.1.1 Consistency

In distributed storage systems, or more formally distributed shared memory systems, we commonly see three basic levels of consistency guarantees. These properties are specified for a single *register* or key in case of key-value stores.
A register is *atomic* or *linearisable* if and only if despite the possibility of multiple clients issuing concurrent operations which could reach different replicas in different orders, each operation appears to take effect instantaneously at some point between its invocation and its response [25]. That is to say, a read will always return the latest write in a globally observable sequence. Linearisability or atomic consistency is the strongest property for a register. Since linearisablity is a so called *local* property [25] it can be extended to refer to a whole storage system by saying that a storage system is atomically consistent if and only if all its registers are atomically consistent. Or stating the same for key-value stores: A key-value store is atomically consistent if and only if all operations on all of its keys are atomically consistent.
A weaker property called *sequential consistency* was introduced by Leslie Lamport [26] in 1979. It is very similar to atomic consistency, but only requires operations on a register to appear atomic from the point of view of a single process, while atomic consistency requires this property globally for all processes.
A register is called *eventually consistent* [27] when after a time of no writes eventually all reads return the latest value. Eventually consistency is a weaker property than sequential or linearisable consistency for a distributed storage system.

A bit apart stands a fourth notion of consistency, which was defined by Yahoo! for *records* in a distributed storage system, called *per-record timeline consistency* [14]. This property guarantees that all replicas apply all updates to a record r in the same order, increasing the version v_r^* of r every time. A read of r on any replica p will always return a consistent version v_r^p from this timeline, though there might exist a replica $o \neq p$ with $v_r^o > v_r^p$. This model allows PNUTS to give different guarantees on the recency of a read per operation, by deciding how many replicas to read from.

2.1.2 Consistent Quorums

There is a somewhat related notion of consistency for quorums. A quorum is subset of nodes in a replication group G of size n, chosen such that for any two consecutive operations a, b on the same register (or key) and their respective quorums q_a, q_b the relation $size(q_a) + size(q_b) > n$ always holds [28]. Let a *view* v over G be a tuple that defines important information for a replication group, identified by a view id $i_v \in \mathbb{N}$. Certainly v will contain a list of nodes that currently (i.e. in v) make up G, but depending on the algorithm it could also include other information such as the range that is being replicated. A quorum Q over G is called a *consistent quorum* if and only if for all $n, m \in Q$ with respective views v_n, v_m the property $v_n = v_m$ holds [20, 21, 15, 19].

2.2 Systems and Algorithms

2.2.1 Key-Value Stores

As amply demonstrated in 1.3 there is a huge variety of storage systems with vastly different application programming interfaces (APIs) out there. There is, however, a very simple API that is often used as a reference, which defines the following operations on a key $k \in K$ with a key space[1] K and a value $v \in V$ for a value set V.

- $\varnothing$ `put(`k`, `v`)`
- V `get(`k`)`

We shall call a system that provides a form of this API a *key-value store*. Sometimes an additional command $\varnothing$ `delete(`k`)` is given, but many authors simply replace it by `delete(`k`) = put(`k`, `$\perp$`)` where $\perp$ represents a special *null* value. For real systems, this definition is somewhat questionable as it brings some garbage collection issues with it[2], but for an abstract description it should be adequate.

2.2.2 ABD

In 1995 Attiya, Bar-Noy, and Dolev provided a message passing algorithm that implements an atomic shared $(1, N)$-register for static groups, that is a register with 1 dedicated write and N readers [22]. This algorithm was later extended for an (M, N)-register, using two phases for read and write operations by Lynch

[1]The therm *space* does not necessarily refer here to a specific one of the mathematical constructs. The set of keys should probably provide an associated distance metric, though. Technically, total order relations should suffice.

[2]Consider a system that stores timestamps with its values. When should a (ts, v) tuple be entirely removed from the underlying storage?

and Shvartsman [29]. Furthermore, this algorithm was augmented for decentralised dynamic reconfiguration using consistent quorums in [20, 21, 15, 19]. It is this reconfigurable version that is going to be referred to as *ABD* for the remainder of this document.

2.2.3 The Paxos Family

In the 1990s Leslie Lamport came up with a two phase protocol algorithm that solves *uniform consensus* for systems with eventual leader election. In this algorithm he divided the participants of the protocol into three groups: *Proposers*, *acceptors* and *learners*. In phase one proposers try to get a quorum of *promises* from the acceptors for a specific *ballot* number. Once a proposer has gathered such a quorum, it enters phase two by sending *accept* messages for a value to the acceptors. If there are no collisions the acceptors will *vote* on that value and send an *accepted* message to the learners, which again try to gather a quorum of these messages. If this is successful the value is *decided*.
L. Lamport wrote a paper packing this algorithm into an elaborate parable about greek politicians in the parliament on the island of *Paxos* [30]. Tragically, this paper was badly understood and it took another 22 years for its full significance to be officially recognised with an ACM SIGOPS Hall of Fame Award in 2012.
Today there is a large family of Paxos-based and Paxos-like algorithms, among them *Multi-Paxos* [30] which runs multiple instances of the Paxos protocol in parallel and *Fast-Paxos* [31] which aims to reduce the end to end message latency in the stable state. Furthermore, an implementation of the *atomic broadcast* abstraction [32], using a variant of Paxos to decide on the order of operations, might be called Paxos as well. Lastly, Paxos is used for replication in a replicated state machines (RSM) system in a similar fashion to the atomic broadcast abstraction. This can for example be seen in Google's Megastore [33] or in Scatter [18] where it is combined with a two phase commit protocol. Since the term Paxos is so ambiguous the following algorithm shall be referred to as Paxos or Paxos state machine replication (SMR) for the remainder of this document:
A RSM implementation which changes its state $s_i \in S$ (S being the set of possible states) to $s_{i+1} \in S$ by applying the command $c_i : S \to S$. that was decided in instance $i \in N$ of an underlying Multi-Paxos uniform consensus implementation, to s_i, or in short $s_{i+1} = c_i(s_i)$. The implementation shall also be reconfigurable by passing a special $reconfigure_v : S \to S$ command to the consensus protocol, which upon application changes the view v_i of state s_i to $v_{i+1} = v$ in s_{i+1}. State transfer during reconfiguration is not included in this protocol but it is certainly necessary and can be considered an extension.

2.3 Load Balancing

In order to be able to reason about the problem of load balancing later, it is necessary to define the problem and associated terms first. We use vocabulary that is close

[34], since we are facing a similar problem.

2.3.1 Terms

For every data item $v \in \mathcal{V}$ where $\mathcal{V}$ is the set of all possible data items, we define a number of load types, the set of which is $\mathcal{T}$. For the remainder of this work $\mathcal{T} = \{\text{size}, \text{accesses}/s\}$ shall be considered. The *load* $l_t : \mathcal{V} \to \mathbb{R}, t \in \mathcal{T}$ is a function that assigns each item in $\mathcal{V}$ its associated load of type t. For every *unit of replication* $U \subseteq \mathcal{V}$, there is an associated value $l_t^U = \sum_{v \in U} l_t(v)$ called the load of U. And for any node H in the system there is a set U_H containing all units of replication at that node, and an associated value $l_t^H = \sum_{U \in U_H} l_t^U$ called the load of H. Nodes also have a *capacity* $c_t^H \in \mathbb{R}$ of each type $t \in \mathcal{T}$ that limits the load they can handle of t. Thus the equation $l_t^H < c_t^H$ must be maintained as invariant, as violation would result in failure of H. A node also has a *utilisation* $u_t^H = \frac{l_t^H}{c_t^H}$ of a $t \in \mathcal{T}$. Furthermore, a set S of nodes in the system has a utilisation μ_t^S and and average utilisation $\hat{\mu}_t^S$ of $t \in \mathcal{T}$, given as:

$$\mu_t^S = \frac{\sum_{H \in S} l_t^H}{\sum_{H \in S} c_t^H}, \hat{\mu}_t^S = \frac{1}{|S|} \sum_{H \in S} u_t^H \tag{2.1}$$

In a similar way max_t^S and min_t^S are defined as the maximum and minimum utilisation of a node in S. In addition to the above a third parameter $\rho_U : S \times S \to \mathbb{R}$ is considered, which represents the *cost* of moving a replication unit U from one host in S to another. For most systems ρ_U will depend linearly on l_{size}^U. If the system is considered uniform in the sense that for all $H, H', H'' \in S, U \in H$ $\rho_U(H, H') = \rho_U(H, H'')$ then the model can be simplified to $\rho_U \in \mathbb{R}$ constant.
Let $L_S^H = \{l_t^U \mid t \in \mathcal{T}, U \in H\}$, $C_H = \{c_t^H \mid t \in \mathcal{T}\}$, $L_H = \{l_t^H \mid t \in \mathcal{T}\}$ and $L_S = \{(C_H, L_H, L_S^H) \mid H \in S\}$ and we call L_S the *system statistics* for S.
The set of *actions* $\mathcal{A}$ contains at the very least $\perp$, which is equivalent to "do nothing", and `MOVE(`U`, `H`)` for a unit of replication U and a host H. In our example we shall also consider an additional operation `SPLIT(`U`, `U_1`, `U_2`)`, that splits U into U_1 and U_2 such that $U_1 \cup U_2 = U$.

2.3.2 The Load Balancer

A *load balancer* is an algorithm that takes tuples (S, L_S) as input and produces $A \in \mathcal{A}$ as output. Since it might be desirable that a load balancer also keeps some internal state $\mathscr{S}$, we amend the definition to taking triples $(S, L_S, \mathscr{S})$ as input and producing output tuples $(A, \mathscr{S}')$ with $A \in \mathcal{A}$ and $\mathscr{S}'$ a new state. And since load balancing is an ongoing (hopefully) converging process the algorithm should execute in rounds with a constant[3] time interval between successive invocations.
Any such load balancing algorithm should strive for the following two conflicting load balancing goals (LBGs):

[3]arguably

1) Minimise the load imbalance in the system, that is minimise $max_t^S - \hat{\mu}_t^S$.

2) Minimise the sum of transfer costs ρ incurred for load balancing actions in the system.

Technically, LBG.1 could also be achieved by padding all nodes H with $l_T^H < max_t^S$ with extra load until all nodes have max_t^S. Since this is somewhat contra-productive, LBG.1 should be amended by saying that any attempt to change $max_t^S - \hat{\mu}_t^S$ should not influence μ_t^S.
Since LBG.2 is clearly in conflict with LBG.1, it is necessary to define a *gain* function $g : \mathcal{A} \to \mathbb{R}$ that can be weighted against the cost $\rho : \mathcal{A} \to \mathbb{R}$, that is the cost ρ of any transfer involved in $A \in \mathcal{A}$. Such a function is very specific to a load balancing algorithm, but any algorithm should strive to maximise $w : \mathcal{A} \to \mathbb{R}, A \mapsto \frac{g(A)}{\rho(A)}$ for all possible actions for an input tuple, but at least it should not output any $A \in \mathcal{A}$ with $w(A) \leq 0$. Furthermore, the choice of g should be such, that the system does not fall into an endless loop of moving some data from one host to the next, in response to minor imbalances. In order to achieve this gain-threshold might be considered, below which no actions are decided.
Another conflict in LBG.1 can occur when load balancing over different $t, t' \in \mathcal{T}$. This is conceptually similar to the problem of multi-resource allocation considered in [35], and might be approached in a similar way by a load balancing algorithm.

Chapter 3

CaracalDB Design and System Architecture

We have outlined a number of existing distributed storage solutions in chapter 1, and it has hopefully become clear that there is a great variety in the types of APIs and consistency guarantees provided by the presented systems and these chosen trade-offs are reflected in the respective systems' performance under different conditions. However, it is clear that there will never be a single system that is optimal for every application. Yet, many problems are a shared concern between all of these systems. Foremost among these are *bootstrapping*, i.e. starting up the system, *lookup*, that is locating of data on the nodes, and *load balancing* for access and storage distribution. We shall consider *data partitioning* to be part of lookup and load balancing.
To address this issue, we present *CaracalDB*, a distributed storage framework that provides solutions for bootstrapping, lookup, and load balancing, while remaining flexible on the choice of *replication* algorithm and underlying *local storage*.

3.1 Concepts

CaracalDB is basically a *key-value store* (s. 2.2.1) offering *put* and *get* operations, but the exact semantics of operations it can support depend mostly on the choice of replication algorithm and local storage. For example, with Paxos SMR CaracalDB could easily provide a compare-and-swap (CAS) operation.
CaracalDB has the concept of a *schema*, which is similar to a *collection* in Walnut [13]. A schema contains policies for the replication algorithm, the size of the replication group, the quorum size in that group, and the underlying storage implementation. It is likely that it will be extended in the future to also include information about the structure of values.
For every schema there exist one or more *partitions* in the system which have a maximum size of data they are responsible for. We call this limit the *capacity* of the partition and the exact value is configurable for the system. Realistic values are probably between 100MB and and 1GB, but there are no inherent assumptions on

this. A partition is uniquely identified by an id from the key space (s. below). In CaracalDB partitions are the unit of replication.
Each partition is mapped to exactly one *replication group* that has the correct number of nodes for the partition's schema in it. Multiple partitions can be assigned to the same replication group.

3.2 Key Space

CaracalDB is not a DHT since it does not hash its keys. Also its key space is not circular as in systems like Chord [17] or CATS [20, 21, 15, 19], but linear. It is defined herein:
Let $\mathcal{B}$ be the set of byte values. For ease of use we shall represent them in hexadecimal notation, hence $\mathcal{B} = \{00, 01, \ldots, FF\}$. A finite *sequence* over $\mathcal{B}$ is a function $s : N \to \mathcal{B}$ where $N \subset \mathbb{N}$ finite, such that either $1 \in N$ or $N = \varnothing$ and N forms a ring under modulo-$|N|$ arithmetic. Or in other words N is a gap-free prefix of $\mathbb{N}$. The set of all such sequences we shall call $\mathcal{K}$ and this is the key space of CaracalDB.

3.2.1 Topology

It will be necessary for our purposes that $\mathcal{K}$ has the topology of a totally ordered metric space. One example of such a space is $(\mathbb{R}, d)$ with the metric $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}, (x, y) \mapsto |x - y|$. Since this is very straight forward, one approach to applying a similar topology to $\mathcal{K}$ is to find an injective map $\kappa : \mathcal{K} \to \mathbb{R}$ and use the ordering and metric induced in this way.
So let the function $len : \mathcal{K} \to \mathbb{N}$ be defined as $len : s \mapsto |\text{dom}(s)|$ and give the length of a sequence s. Further let $\hat{\mathcal{B}} = \mathcal{B} \cup \{\varnothing\}$ and $val_{\mathcal{B}} : \hat{\mathcal{B}} \to \mathbb{N}$ such that $\varnothing \mapsto 0$ and all $b \in \mathcal{B}$ are mapped to their integer equivalents plus one. That is $val_{\mathcal{B}}(00) = 1, val_{\mathcal{B}}(01) = 2, \ldots, val_{\mathcal{B}}(FF) = 256$. Let further for any $s \in \mathcal{K}$ be $\hat{s} : \mathbb{N} \to \hat{\mathcal{B}}$ the function that maps an $i \in \text{dom}(s)$ to $s(i)$ and for any $i \in \mathbb{N} \setminus \text{dom}(s)$ to $\varnothing$. Clearly this $\hat{} : \mathcal{K} \to \hat{\mathcal{K}}$ function is bijective, hence the preimage of any $\hat{s} \in \hat{\mathcal{K}}$ is $s \in \mathcal{K}$.
Finally, let $\kappa : \mathcal{K} \to \mathbb{R}$ be defined as:

$$\kappa(s) = \sum_{i=1}^{\infty} val_{\mathcal{B}}(\hat{s}(i)) \cdot 257^{-i} \tag{3.1}$$

Theorem 1 (Injectivity). *κ is injective.*

Proof. We have to show that for all $x, y \in \mathcal{K}$ the expression $\kappa(x) = \kappa(y) \Rightarrow x = y$ holds. So let $x, y \in \mathcal{K}$ such that $\kappa(x) = \kappa(y)$. We shall provide a proof by contradiction.
So assume $x \neq y$, and without loss of generality $x(1) \neq y(1)$. Suppose $val_{\mathcal{B}}(\hat{x}(1)) <$

$val_{\mathcal{B}}(\hat{y}(1))$ (the opposite case is equivalent). Then we can see that

$$\begin{aligned}
\kappa(x) &= \sum_{i=1}^{\infty} val_{\mathcal{B}}(\hat{x}(i)) \cdot 257^{-i} \\
&= val_{\mathcal{B}}(\hat{x}(1)) \cdot \sum_{i=2}^{\infty} val_{\mathcal{B}}(\hat{x}(i)) \cdot 257^{-i} \\
&\leq val_{\mathcal{B}}(\hat{x}(1)) \cdot \sum_{i=2}^{\infty} 256 \cdot 257^{-i} \\
&= val_{\mathcal{B}}(\hat{x}(1)) \cdot 256 \cdot \sum_{i=2}^{\infty} 257^{-i} \\
&= val_{\mathcal{B}}(\hat{x}(1)) \cdot 256 \cdot \sum_{i=2}^{n+2} 257^{-i} && \text{with } n \text{ such that } \forall_{m>n} \hat{y}(m) = 0 \\
&< val_{\mathcal{B}}(\hat{y}(1)) \cdot 256 \cdot \sum_{i=2}^{n+2} 257^{-i} \\
&= val_{\mathcal{B}}(\hat{y}(1)) \cdot 256 \cdot \sum_{i=0}^{n} 257^{-i-2} \\
&= val_{\mathcal{B}}(\hat{y}(1)) \cdot \frac{256}{257^2} \cdot \sum_{i=0}^{n} (\frac{1}{257})^i \\
&= val_{\mathcal{B}}(\hat{y}(1)) \cdot \frac{256}{257^2} \cdot \frac{1 - (\frac{1}{257})^n}{1 - \frac{1}{257}} && \text{(s. finite sums of the geometric series)} \\
&= val_{\mathcal{B}}(\hat{y}(1)) \cdot \frac{256}{257^2} \cdot \frac{1 - (\frac{1}{257})^n}{\frac{256}{257}} \\
&< val_{\mathcal{B}}(\hat{y}(1)) \cdot \frac{256}{257^2} \cdot \frac{1}{\frac{256}{257}} && \text{since } 1 - (\frac{1}{257})^n < 1 \\
&= val_{\mathcal{B}}(\hat{y}(1)) \cdot \frac{1}{257} \\
&< val_{\mathcal{B}}(\hat{y}(1)) < \sum_{i=1}^{\infty} val_{\mathcal{B}}(\hat{y}(i)) \cdot 257^{-i} \\
&= \kappa(y)
\end{aligned}$$

This is a contradiction to $\kappa(x) = \kappa(y)$. □

Now that we have shown that κ is injective, it remains to define the ordering relation and the metric. So let $<_{\mathcal{K}} \subset \mathcal{K} \times \mathcal{K}$ be relation such that for all $x, y \in \mathcal{K}$ the expression $x <_{\mathcal{K}} y \Leftrightarrow \kappa(x) < \kappa(y)$ holds. Since κ is injective and $<$ is a total ordering on $\mathbb{R}$, $<_{\mathcal{K}}$ imposes a total ordering on $\mathcal{K}$. The minimum of $\mathcal{K}$ is $s_0 : \varnothing \to \{\varnothing\}$ and the supremum of $\mathcal{K}$ is given by $s_\infty : \mathbb{N} \to \{FF\}$. For the remainder of this document whenever two keys in $\mathcal{K}$ are compared with $<, >, \leq, \geq$, min or max the usage of $<_{\mathcal{K}}$ is implied.
In the same way we define the induced metric $d : \mathcal{K} \times \mathcal{K} \to \mathbb{R}, (x, y) \mapsto |\kappa(x) - \kappa(y)|$.

Since $\mathbb{R} \times \mathbb{R} \to \mathbb{R}, (a,b) \mapsto |a-b|$ is a metric in $\mathbb{R}$ and κ is injective, it follows that $(\mathcal{K}, d)$ is a metric space. Consequently for the remainder of the document when we say that $\mathcal{K}$ is a key space, we mean that K is the set of keys and $(\mathcal{K}, d)$ forms a metric space.

3.2.2 Conventions

To make writing about elements of $\mathcal{K}$ easier the following convention shall be observed: For any $k \in \mathcal{K}$ with $n = len(k) > 0$ let $k_i = k(i), 0 \leq i < n$ and we can write $k = (k_0, \dots, k_{n-1})$. In case of $len(k) = 0$ we simply write $() = s_0$.
Furthermore, we define the *concatenate* operation $\vee : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$ such that for two keys $k, k' \in \mathcal{K}$ with $n = len(k), m = len(k'), n, m > 0$, using infix notation, $k \vee k' = (k_0, \dots, k_{n-1}, k'_0, \dots, k'_{m-1})$. Trivially for any $k \in \mathcal{K}$ it holds that $k \vee \varnothing = k$ and $\varnothing \vee k = k$. Additionally, let the *prefix relation* $\prec \subset \mathcal{K} \times \mathcal{K}$ be a relation such that

$$(k, k') \in \prec \iff \exists_{l \in \mathcal{K}}\ k \vee l = k'.$$

Finally, we define $\wedge : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$ as the *longest shared prefix* function such that

$$p = k \wedge k' \iff p \prec k, p \prec k' \text{ and } (\forall_{p' \in \mathcal{K}, p' \neq p}\ p' \prec k, p' \prec k' \Rightarrow len(p') < len(p)).$$

3.3 Virtual Nodes

CaracalDB employs the concept of *vnodes*, which are globally uniquely addressable entities that share a network port on a certain physical node, which we shall henceforth call a *host* to avoid confusion between the terms. A vnode is assigned a key $k \in \mathcal{K}$ upon creation, which will not be changed over the lifetime of the vnode. Both hosts and vnodes are addressable over the network, with hosts having an address of the form $(ip, port, \bot) = (ip, port)$ while vnodes' addresses are $(ip, port, k)$. Please note that, while k will not usually be globally unique, the triples $(ip, port, \bot)$ and $(ip, port, k)$ will be.
In Kompics terms a vnode represents a subtree of the full component tree, connected to the common network port via a channel filtered on the vnode's id. It may also share other components from the host, but generally those connections won't be filtered but either be a one-way communication from vnode to host, or utilise a request-response-pattern which keeps track of the exact path an event traveled.
In CaracalDB a partition p with key $k_p \in \mathcal{K}$ of schema s with replication degree deg_s will be assigned to at least deg_s vnodes on different hosts with addresses of the form $(ip, port, k_p)$. This assignment makes all the vnodes with k_p *members* of p, call them m_p and the hosts of m_p the replication group of p. Recall that multiple partitions might be assigned to a single replication group. However, the members of p are only responsible for p and will never be changed to another partition.

There are three main reasons for the usage of vnodes in CaracalDB. Firstly, we can start a different set of components on a per-vnode basis. This is the reason

that CaracalDB can support different replication algorithms and operations per collection.
Secondly, vnodes allow us to distribute the load better over the available hosts in the system and also give us a unit that we can move around in the cluster to change the current load distribution.
And lastly, we can use clever vnodes assignments to replication groups to improve the re-replication time in case of host failures. This is an idea that was exploited to an extreme in [10], but it also applies on our coarser partitioned case. To exemplify the problem, we consider a system where each host stores a single dataset which is replicated at two other nodes. Let hosts a, b, c form a replication group and consider what happens when a dies. Since the data that was on a needs to be re-replicated to preserve the replication degree a new node d is commissioned into the replication group (d could have been a standby node, or carry data in another overlay, it makes little difference). Since the data d needs is only available at b, c it needs to be copied from there making the disk speeds of b, c, d a re-replication performance bottleneck. In this model it could take hours to re-replicate large amounts of data, leaving the system in a state where a single failure of b or c could leave it unable to take decisions on that group or a failure of both hosts could even result in data loss. During all this time the other hosts in the system are sitting idle in terms of re-replication of the data from a. They might of course be running normal operations or re-replication from another failure.
As opposed to this consider a system where data is partitioned and partitions are mapped to a number of different replication groups. If there are p partitions all with replication degree deg over h hosts that means there will be $avg_p = \frac{p \cdot deg}{h}$ vnodes on average per host, if the system is in a well balanced state. Now consider if a single host a fails. If p is large enough there is a good chance that every other host in the system has some replication group in common with a. If this system re-replicates by assigning the failed vnode for every partition to a random other host that doesn't already have a replica of the data, for example, the result will be that re-replication is done using all available disks in the cluster. Every host will most likely be transferring data to and from a number of different hosts. This will be much faster than the previous solution. Of course, it also means that normal operations for all partitions in the cluster are affected. Yet, we argue that it is preferable to have a latency increase on all operations in the cluster for a relatively short time, than to have a latency increase in only a small subset but for a very long time with the added higher possibility for data loss.

3.4 Data Partitioning

As explained CaracalDB provides schemas and splits their data into partitions. However, we have not yet shown how the full key-space $\mathcal{K}$ is split up into the partitions. To see this, first of all note that from a key space perspective a schema is nothing more but a prefix on all keys in all partitions of the schema. This prefix

is at the same time the schema id, so if we want a schema for users we could simply give it the id $(75, 73, 65, 72, 73) \in \mathcal{K}$ (ASCII for "users"). This is certainly a valid schema id, however, one should be careful that different schemas don't have ids that are prefixes of each other. A safer solution would be to hash schema names to obtain their ids, resulting in equal length ids, which clearly can't be prefixes of each other. CaracalDB makes no assumptions on how schema ids are created, though, and developers are free to choose them by whatever means seem appropriate.
It was already mentioned that partitions are also assigned an id from $\mathcal{K}$ and now it is clear that for any schema s with id $k_s \in \mathcal{K}$ and partition p of s with key $k_p \in \mathcal{K}$ k_s must be a prefix of k_p. In fact there will always be exactly one partition with $k_s = k_p$, so that the layout of schemas on the key space is clearly delimited and no vnode can end up being responsible for data from more than one schema. We call this property *alignment*, named after the way structs are laid out in memory by languages like C and C++.
Let $\mathcal{P} \subset \mathcal{K}$ be the set of all partition ids in the system at any particular point in time. $\mathcal{P}$ is still totally ordered by restricting $<_{\mathcal{K}}$ to $\mathcal{P}$. With this in mind let $succ : \mathcal{P} \to \mathcal{P} \cup \{s_\infty\}$ be the function that maps each $k \in \mathcal{P}$ to the smallest value $k' \in \mathcal{P}$ with $k < k'$ or to s_∞ if k is the maximum of $\mathcal{P}$.
We then define the *responsibility* R_p of a partition p with key $k_p \in \mathcal{P}$ as $R_p \subseteq \mathcal{K}, R_p = [k_p, succ(k_p))$.
This is opposed to the way Chord and CATS handle this where a node with id n is responsible for the range $(pred(n), n]$ [17, 20, 21, 15, 19]. The reason for our decision to diverge from the well established pattern, is that in our linear, lexicographically sorted key space it is more intuitive if a partition is responsible for any key of which its own id is a prefix, up to the id of its successor. Furthermore, this system allows us to have the very intuitive alignment property presented above.

3.5 Lookup and the Global View

3.5.1 Locating Data

There are many solutions for locating data in a distributed storage system. Most of them involve assigning data and hosts into a bucket, for example by using a hash function. And then using a routing scheme to locate the host with the right bucket, and thus the desired data. Chord [17] for example hashes keys and assigns nodes ids in the same key space. A key will then be stored at the node with next greater id on the modulo ring. To route a request to that node Chord maintains a number of fingers to other nodes in the ring, such that whenever a finger is followed the distance to the target node is at least halved. In this way Chord can locate a key in $O(\log N)$ messages.
An alternative approach to this is keeping a full view of the system membership and responsibilities at every node in the system and maybe even distribute this view to clients outside the storage system, as Nightingale et al. are doing for example [10]. Such a system can guarantee $O(1)$ routing and is aimed at reducing the latency

of operations by avoiding as many network trips as possible. Depending on how current this global view is kept at different locations, such a system can do *single-hop lookup*, by sending the request directly from a client to the responsible system node. It could also implement *two-hop lookup*, by sending the request from client to a random node in the system and forwarding from there to the responsible node. If views can be outdated at the system's nodes some bounded *multi-hop lookup* can be implemented, whereby the client sends to a random node in the system and that node forwards to the node its current view of the system (which might be outdated) suggests is responsible and so on, until the an up-to-date view is found and the right node located.
Clearly the space requirements for such a full view increase linearly with the number of nodes in the system, making it very difficult to scale to very large cluster sizes. Despite this fact, CaracalDB uses exactly such a full view system with possibly outdated views which can optionally be installed at clients. We simply want to provide the fastest possible steady state lookup, even if we might pay for this during high-churn episodes. In this sense CaracalDB provides all three full view lookup versions described above. It provides single-hop lookup if the system is in a steady state and a client has a recent version of the global lookup table (LUT) (s. below) installed. If the client is *thin* we provide two-hop lookup in the steady state. And if the system is undergoing churn CaracalDB provides multi-hop lookup with a configurable bound ϵ.

3.5.2 The Lookup Table

The LUT is a data structure that maintains the current view of the CaracalDB system. It provides four types of information:

(1) A mapping from four byte integer ids, call them host ids for short, to host addresses.

(2) A mapping from four byte integer ids, call them replication group ids, to lists of host ids and an eight byte version number.

(3) An ordered mapping from partition ids ($\in \mathcal{K}$) to replication group ids.

(4) A mapping from schema ids ($\in \mathcal{K}$) to associated schema information (s. 3.1).

The first two are not a major size problem. Even for large systems with $n = 10000$ nodes, (1) will be around 120kB. For (2) we assume an average replication group size of three and also $3n$ replication groups and get 600kB. Note that systems like [10] use n^2 replication groups for this table, but their load distribution is dependent on the size of the table, while ours is not or at least to a lesser degree. Thus, 3 seems to be an amply large factor, which can of course be adapted if necessary. The larger this factor, the more even will be the distribution of vnodes over hosts and the larger the diversity of replication groups per host. Furthermore, (4) will likely not be an issue since it only grows in the number of schemas, which is probably

relatively small even for a large system.
Clearly the major issues are with (3). Consider a system where each host has a 1TB hard disk drive (HDD) and partitions have a maximum size of 100MB (after this they get split). If this system is fully loaded that means there will be 10000 vnodes per host, so 100 million vnodes in the system and with the average replication degree of 3 this means there will be over 33 million partitions in the cluster. Each such partition is mapped to a four byte integer in (3) resulting in 132 MB of data for the replication groups ids alone. This is, however, still bearable for a node that is member of the system or even a thick client like an application server for example. But this figure does not include the size of the partition keys in the map itself. If these keys are naïvely stored, even a conservative estimate of 20 bytes per key brings the total size of (3) up to 660MB plus the 132MB from above makes almost 800MB. This is close to 1GB of LUT data. A new node joining the system would spend quite some time just getting a current copy of this LUT.

3.5.3 Space Optimisations for the LUT

However, there is a better way of storing (3). In their paper for a solid state disk (SSD) optimised local key value store called *SILT*, Lim et al. found a very space efficient way of storing huge key to inode mappings in memory [36]. They keep the keys in a sorted *prefix tree*, also called a *trie*. This is a tree data structure that stores a mapping of keys to values where each leaf node represents one key in the array plus its value, and each internal node represents the longest common prefix of the keys represented by its descendants. However, most common trie implementations use pointers, thus ruining their space advantage. Lim et al., however, came up with a pointer-free representation that takes full advantage of the space saving capability of "sharing" prefixes. This representation has one disadvantage, though: It is immutable. Whenever changes are made to the mapping the whole structure has to be rewritten. To make their system viable despite this, they proposed two additional solutions. Firstly, they use two intermediate stores with lower space efficiency but better write performance. And secondly, they partition the actual store into a number of sub-stores, so that when data is actually written to the trie, only a certain range is affected as opposed to the whole store.
Thus, our proposition for CaracalDB's LUT is as follows: Use a modified version of SILT for (3), such that it supports *ceiling* and *floor* operations as well as a serialisation protocol that includes first merging the intermediate stores into the trie, then exporting the whole prefix tree plus data (the replication groups ids) in exactly the in-memory representation proposed by Lim et al.. This blob of data can be compressed for further size reduction during transfer. Furthermore, the store for (3) shall be split into 256 sub-stores based on the first byte of the key. These sub-stores can be serialised and transferred independently allowing clients, for example, to lazily fetch only ranges of the LUT they are actually interested in. Furthermore it is not necessary to store the full key in-memory in the trie (SILT does not do this either [36]). The shortest prefix that makes a key unique in the

store at the time of insert is sufficient. However, in this case the full key needs to be stored on disk somewhere, so that it can be fetched during lookup in order to construct complete vnode addresses. Each of the sub-stores shall additionally be assigned an eight byte version number, so that clients and even servers can identify if they have an outdated version of it, and request specific LUT updates. The space savings gained by employing the described method are very hard to predict as they depend strongly on how long prefixes are on average between keys. Lim et al. report storage requirements of as little 0.4bytes/key with fixed size[1] keys and entropy encoded storage. Since we have variable length keys and can't use hashing or entropy encoding on them, as this would make the necessary *ceiling* and *floor* operations impossible, the gains will likely be smaller. To offset this, however, our keys are in no way uniformly distributed over the key space, since we specifically use prefixes to build them, which in turn works in our favour in the trie representation. From very quick and superficial experiments with a similar data representation we estimate a key size reduction to 20%. This would bring our LUT size down to around 250MB which remains manageable for a system of that size.

3.5.4 Managing the LUT

In order to make a global LUT viable and also to allow for a bound ϵ on the multi-hop lookup scheme described above there should be an always current *canonical* version of the LUT maintained somewhere in the system. CaracalDB uses a special partition for this, called the *zero partition* and depicted as 0_P. The range R_0 of 0_P is $[0_{\mathcal{K}}, (00, 00, 00, 01))$ and this range is also reserved in CaracalDB for maintenance information, in a similar fashion to the usage of *directories* in [34]. Additionally, 0_P is always assigned to the replication group with id 0, which always contains exactly three hosts. The replication protocol for 0_P is Paxos SMR.
In order to avoid superfluous changes in the LUT, which might involve expensive operations such as data transfer, only the member of 0_P that currently considers itself the leader (using the same leader detection component as the Paxos implementation) proposes new changes to the LUT. However, during periods of churn or network partitions, the Paxos algorithm still guarantees orderly transitions from one LUT version to the next, even if they might be based on false suspicions. Additionally, to ensure that hosts and vnodes at least have current information about themselves (i.e. their assigned partitions, their current view, their current range), for every decision that changes the LUT a secondary Paxos instance is run, which includes all hosts that are affected by the change (even the ones suspected to have failed) in addition to the members of 0_P and requires a majority quorum to succeed. To ensure that this majority can be achieved a proposed change should not include more than one host that is suspected to have failed.
For all non-affected hosts the changes to the LUT can be either simply broadcasted or gossiped. Nodes should take care, however, to apply changes always from one

[1]They neglect to mention the original size of the keys, though.

LUT version to the next without skipping. If a node finds that an update is missing in between its current state and an update that it got, it can request the missing update(s) from one of the member of 0_P.

3.5.5 ϵ-bounded Multi-Hop Lookup

It has been referred to a few times already that it is possible to bound the number of hops needed for lookup under churn to a specified maximum value ϵ in CaracalDB. The mechanism for this is really straight forward. Let us take any $\epsilon \geq 3$ (s. below for the reason for this lower bound). Now consider that for every message m that needs to be routed to a vnode that is responsible for a specific key k_m, call it n_r, we add a value $f_m \in \mathbb{N}$ to m that gets incremented by 1 every time the message is forwarded. After this increment happens, the forwarding node n checks whether or not $f_m + 2 = \epsilon$ holds. Should this not be the case, the message is forwarded normally to the address n thinks is responsible for k_m. However, should the property hold, then instead n forwards the message to any member n_0 of 0_P. As described in section 3.5.4 the members of 0_P provide a canonical version of the LUT. Hence after incrementing f_m to $\epsilon - 1$, n_0 can directly forward m to the currently responsible node n_r. In this way m will have been received by n_r in at most ϵ hops. Clearly the choice of ϵ influences the load on the members of 0_P, with larger ϵ values resulting in lower load. To see why ϵ should be at least 3, consider this: It takes two hops from the host that fulfils the $f_m + 2 = \epsilon$ condition to reach n_r (unless it happens to be a member of 0_P by chance). But this nodes should not be origin of the message, since all clients sending their requests through members of 0_P will result in system throughput bottlenecking on three hosts. Thus $\epsilon \geq 2 + 1$ for the hop from client to any host in the system.

3.6 Bootstrapping and Churn

In distributed systems node failures are common place. The larger the system, the greater the likelihood of such failures. Additionally, for a long running system failed nodes will need to be replaced by new nodes, or nodes might simply join to extend the system's capacity. These join and leave events of nodes are commonly referred to as *churn* events. Churn is a matter of fact for distributed systems and must be handled.

In distributed (storage) systems that are started by simply creating the required number of nodes and relying on the normal node join handling to converge into a single system in, extended startup times of multiple minutes or even hours are not uncommon, due to the rate of node joins being limited [37]. Hence many systems, like CATS for example [20, 21, 15, 19], provide a separate bootstrapping protocol that is run once at the start of the system. In CATS all nodes report to the bootstrap server, and when a certain threshold in the number of nodes is reached the bootstrap server locally generates a converged view of the ring topology and imposes it on the system's nodes, in order to avoid running many iterations of the

periodic stabilisation protocol.
CaracalDB provides services for bootstrapping even large deployments of over 1000 nodes in a coordinated manner, and handles the joining and leaving of hosts during runtime. However, it should be kept in mind that CaracalDB is aimed at datacenter deployments and is not meant to cope the with the levels of churn typically seen in large internet deployments of peer-to-peer (P2P) protocols like BitTorrent [38], for example.

3.6.1 The Bootstrap Protocol

CaracalDB uses a rather straight forward protocol for bootstrapping, which is also conceptually similar to the one employed in CATS [20, 21, 15, 19]. When CaracalDB is started all nodes enter in a *bootstrapping* phase. In this phase the only available services are network and timers. A single machine is declared to be the *BootstrapServer* by an administrator, and all other hosts will act as *BootstrapClient*s. The clients will need to be given the address of the server somehow. This could be done by the administrator or using a network service discovery protocol. The actual bootstrap protocol consists of three phases. In the first phase the server tries to collect a large enough number of hosts to pass a preconfigured threshold γ. The minimum value for γ is 3 since the replication group for 0_P is required to contain exactly three hosts. Clients regularly send `BootstrapRequest` messages, which for one serves to avoid problems caused by server and clients starting in the wrong order and more importantly is used by the server to keep track of who is actually alive. This is important in case nodes already fail during the collection phase. If the server fails at any time during bootstrapping, the process needs to be started anew, though.
Once the currently alive number of nodes that have registered at the server (counting the server itself as well) reaches or surpasses γ, that set of node becomes the *boot set*. The server proceeds into the second phase called *seeding* by generating an initial LUT using the boot set. This initial LUT contains in addition to the hosts of the boot set, a replication group mapping that is randomly generated from the boot set, such that all replication groups are of size 3 and no group contains a host more than once. As alluded to before it is desirable for this mapping to be as random as possible as it will affect the inherent load balancing and re-replication speed of the system later. However, since the mapping is not static, this property is not as crucial as in [10], for example. Additionally, a configurable number of initial partitions (in addition to 0_P) is more or less evenly distributed over the subspace $\mathcal{K}_4$ of four byte keys of $\mathcal{K}$. This is done so that in a schema-less system there will be some initial distribution of partitions. When schemas are added later during the runtime of the system, additional aligned partitions will be created.
Once the LUT generation is completed it is serialised and distributed to all hosts in the boot set. This can be done either via direct server to client transfer, or more efficiently using a gossiping broadcast algorithm. Additionally, if the server receives `BootstrapRequest` messages from new nodes during the seeding phase, it

will respond by sending them the initial LUT immediately.
Upon reception of a complete LUT clients stop sending `BootstrapRequest` messages and instead start to periodically send `Ready` messages. Once the server is aware that all nodes from the boot set that are still alive are also ready it broadcasts a `Boot` message to all alive nodes. Locally the server also triggers a `Bootstrapped` event that in addition to the LUT contains a list of hosts that failed and joined since the creation of the initial LUT. These nodes will be added/removed to/from the system via normal join/leave procedures.
When nodes receive the `Boot` message, they switch from bootstrapping to normal operations, by using the LUT to find their assigned vnodes and starting them up. After at least enough hosts have started such that every partition in the system can form a majority for its replication algorithm (the standard is Paxos SMR) the system is ready for operation.
During normal operations the LUT manager component of 0_P takes the responsibilities of the BootstrapServer.

3.6.2 Node Join

If a new node wants to join into an existing CaracalDB cluster, it starts in the same way as during bootstrapping. It could contact any node in the cluster to be routed to the members of 0_P, but usually an administrator will configure it with the correct address during start.
For the LUT manager component the work is somewhat different than during bootstrapping, though. Instead of generating a totally new LUT it needs to alter the replication groups in such a way that the mapping preserves its properties and the data movement involved is reasonable. Nightingale et al. have a very good approach in their Flat Datacenter Storage (FDS) system, which keeps the table size constant [10]. While this is required by their modulo based key to host mapping, for us it is neither necessary nor desirable to keep the number of replication groups constant. It is however desirable, yet not necessary, to keep the ratio of hosts to replication groups τ approximately constant. Also each host should be a member of $\tau \cdot avg_{deg}$ [2] replication groups on average, since there are $\tau \cdot n$ groups, each with avg_{deg} members, meaning $\tau \cdot n \cdot avg_{deg}$ entries in the mapping. Hence, for each joining node n the replication group mapping should be extended by τ rows, and from $\tau \cdot avg_{deg}$ random existing replication groups one node each is entered into those new rows and replaced by n in its original location. The amount of data n will receive by this method depends on how uniformly the load was distributed over the replication groups prior to its joining. It is however of no great concern, since mis-distributions can be corrected at a later time by the load balancing algorithm.
After the decisions are made the changes to the LUT are propagated in the manner described in section 3.5.4. The actual data transfer, however, is handled by the individual replication protocol for each partition n joins into. While CaracalDB

[2] avg_{deg} is the average replication degree in the system.

provides a general mechanism for transfer of large amounts of data, the correctness of some replication protocols depends on how data transfer is handled while normal operations are ongoing in parallel. Thus, we cannot take this responsibility away from the replication component. The same argument holds for the actual view change of the replication group.

3.6.3 Node Failure

In CaracalDB we assume that vnodes can not fail without their host failing as well. They can be stopped and started individually, but a crash of any vnode will take down its host and all other vnodes on it as well.
Hence, it is sufficient for us to provide failure detection on a per-host basis. Any vnode can subscribe to the state of any other vnode by using an eventual failure detector service provided by its host. If a host suspects another host to have failed it will inform all its vnodes that are subscribed to vnodes on that host. However, that is all that happens. Hosts are not responsible for the decision to replace another host, they merely take note of its (possible) failure and inform their subscribers.
In addition to this normal P2P like monitoring, hosts also regularly send heartbeats to the LUT manager. The timeouts for these should have a somewhat longer period in order to provide fewer false positives. If the LUT manager finally considers a host to have failed, it will replace all it's occurrences in the replication group mapping by another random node (without violating the no duplicates rule, of course). Should a series of failures reduce the number of hosts in the system to one, the last one simply shuts down, as is cannot provide any service anymore.
The application of the new LUT works exactly as for node joins.

3.7 Load Balancing

CaracalDB provides the setting for load balancing as discussed in section 2.3. Here again CaracalDB is very flexible and is not limited to one specific load balancing algorithm. It could use anything from a simple preprogrammed reactive state machine, to a complicated machine learning based implementation, that supports proactive behaviour based on learned patterns. Hence, we will not give any specific algorithm for load balancing in CaracalDB here, but instead point out, where decisions need to be made and what data is available to base those decisions on. Load balancing decisions in CaracalDB are federated to different levels, to reduce work for the LUT manager.

3.7.1 Global Level

Each host in CaracalDB periodically writes its current CPU, and disk space usage into a set of special keys in the reserved range R_0 of $\mathcal{K}$. For every load balancing period δ the LUT manager reads that range (which is local for all member of 0_P) and uses its data as input for the load balancing system. The possible actions of the global level load balancing algorithm in CaracalDB are $\mathcal{A} = \{\perp, \text{MOVEANY}(H), \text{MOVE}(U, H, H')\}$ for hosts H, H' and $U \in H$ a partition of H. If the algorithm produces an output $A \in \mathcal{A} \setminus \{\perp\}$ the following two cases can occur:

MOVEANY(H) Send a request to H to propose some partition $U \in H$ for being relocated to another host.

MOVE(U, H, H') Instructs the LUT manager to move U from H to H'. This is done by changing the *LUT* accordingly and then following the normal procedure for LUT changes as in node joins and node fails.

No other decisions will be taken on global level in CaracalDB and the MOVE decision is usually preceded by the response to the MOVEANY decision.

3.7.2 Host Level

In a similar fashion to how hosts write statistics into R_0, vnodes periodically inform their hosts of their current size and the number of operations they handled per second on average since the last report. Whenever a host received a MOVEANY message, the host level load balancing implementation can use this information to decide which partition it should report for movement. This could for example be the one with highest op/s to size ratio. The host level load balancer does not take any action on it's own.

3.7.3 Vnode Level

Vnodes need to keep track of some statistics about the data they are responsible for as well. They need to know the cumulative size of all their key value pairs, the number of op/s they handle and the access frequency of their top-k [39] keys for a certain interval in a fashion similar to [40]. In addition to this vnodes have two threshold values θ_s, θ_f that are constant for the whole system and configured by an administrator prior to startup. θ_s defines the maximum size of a partition before it needs to be split into two, and θ_f maximum load (in op/s) a single vnode can handle, before its partition should be split.
This information is available to the vnode level load balancer, that has the action set $\mathcal{A}_v = \{\perp, \text{SPLIT}(k_s)\}$ where $k_s \in \mathcal{K}$ is the id of the new partition. Such an algorithm can decide whether to split based on size, access frequency or a combination of both, by deciding on k_s such that by removing all data with keys $k_d \geq k_s$ from its own responsibility it achieves its target load.
If a vnode of partition P decided on a SPLIT operation it sends this recommendation to the LUT manager, which is then responsible for ensuring that it is executed and has to avoid executing similar decisions from other vnodes of P (that are likely to have come to the same or a similar conclusion and made a proposition to that end).

3.7.4 Data-locality Concerns

From the elaborations above it should be clear that placing data close together in the key space, that is with long shared prefixes, will often result in the data being placed in the same partition and thus on the same host. The only exception to this rule are partition boundaries which leads us to the following proposal concerning partition splits.
Instead of deciding a split on a single key k_s from the partition's store $S_p \subset \mathcal{K}$, it is advantageous for the load balancing algorithm to decide a range $R_s = [k_{min}, k_{max}]$, such that $k_{min}, k_{max} \in S_p$ and any $k_s \in R_s$ will lead to a 'well balanced' split according to the criteria of the algorithm. From $R_s \cap S_p$ we then calculate for all adjacent (in an ordered representation) keys k, k' the longest shared prefix $k_x = k \wedge k'$ and call the set of all such prefixes S_x. Let now $S_{lx} = len(S_x)$, that is the image of S_x under the len function, be the length set of S_x. Finally, we split S_p such that k_s is any element of the preimage $len^{-1}(\min S_{lx})$.
This approach does not only influence the data-locality of a partitions, but the choice of partition ids also influences the size of the LUT, hence choosing shorter ids is preferred and should be attempted whenever possible.

3.8 Replication

While CaracalDB is not limited to a specific replication algorithm, it does provide some and also has some requirements on others' implementations. The algorithms that are part of CaracalDB are:

Async Replication is an algorithm that provides eventual consistency and is used in Dynamo [4]. This algorithm is very fast and highly available, but it requires the application layer to be able to reconcile conflicting versions of data in the system.

ABD has been described in section 2.2.2 and provides atomic consistency. It provides very high throughput as operations on different keys are completely independent. However, it supports only simple put and get operations.

Paxos SMR has been explained in 2.2.3 and also provides atomic consistency. It enforces a total order on all operations in a partition and allows for more powerful operations like CAS, for example. It sacrifices throughput compared to ABD but has basically the same latency in a stable state.

For custom algorithms it is required that they implement a reconfiguration protocol that includes decisions about when and how data transfer has to take place. CaracalDB provides a component that can handle data transfer and report its status on request, but replication algorithms need to decide when to initialise data transfer in relation to the time they get a reconfiguration message, and also how to handle normal operations during this syncing phase.

Furthermore, replication components are directly connected to a storage component and need to issue the required commands to it in response to operation requests that are routed to them. Whether or not these commands are issues before or after replication depends on the algorithm.

3.9 Storage

Just as replication, storage implementations are pluggable. However, they are not handled on a per-vnode basis, but on a per-host basis. The reason for this is the circumstance that some databases are optimised to minimise seeks on HDDs [41] and having one of those for each vnode would defeat that purpose. For the same reason one should be careful not to locate two different databases on the same HDD. For SSDs this problem is not as pronounced.
CaracalDB is designed with the following four storage implementations in mind, but really any engine that provides operations for put, get, iterate and batch write can be used.

InMemory provides non persistent, fast, but also very limited storage capabilities. Since the size of RAM is limited and the speed is so much superior to network, the actual implementation of this engine does not make a huge difference. It needs to be lexicographically sorted, though, ruling out hash maps, for example.

SILT is a SSD optimised store [36] and should be used if such a disk is available. It is probably the next best choice if RAM is not sufficient anymore.

bLSM is a fast engine based on log structured merge trees and bloom filters [41]. Due to its superior performance and avoidance of major write lag induced by the merge operation, bLSM is the preferred choice for HDDs. bLSM is also used as part of Walnut [13] for objects below 1MB.

LevelDB [42] is similar to bLSM and much more mature and in widespread use. CATS for example has a persistence layer using LevelDB [43]. However it performs worse, especially for object larger than 100kB and it also has the aforementioned problem of write pauses during merge operations.

3.10 Networking

CaracalDB is not bound to a specific networking protocol stack. It was designed with the transmission control protocol (TCP)/internet protocol suite (IP) stack in mind, but could technically be used with any stack that provides reliable, ordered channels.
There is, however, one very specific networking concern that was addressed during the conception of CaracalDB. To understand the problem, consider the second

re-replication example from section 3.3 again, where a large number of 100MB partitions are distributed over vnodes in the cluster, such that every host shares a replication group with (almost) every other host. In that section we argued that re-replication would greatly benefit from this scenario, because almost every disk in the cluster would be utilised. Of course, any network engineer will immediately notice that all we have achieved here is moving the bottleneck from disk speed to the core switch in a hierarchical network layout. While TCP's flow control mechanism will assure that this scenario still performs better than the original, it will not be as fast as the elaborations in section 3.3 might have initially suggested.
There is, however, another type of networking architecture under which this kind of re-replication can realise its full potential. Such a network is called a full bisection bandwidth CLOS network [44, 45, 10]. In a CLOS network, instead of a single core switch bottlenecking the whole system, we see a number of smaller commodity switches with redundant interconnections and some clever routing algorithms providing, if set up appropriately, full end to end bandwidth between any pair of nodes, making any considerations of network topology in storage systems redundant. While Nightingale et al. note that the hardware for such a cluster is about 30% more expensive [10] than a hierarchical cluster of similar size, we are convinced that datacenter deployments of CLOS networks are the future. Hence, our optimisation for such a networking architecture.

With full bisection bandwidth CLOS networks, however, theres a new bottleneck appearing. Consider again the same scenario from above, and note that the data transfers involved might not all be of equal size. Since TCP's flow control is reactive it might happen that temporarily more bandwidth is used between two specific nodes than between others. If these occurrences coincide between multiple hosts trying to stream data to a single target host it may happen that the network interface card (NIC) of the target host gets overloaded and starts to drop packages. This problem is known as *incast* [46]. To counter the incast issue, Nightingale et al. have implemented a request-to-send/clear-to-send (RTS/CTS) side flow control mechanism similar to what is used in wireless networks [10]. With this mechanism large data transfer messages are queued at the sender, and the receiver is notified with an RTS. The receiver provides a limited the number of 'virtual channels' for which it provides CTSs to one sender at a time, thus limiting the number of senders competing for its receive bandwidth and colliding at its switch port.
We use a similar approach in CaracalDB, but we even go one step further. Instead of simply queuing a potentially large number of messages at the sender, we forward the CTS messages to higher level components connected to the network layer. These messages define a number of messages (with a maximum size) the component is allowed to send on the assigned channel. The idea is, that only as much data is actually read up from the storage as the component has been cleared to send. The data transfer component provided by CaracalDB implements this pattern in an easy to use, black-box way.
An alternative to the RTS/CTS mechanism we considered is the low extra delay

background transport (LEDBAT) protocol [47]. LEDBAT provides low-priority flows that exhaust available network bandwidth, but yield to TCP's more aggressive algorithm. Since LEDBAT is specifically designed for background bulk transfer applications, it is a noteworthy alternative for our system. However, if this is applicable and which system provides better performance is subject to future research.

Chapter 4

Experimental Evaluation

In order to show that our basic approach in CaracalDB is practical and competitive performance-wise, we present a small experimental performance evaluation of an early state of the CaracalDB Java implementation. We report latencies and throughput for different workloads in a small cluster.

4.1 Implementation

CaracalDB is at the time of writing being developed at the Swedish Institute of Computer Science (SICS) in Stockholm, using the Java programming language and the Kompics [15] framework. As this is ongoing work, the experiments in this chapter were done on an early prototype that is strongly limited in the features it provides. The following lists the status of CaracalDB's features when the experimental results were gathered:

Bootstrapping Protocol was implemented, with the exception of handling nodes that joined or failed after LUT generation.

Churn Handling was not implemented at all.

Automatic Load Balancing was not implemented at all.

Replication The only algorithm for replication that was provided was a custom implementation of Paxos SMR and that might have had a few minor issues, as well.

Storage Only InMemory storage was available.

Lookup Routing and data assignment was fully implemented, however since the LUT was basically static, no LUT management was implemented. Furthermore, the LUT was still using the naïve layout for the partition to replication group mapping.

Client was only available in 'thin', hence all lookups were two-hop.

Schemas The implementation was schema-free and provided the whole key space $\mathcal{K}$ to write into (even the reserved range R_0).

Operations that were supported were simple put and get on a single key at a time.

4.1.1 YCSB

To apply the workload to the prototype system and collect the results we used the well known Yahoo! Cloud Serving Benchmark (YCSB) [48]. We combined the asynchronous CaracalDB thin client implementation with a blocking interface that was in turn used by the YCSB DB engine. However, we also had to make a small modification to YCSB's CoreWorkload generator, as the provided implementation prepended the string “user” to every key it created. Since CaracalDB does not hash keys and we didn't have support for schemas or load balancing, yet, this would have resulted in all requests ending up on the same partition. So instead of measuring the performance of the whole cluster, we would only have measured the performance a single partition. In order to avoid this problem we decided to simply remove this prefix.

4.2 Evaluation Setting

4.2.1 Hardware and Software

The experiments presented in the next section were run on SICS' internal cluster using seven homogenous servers in a rack. Each node had two 6-core AMD Opteron 2.5GHz processors and 32GB of RAM, as well as $Gbit/s$-ethernet connectivity. The machines were running a 64-bit Debian with Linux kernel 2.6.38 and we used Oracle's Java 64-BitHotSpot$^{\text{TM}}$ Server VM 1.6.0 for the experiments.
Of these seven nodes, six were used for running the CaracalDB storage system, and one was dedicated as a YCSB client running workloads with 24 threads in parallel.

4.2.2 Workloads

YCSB comes with a number of workloads predefined and allows users to customise things like read/write ratio, key distribution and of course number of keys and number of operations. We decided to use two similar workloads A and B. For both of them we first introduced 10000 keys into the storage, without measuring performance. Then each workload would run 100000 operations, with A having 50% updates and 50% reads, while B was doing only 5% updates and 95% reads. Both workloads used a uniform key distribution over an eight byte long key space. These keys were translated into string representations of the numbers by YCSB and those strings in turn were translated into a UTF-8 based byte sequence, forming a key in $\mathcal{K}$.

4.3 Performance

The results reported by YCSB after running the workloads include for each type (read or update) the exact number of operations executed, the average latency (μs), the 95th percentile latency (μs), the maximum and minimum latency (μs) and a latency distribution with ms steps. Additionally, for each experiment YCSB reported the overall throughput (op/s).
Since in absence of automatic load balancing the only parameter that allowed us to tune the load distribution was the number of partitions per host n_v created during LUT generation, we ran two series of experiments that were identical except for the value of n_v. The first case with $n_v = 1$ is basically equivalent to sl-replication in CATS [20, 21, 15, 19] and was meant to be a direct comparison to their results. Here there are six partitions in the whole system, all of which are triple replicated, such that every host runs 3 vnodes on average. For the second case we increased n_v to 20 in order to show the improvement of better load distribution. With this scenario there are 360 vnodes in the whole system, coming down to 60 per host.

4.3.1 The Felines

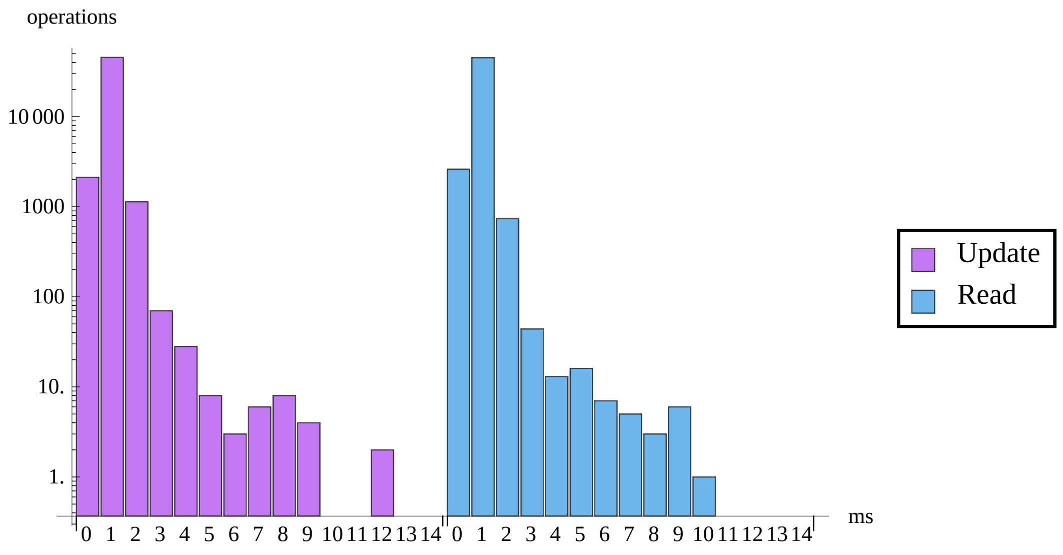

Figure 4.1. Operation latency distribution with $n_v = 1$ in workload A.

For workload A and $n_v = 1$ CaracalDB provided a throughput of a little over $800 op/s$. However, as can be seen in figure 4.1 the tail for the latency distribution of updates and reads in this scenario is very short, with barely any values over 10ms and a 95th percentile of 1ms for updates and reads alike and a minimum read latency of under $400\mu s$ while minimum write latency was just over $400\mu s$. This would usually suggest that the system is not saturated, yet. It should be noted,

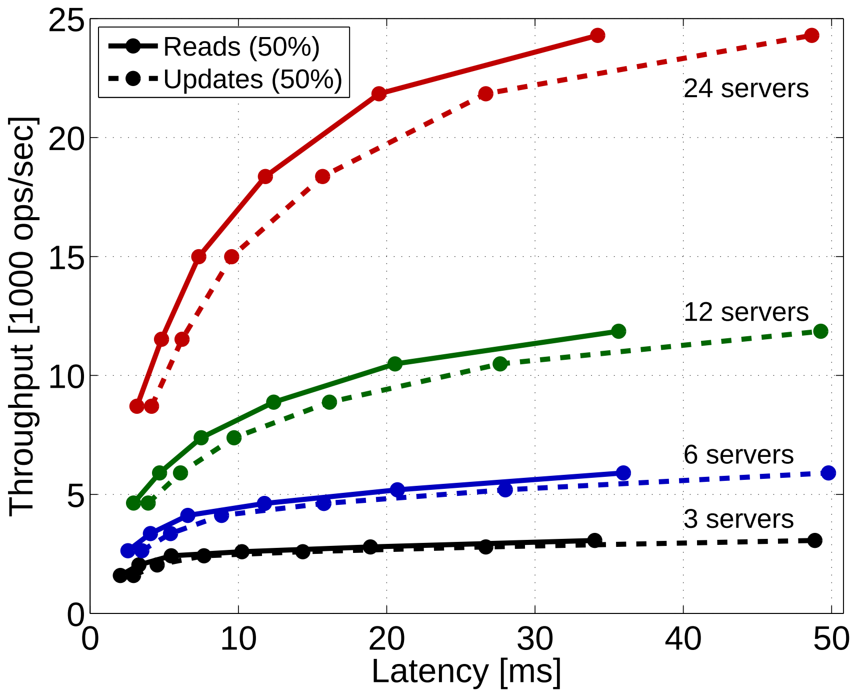

Figure 4.2. Operation performance in CATS with a workload similar to A.

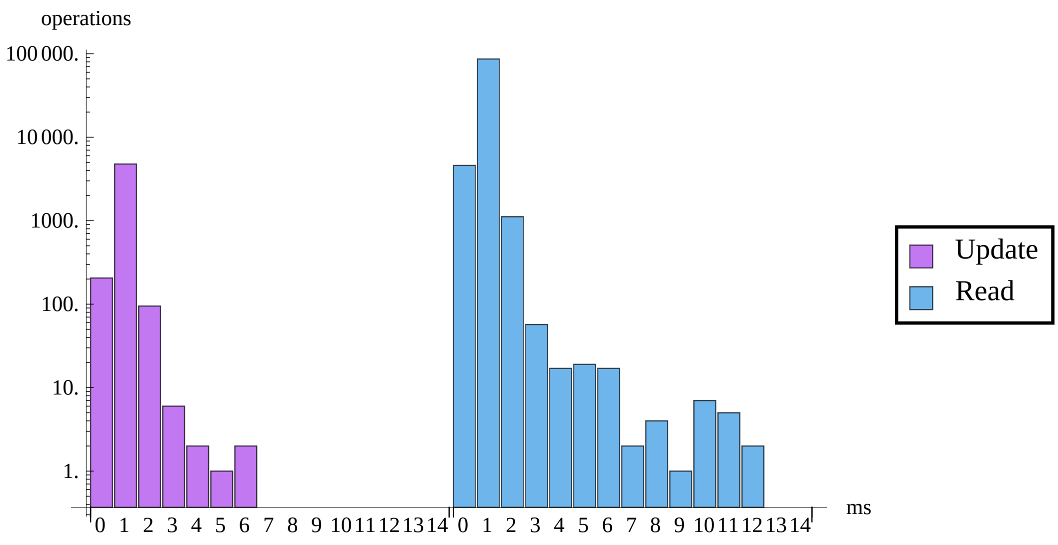

Figure 4.3. Operation latency distribution with $n_v = 1$ in workload B.

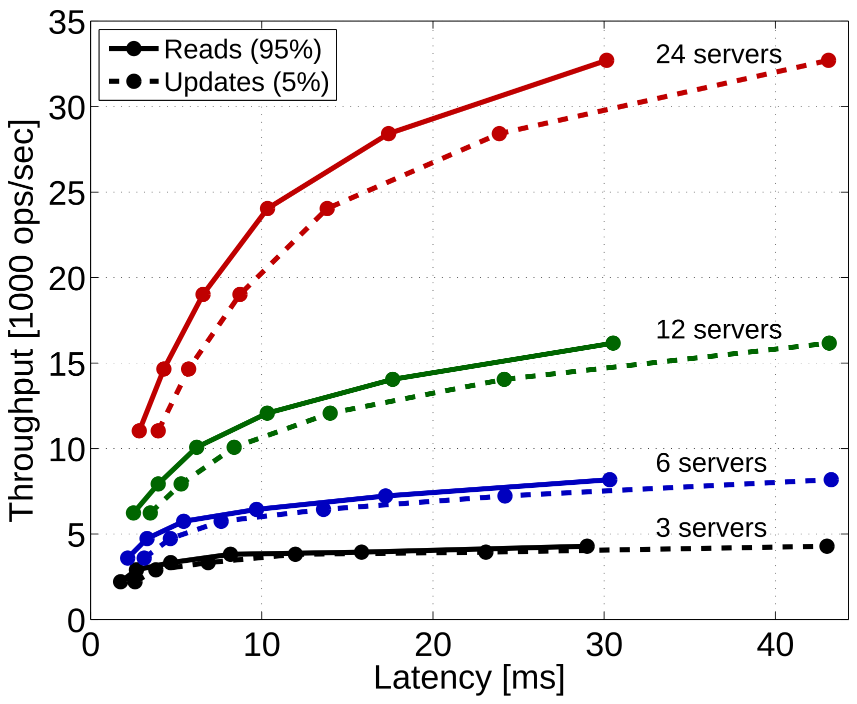

Figure 4.4. Operation performance in CATS with a workload similar to B.

though, that in addition to the shown values there was about 1% of operations that experienced a timeout, which was set to 1s for all scenarios, hence the average latency in the experiments is somewhat distorted and will not be reported. If the system were to be subjected to higher load by adding more threads to YCSB the number of timeouts increased dramatically, so 24 threads was found to be a good value for sustained load. If we compare the latency to that of CATS at a comparable load of around $1000op/s$ it can be extrapolated from figure 4.2 from the technical report [20] that their performance is quite similar. However, due to using ABD in favour of Paxos SMR CATS can sustain higher throughput, since it can perform operations on different keys in parallel. It should also be noted that the difference between read and update latency is negligible, owing to the fact that they are treated the same by the replication algorithm. This is also clearly expressed in the similarity of the results for A and B scenarios. B has basically the same throughput as A and the same 95^{th} percentile of 1ms for either operation. The only difference are in exact numbers of each operation, of course, which is somewhat striking in figure 4.3, and the fact that updates had a slightly higher minimum latency of almost $500\mu s$, not significant in any way. As CATS' ABD has the same property that reads and writes are treated almost the same, at least when it comes to number of network trips, figure [20] shows the same similarity in results for workload B or the read-heavy load as before for workload A.

4.3.2 Virtualisation

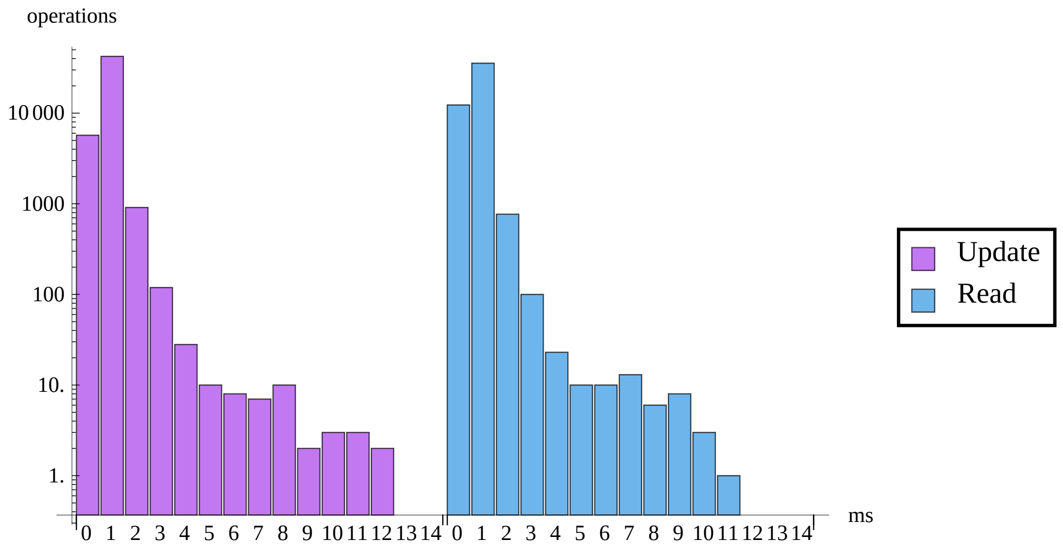

Figure 4.5. Operation latency distribution with $n_v = 20$ in workload A.

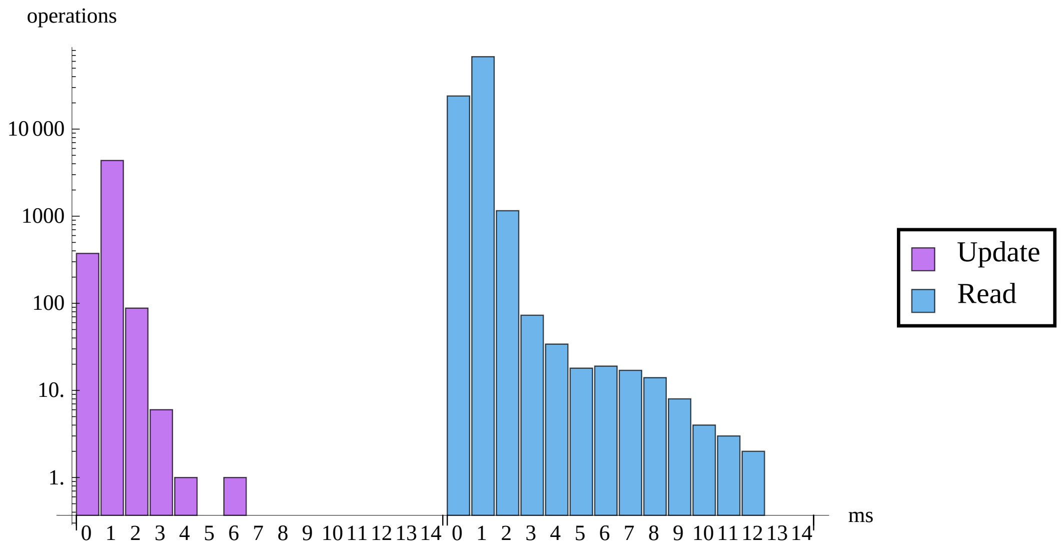

Figure 4.6. Operation latency distribution with $n_v = 20$ in workload B.

For the experiments with more vnodes and $n_v = 20$ some improvement can be seen. The throughput for both workloads has gone up to around $1000 op/s$, but all 95th percentiles have stayed at 1ms and the minimum latencies for all runs have also stayed between $300\mu s$ and 400μ. If we compare the latency distributions between figures 4.1 and 4.5 the latter shows about five times the number of update operations in the under-1ms-bar, which have mostly come out of the 1ms-bar. We observe the same trend comparing figures 4.3 and 4.6 on read operations. This indicates that there is a small but noticeable latency reduction in addition to the increased throughput. It is likely that most of these gains are not only due to improved load distribution, but also reflect the fact that there are more Paxos SMR instances running in the service that can decide operations independently in parallel, bringing the state closer to ABD in CATS.

Chapter 5

Conclusions and Future Work

Modern large-scale applications and data processing solutions depend heavily on the availability of scalable, reliable and performant distributed storage systems. There is a number of issues that need to be solved by any such system and among them the handling of skew in the distribution of data items and data access is one of the most prominent. We have presented the design of *CaracalDB*, a distributed key-value store tackling these issues and providing automatic load-balancing and data-locality, as well as fast re-replication after node failures. Yet, CaracalDB remains flexible enough to provide different replication algorithms and storage solutions, giving application developers their choice of consistency vs. performance trade-offs. Additionally, we have evaluated an early prototype of CaracalDB and shown that the approach is comparable with similar systems, specifically CATS [20, 21, 15, 19], in terms of latency, even when using a replication algorithm that supports more advanced features. In the experiments we paid for this with reduced system throughput, however.

5.1 Conclusions

We have shown that it is possible to design a very flexible system without sacrificing performance. If this is not clear, yet, consider the latencies presented in chapter 4 again. If there is little contention over 95% of operations complete in under 2ms. About 25% even under 1ms. And this with a replication algorithm that provides total ordering of operations per partition, allowing for example to provide CAS or increment/decrement semantics in addition to simple put and get operations. In an exemplary scenario where a small number of application server executes sequential operations, but tries to minimise overall latency of their runs, this is a perfect match. Should, however, a more parallel approach with more throughput be needed, CaracalDB can provide that just as well. Replacing Paxos SMR with ABD CaracalDB should be able to provide the same throughput as CATS, if not better. Should even more throughput be needed, but consistency be not much of an issue, asynchronous replication like in Dynamo [4] can likely provide. And when we consider custom algorithms that can be plugged in, the possibilities are literally endless.

5.2 Future Work

It goes without saying that the most important work still to be done on CaracalDB is the completion of a reference implementation and a full evaluation, including behaviour under churn, comparison of different replication algorithms and the results of different load balancing algorithms. However, there are a number of other ideas and alternatives that were considered during the conception of the presented system. Some of these are interesting enough to warrant additional research work.

5.2.1 Decentralisation

While technically there is no special snowflake node[1] in CaracalDB that should under no circumstances die, it is also not a truly decentralised system, relying on the replication group of a special partition 0_P and especially its leader to take decisions that affect the whole system. While this is unlikely to become a bottleneck in the targeted single-datacenter deployment, it makes CaracalDB unusable for really massive-scale multi-datacenter or open internet deployments. Hence, there will always be a desire for a decentralised solution with similar capabilities. One idea that was considered while designing the system, was using a ring of replication groups that perform periodic stabilisation with their neighbours as is done in Scatter [18] or ID-replication [19], for example. Since there would not be a single point to store system statistics in this setting, we considered using a gradient topology overlay [49] based on the current load of a host[2]. However, the choice of metric for the overlay makes it inherently very unstable, and the behaviour of the overall system particularly difficult to predict. Thus further research with such a system is needed to ascertain whether or not it is a viable model and under which conditions this is the case.

5.2.2 Anti-Entropy

Under churn it may happen, depending on the replication protocol, that replicas of partitions get out of sync. In order to counter this behaviour storage systems like Dynamo [4] and Cassandra [5] use so called Merkle [50] trees to form a multi-round protocol that allows replicas to locate and reconcile differences. If this will be necessary for CaracalDB remains to be seen, but the usage of Dynamo's replication protocol makes this a likely extension.

[1]Even BootstrapServer failures can be handled if there is a way for an administrator to figure out the address of any other live node in the system.

[2]Or multiple such overlays to deal with multiple load types.

5.2.3 Proactive Load Balancing

As was already alluded to in section 3.7 it may be desirable for storage system which runs for months or years without pause to *learn* repetitive behaviour and adapt the system proactively. To consider a powerful example that is a little over CaracalDB's current capabilities, take a cloud deployment the storage system, such that there exists an interface to the cloud API, giving a managing component of the system the ability to increase its capacity when this is deemed necessary. An outline of such a system, called *SLAstic*, in a very general sense, is presented in [51]. This is also called a *self-adaptive* system. If such a system is combined with a manager component that employs machine learning techniques in order to apply measures that it had to do reactively in the past in response to load imbalances or general system overload, proactively instead it could predict future load changes based on repeating load cycles or extrapolation of short term load slopes. Such a system could be very useful when a service provider aims at minimising its costs by providing only enough capacity to meet the current demand, but on the other hand has to adhere to strict service level agreements (SLAs).

5.2.4 Richer Semantics and the Object Model

There is currently ongoing research at SICS on adding a richer set of operations, like a subset of SQL, to CATS. This is conceptually similar to Impala [12], HIVE [11] or Pig Latin [52]. It is very desirable that this project be ported to CaracalDB, as the acceptance among enterprise Java developers for such higher level query languages appears to be much better than for the simpler put-get-API.

5.2.5 Higher-dimensional Key Spaces

During the somewhat digressive contemplations of CaracalDB's key space $\mathcal{K}$ in section 3.2 it occurred to us that if we could provide the necessary properties of a metric and a total ordering in a 1-dimensional, linear key space and this is sufficient for our system to work, then it should be possible to use an n-dimensional key space, if we can provide the same properties. There are already distributed storage systems that use a multi-dimensional key space in order to index multiple fields of an object [53]. This is especially interesting in combination with the ideas of section 5.2.4.

5.2.6 Distributed Transactions

Another important feature that would improve the system in section 5.2.4 tremendously is support for distributed, cross-partition transactions. Those are provided by systems like Google's Spanner [54], for example. Any system that implements fast, distributed transactions opens up multitude of new possibilities with respect to consistent operations on complex, structured objects.

5.2.7 Geo-replication

In order to deal with catastrophic datacenter-scale failure scenarios, or simply reducing end-to-end latency when dealing with globally distributed requests, it could be desirable or even necessary to provide support for geo-replication [55, 54] of partitions in CaracalDB.

5.2.8 Column-oriented Schemas

For OLAP focused systems column-oriented storage formats can vastly outperform row-oriented formats for certain operations [56, 57]. Since the interpretations of the data stored in a CaracalDB schema is up to the application and maybe the storage engine, it is conceivable that it should be possible to mix row-oriented and column-oriented schemas in the same CaracalDB cluster.

5.2.9 Full-system shutdown

Sometimes it might be necessary to shut down a whole storage cluster, without losing any data in the process, for example for maintenance work. With a pure InMemory store, this is obviously impossible. But if data is also persisted to disk, and especially if the host state can be persisted to disk, the full cluster could be shut down and rebooted into the same state it was before the shutdown. This requires additional protocols as well as state persistence for all algorithms that require it, though.
In addition to this, if CaracalDB were to support a fail-recovery model, instead of the current fail-stop mode, then some algorithms like Paxos would actually require their state to be persisted to disk on failures and reloaded during reboots.

5.2.10 The Big Picture

Finally, we come back to section 1.1, where we said that the overall goal should be a system that provides ACID low-latency access to small data, and (quasi-)unlimited storage capacity for big data as well as support for flexible processing solutions. CaracalDB represents a first step into this direction, by providing a very general framework, with flexible semantics and the option to add transaction support (s. section 5.2.6). While one path in this direction would be to use CaracalDB to implement a new, more scalable and more powerful HDFS name node, another more interesting path would be to design a new API for medium and big data storage, that is aimed at future usage. Either path is open with CaracalDB.

Bibliography

[1] G. M. Church, Y. Gao, and S. Kosuri, “Next-Generation Digital Information Storage in DNA,” *Science*, vol. 337, no. 6102, p. 1628, Sep. 2012. [Online]. Available: http://www.sciencemag.org/content/337/6102/1628.abstract

[2] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[3] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, “Data warehousing and analytics infrastructure at facebook,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD ’10. New York, NY, USA: ACM, 2010, pp. 1013–1020. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807278

[4] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *ACM SIGOPS . . .*, pp. 205–220, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1294281

[5] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[6] F. Chang, J. Dean, and S. Ghemawat, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on . . .*, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1365816

[7] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010, pp. 1–10.

[10] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat Datacenter Storage," *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 1–15, 2012. [Online]. Available: http://research.microsoft.com/pubs/170248/fds-final.pdf

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687553.1687609

[12] Cloudera, "Impala," 2012. [Online]. Available: https://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/

[13] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: a unified cloud object store," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, Conference proceedings (article), pp. 743–754. [Online]. Available: http://portal.acm.org/citation.cfm?id=2213947

[14] B. Cooper and R. Ramakrishnan, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1454167

[15] C. I. Arad, "Programming Model and Protocols for Reconfigurable Distributed Systems," Ph.D. dissertation, KTH - Royal Institute of Technology, Stockholm, 2013.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[17] I. Stoica and R. Morris, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *Networking, IEEE/ . . .*, vol. 11, no. 1, pp. 17–32, Feb. 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1180543http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1180543

[18] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in Scatter," in *Proceedings of the Twenty-Third ACM*

Symposium on Operating Systems Principles, ser. SOSP '11. Cascais, Portugal: ACM, 2011, Conference proceedings (article), pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043559

[19] T. M. Shafaat, "Partition Tolerance and Data Consistency in Structured Overlay Networks," Ph.D. dissertation, KTH - Royal Institute of Technology, Stockholm, 2013.

[20] C. Arad, T. M. Shafaat, and S. Haridi, "CATS: Linearizability and Partition Tolerance in Scalable and Self-Organizing Key-Value Stores," Swedish Institute of Computer Science (SICS), Stockholm, Tech. Rep., 2012.

[21] C. Arad, T. Shafaat, and S. Haridi, "Brief Announcement: Atomic Consistency and Partition Tolerance in Scalable Key-Value Stores," in *Distributed Computing SE - 50*, ser. Lecture Notes in Computer Science, M. Aguilera, Ed. Springer Berlin Heidelberg, 2012, vol. 7611, pp. 445–446. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33651-5_50

[22] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/200836.200869

[23] S. H. Afzali, "Consistent Range-Queries in Distributed Key-Value Stores," M.S. thesis, KTH - Royal Institute of Technology, Stockholm, 2012.

[24] M. E. ul Haque, "Persistence and Node Failure Recovery in Strongly Consistent Key-Value Datastore," M.S. thesis, KTH - Royal Institute of Technology, 2012.

[25] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[26] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *Computers, IEEE Transactions on*, vol. C-28, no. 9, pp. 690–691, 1979.

[27] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[28] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the seventh ACM symposium on Operating systems principles*, ser. SOSP '79. New York, NY, USA: ACM, 1979, pp. 150–162. [Online]. Available: http://doi.acm.org/10.1145/800215.806583

[29] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, ser. FTCS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 272—-. [Online]. Available: http://dl.acm.org/citation.cfm?id=795670.796859

[30] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[31] ——, “Fast Paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006. [Online]. Available: http://dx.doi.org/10.1007/s00446-006-0005-x

[32] X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1041680.1041682

[33] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, “Megastore: Providing Scalable, Highly Available Storage for Interactive Services,” in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234. [Online]. Available: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[34] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica, “Load balancing in dynamic structured peer-to-peer systems,” *Perform. Eval.*, vol. 63, no. 3, pp. 217–240, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2005.01.003

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: fair allocation of multiple resource types,” in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011, p. 24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972490http://static.usenix.org/events/nsdi11/tech/full_papers/Ghodsi.pdf

[36] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “SILT,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP ’11*. New York, New York, USA: ACM Press, 2011, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2043556.2043558

[37] D. Liben-Nowell, H. Balakrishnan, and D. Karger, “Analysis of the evolution of peer-to-peer systems,” in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, ser. PODC ’02. New York, NY, USA: ACM, 2002, pp. 233–242. [Online]. Available: http://doi.acm.org/10.1145/571825.571863

[38] B. Cohen, “Incentives Build Robustness in BitTorrent,” 2003.

[39] A. Metwally, D. Agrawal, and A. E. Abbadi, “Efficient Computation of Frequent and Top-k Elements in Data Streams,” in *IN ICDT*, 2005, pp. 398–412.

[40] J. a. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "AUTOPLACER : Scalable Self-Tuning Data Placement in Distributed Key-value Stores," in *Proceedings of the 10th International Conference on Autonomic Computing*, 2013.

[41] R. Sears and R. Ramakrishnan, "bLSM," in *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*. New York, New York, USA: ACM Press, 2012, p. 217. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2213836.2213862

[42] Google, "LevelDB." [Online]. Available: https://code.google.com/p/leveldb/

[43] M. E. ul Haque, "Persistence and Node Failure Recovery in Strongly Consistent Key-Value Datastore," Ph.D. dissertation, KTH Royal Institute of Technology, Stockholm, 2012.

[44] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a next generation data center architecture: scalability and commoditization," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, ser. PRESTO '08. New York, NY, USA: ACM, 2008, pp. 57–62. [Online]. Available: http://doi.acm.org/10.1145/1397718.1397732

[45] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, ser. SIGCOMM '09. New York, NY, USA: ACM, 2009, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/1592568.1592576

[46] V. Vasudevan, H. Shah, A. Phanishayee, E. Krevat, D. G. Andersen, and G. A. Gibson, "Solving TCP Incast in Cluster Storage Systems." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.158.4195

[47] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)," 2012. [Online]. Available: https://datatracker.ietf.org/doc/rfc6817/

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[49] J. Sacha, "Exploiting heterogeneity in peer-to-peer systems using gradient topologies," 2009.

[50] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *A Conference on the Theory and Applications of Cryptographic*

Techniques on Advances in Cryptology, ser. CRYPTO '87. London, UK, UK: Springer-Verlag, 1988, pp. 369–378. [Online]. Available: http://dl.acm.org/citation.cfm?id=646752.704751

[51] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring, "An adaptation framework enabling resource-efficient operation of software systems," in *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, ser. WUP '09. New York, NY, USA: ACM, 2009, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1527033.1527047

[52] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1099–1110. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376726http://dl.acm.org/citation.cfm?id=1376726

[53] V. March and Y. M. Teo, "Multi-attribute range queries in read-only DHT," in *in 'Proc. of the 15th IEEE Conf. on Computer Communications and Networks (ICCCN 2006*, 2005, pp. 419–424.

[54] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner : Google ' s Globally-Distributed Database," in *OSDI 2012*, 2012, pp. 1–14.

[55] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 385–400. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043592

[56] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," in *Proceedings of the 31st international conference on Very large data bases*, ser. VLDB '05. VLDB Endowment, 2005, pp. 553–564. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083658

[57] Vertica Systems and Hewlett Packard, "Vertica," 2005.

Acronyms

ACID atomicity, consistency, isolation and durability properties. 3, 42

API application programming interface. 8, 13, 41, 42

CAP consistency, availability and partition tolerance. 2

CAS compare-and-swap. 13, 27, 39

DHT distributed hash table. 4, 14

DNA deoxyribonucleic acid. 1

DSL domain specific language. 5

FDS Flat Datacenter Storage. 24

GFS Google file system. 2

HDD hard disk drive. 20, 28

HDFS Hadoop distributed file system. 2, 3, 42

IP internet protocol suite. 28

LBG load balancing goal. 10, 11

LEDBAT low extra delay background transport. 29, 30

LUT global lookup table. 19–27, 31, 33

NIC network interface card. 29

OLAP online analytical processing. 2, 42

OLTP online transaction processing. 2

P2P peer-to-peer. 23, 25

RDBMS relational database management system. 2

RSM replicated state machines. 9

RTS/CTS request-to-send/clear-to-send. 29

S3 simple storage service. 2

SICS Swedish Institute of Computer Science. 31, 32, 41

sl-replication successor list replication. 5, 33

SLA service level agreement. 41

SMR state machine replication. 9, 13, 21, 24, 27, 31, 35, 37, 39

SSD solid state disk. 20, 28

TCP transmission control protocol. 28–30

vnode virtual node. 4, 16–22, 24–26, 28, 29, 33, 36

YCSB Yahoo! Cloud Serving Benchmark. 32, 33, 35