Parqua: Online Reconfigurations in Virtual Ring-Based NoSQL Systems

Yosub Shin, Mainak Ghosh, Indranil Gupta
Department of Computer Science
University of Illinois, Urbana-Champaign
{shin14, mghosh4, indy}@illinois.edu

***Abstract*—The performance of key-value/NoSQL storage systems is highly tied to the choice of (primary) key for the database table. As application (e.g., business) requirements change over time, and in order to fine-tune the performance of the database to the real query workload, system administrators need to change the primary key of the table. The primary key change is a specific example of a broader class of reconfiguration operations that affect a lot of data all at once. In industry deployments of key-value/NoSQL stores, such reconfigurations are known to be a major pain point.**

We seek to support reconfiguration operations in key-value/NoSQL storage systems in an automated, online, and efficient manner, i.e., without interrupting the serving of incoming reads and writes, and quickly. Our previous work, titled Morphus, tackled the online reconfiguration problem for sharded NoSQL stores like MongoDB. However, Morphus is inapplicable to ring-based key-value/NoSQL systems (like Cassandra, Riak, and Voldemort) because these rely on a virtual ring (and often consistent hashing). This makes the problem more constrained.

In this paper we propose a system called Parqua, which imbues ring-based key-value/NoSQL stores with the ability to perform reconfiguration operations in an online and efficient manner. We present the design and implementation of Parqua. We have integrated Parqua into Apache Cassandra. Experiments based on our cluster deployments show that during reconfiguration Parqua maintains high availability, and with a small impact on read and write latencies.

1. Introduction

Key-value/NoSQL systems today fall into two categories: 1) (virtual) ring-based and 2) sharded databases. The key-value/NoSQL revolution started with ring-based databases. The Dynamo system [3] from Amazon. Dynamo, and subsequent open-source variants of it including Facebook's Apache Cassandra [22], Basho's Riak [6], and LinkedIn's Voldemort [8] all rely on the use of a "virtual ring" to place servers as well as keys; keys are assigned to servers whose segment they fall into. For fault-tolerance, a key and its values are replicated at some of the successor servers as well.

Unlike the ring-based databases, sharded databases like MongoDB [5], BigTable [13], etc., rely on a fully flexible assignment of shards (sometimes called chunks or blocks) across servers, along with some degree of replication. Both the ring-based and sharded NoSQL databases have grown very quickly in popularity over the past few years, and are expected to become a $3.4 billion market by 2020 [10].

In these databases, performing reconfiguration operations seamlessly is a major pain point. Such operations include changing the primary key or changing the structure of the ring (e.g., where servers and keys are hashed to) – essentially such operations have the potential to affect all of the data inside the table. Today's "state of the art" approach involves exporting and then shutting down the entire database, making the configuration change, and then re-importing the data. During this time the data is completely unavailable for reads and writes. This can be prohibitively expensive – for instance, anecdotes suggest that every second of outage costs $1.1K at Amazon and $1.6K at Google [30].

The reconfiguration operation itself, though not as frequent as reads and writes, is in fact considered a critical need by system administrators. When a database is initially created, the admin may play around with multiple prospective primary keys in order to measure the impact on performance, and select the best key. Later, as the workload or business requirement changes, such reconfiguration operations may become less frequent but their impacts (on availability) are significant, because they are being made on a live database. Thus the need is for a system that allows administrators to perform reconfigurations anytime, automatically, and seamlessly, i.e., completely in the background, without affecting the serving of reads and writes.

The lack of an efficient online reconfiguration operation has led to outages at Foursquare [7], JIRA (bug tracking) issues that are hotly debated [2], and many blogs [25], [28]. The manual approach to resharding took over two years at Google [16].

In our past work [20], we have solved the problem of online reconfiguration for *sharded* NoSQL databases such as MongoDB. That system, called Morphus, leveraged the full flexibility of being able to assign any shards to any server, in order to derive an optimal allocation of shards to servers.

This work was supported in part by the following grants: NSF CNS 1319527, NSF CNS 1409416, NSF CCF 0964471, and AFOSR/AFRL FA8750-11-2-0084.

The optimal allocation was based on maximal matching, which both minimized the network traffic and ensured load balancing.

Unfortunately, the techniques of Morphus cannot be extended to *ring-based* key-value/NoSQL stores like Cassandra, Riak, Dynamo, and Voldemort. This is due to two reasons. First, since ring-based systems place data strictly in a deterministic fashion around the ring (e.g, using *consistent hashing*), this constrains which keys can be placed where. Thus, our optimal (maximal matching-based) placement strategies from Morphus no longer apply to ring-based systems. Second, unlike in sharded systems (like MongoDB), ring-based systems do not allow isolating a set of servers for reconfiguration (a fact that Morphus leveraged). In sharded databases each participating server exclusively owns a range of data (as master or slave). In ring-based stores, however, ranges of keys overlap across multiple servers in a chained manner (because a node and its successors on the ring are replicas), and this makes full isolation impossible.

This motivates us to build a new reconfiguration system oriented towards ring-based key-value/NoSQL stores. Our system, named Parqua [1], enables online and efficient reconfigurations in virtual ring-based key-value/NoSQL systems. Parqua suffers no overhead when the system is not undergoing reconfiguration. During reconfiguration, Parqua minimizes the impact on read and write latency, by performing reconfiguration in the background while responding to reads and writes in the foreground. It keeps the availability of data high during the reconfiguration, and migrates to the new reconfiguration at an atomic switch point. Parqua is fault-tolerant and its performance improves with the cluster size.

We have integrated Parqua into Apache Cassandra. Our experiments show that Parqua provides high nines of availability with little impact on read and write latency. The system scales well with data and cluster size.

2. System Model & Background

In this section, we present the system model and background for Parqua system. We demonstrate assumptions about the underlying distributed key-value store in order to implement Parqua. Then, we provide background information on Apache Cassandra.

2.1. System Model

Parqua is applicable to any key-value/NoSQL store that satisfies the following assumptions. First, we assume that a distributed key-value store is fully decentralized without the notion of a single master node or replica. Second, each node in the cluster must be able to deterministically decide the destination of the entries that are being moved due to the reconfiguration. This is necessary because there is no notion of the master in a fully decentralized distributed key-value store, and for each entry all replicas should be preserved after the reconfiguration is finished. In our implementation of Parqua on Apache Cassandra, we use consistent hashing [26] for determining the destination of an entry, but we could alternatively use any other partitioning strategies that satisfy our assumption. Third, we require the key-value store to utilize *SSTable (Sorted String Table)* to persist the entries permanently. An SSTable is essentially an immutable sorted list of entries stored on disk [13]. We utilize SSTables in Parqua for efficient recovery of entries in the Recovery phase. Next, we assume each write operation accompanies a timestamp or a version number which can be used to resolve a conflict. Finally, we assume the operations issued are idempotent. Therefore, supported operations are insert, update, and read operations, and non-idempotent operations such as counter incrementation are not supported.

1. The Korean word for "change."

2.2. Cassandra Background

We incorporated our design in Apache Cassandra, which is a popular ring-based distributed key-value store [23]. Cassandra borrows the architecture designs heavily from Distributed Hash Tables (DHTs) such as Chord [26].

Machines in Cassandra (henceforth called nodes) are organized logically in a ring, without involving a central master. Nodes may be either hashed to a ring or assigned uniformly within the ring. In the Cassandra data model, each row is uniquely identified with a *primary key*. The primary key of each entry is hashed onto the ring, and whichever node's segment it falls into, stores that key/value. A node's segment is defined as the portion of the ring between the node and its predecessor node. Some of the successors of that node may also replicate the key-value for fault-tolerance.

A read or write request can go from a client to any node on the ring, and the contacted node is called a *coordinator*. The coordinator routes the requests to the correct node(s) by hashing the primary key used in the query. Cassandra serves writes by appending a log to a disk-based *commit log*, and adding an entry to an in-memory dictionary data structure called *Memtable*. When a Memtable's size exceeds certain threshold, the Memtable is flushed to disk. This on-disk file is called a SSTable, and it is immutable. When a read request is issued and routed to the correct node, the node goes through the Memtable and possibly multiple SSTables that store the requested primary key's value. If there are multiple instances of the same column's value in the aggregated entry, the value with higher timestamp is chosen.

A single database table is called a *column family*. A database, also called a *keyspace*, contains multiple column families. Each column family has its own primary key. Cassandra supports adjustable consistency levels where a client can specify for each query the minimum number of replicas it needs to touch – popular consistency levels include ONE, QUORUM, and ALL.

3. System Design and Implementation

This section describes the design of our Parqua system, intended to support online and automated reconfigurations in any ring-based key-value store. For concreteness, we have integrated Parqua into Apache Cassandra. Below, we first give the overview of Parqua system. Then we describe the design of our system and its individual phases during reconfiguration.

3.1. Parqua: Overview

Parqua runs reconfiguration in four phases. The graphical overview of Parqua phases is shown in Fig. 1. When the reconfiguration is initiated, Parqua starts first with the *Isolate phase*, where it creates a new reconfigured database table (column family in Cassandra) with the desired new configuration that will supersede the original database table. Second, in the *Execute phase*, Parqua copies entries from the original database table to the reconfigured database table. Third, once entries are copied to the reconfigured database table, the *Commit phase* updates the two database tables by atomically swapping their SSTables and schemas. Finally, Parqua executes the *Recovery phase* which applies missing updates that were not copied in the Execute phase.

Parqua can support any reconfiguration operation that involves a large amount of data movement among nodes. In this work, our implementation of Parqua addresses primary key changes in Cassandra, where a primary key is composed of a single partition key column.

Next, we discuss these individual phases in detail.

3.2. Reconfiguration Phases in Parqua

Isolate phase: In this phase, the initiator node – the node in which the reconfiguration command is run – creates a new (and empty) column family (database table), denoted as *Reconfigured CF (column family)*, using a schema derived from the Original CF except it uses the desired key as the new primary key. The Reconfigured CF enables reconfiguration to happen in the background while the Original CF continues to serve reads and writes using the old reconfiguration. We also record the timestamp of the last operation before the Reconfigured CF is created so that all operations which arrive while the Execute phase is running, can be applied later in the Recovery phase. We disable automatic compaction in this phase in order to prevent disk I/O overhead during reconfiguration and to avoid copying unnecessary entries in the Recovery phase (later, our experimental results will explore the impact of doing compaction).

Execute phase: The initiator node notifies all other nodes to start copying data from the Original CF to the Reconfigured CF. Each node can execute this migration in parallel. Read and write requests from clients continue to be served normally during this phase.

At each node, Parqua iterates through all entries that it is responsible for, and sends them to the appropriate new destination nodes. The destination node for an entry is determined by: 1) hashing the new primary key value on the hash ring, and 2) using the *replica number* associated with the entry. Key-value pairs are transferred between corresponding nodes that have matching replica numbers in the old configuration and the new configuration. For example, in the Execute phase of Fig. 1, the entry with the old primary key '1' and the new primary key '10' have replica number of 1 at node A, 2 at B, and 3 at C. In this example, after primary key is changed, the new position of the entry on the ring is between node C and D, where node D, E, and F are replica numbers 1, 2, and 3, respectively. Thus, in the Execute phase, the said entry in node A is sent to node D, and similarly the entry in B is sent to E, and from C to F.

Commit phase: After the Execute phase, the Reconfigured CF has the new configuration and the entries from Original CF have been copied to Reconfigured CF. Now, Parqua atomically swaps both the schema and the SSTables between the Original CF and the Reconfigured CF. The write requests are locked in this phase while reads still continue to be served. In our implementation, we drop the write requests, in order to prevent any successfully returned writes from being lost during this phase. Reads are served from the Original CF before column families are swapped, and from Reconfigured CF after they are swapped.

The schema is updated for both column families by modifying the "system" keyspace – a special keyspace in Cassandra that stores metadata of the cluster such as schema information – with the appropriate primary key. For SSTable swap, first, the Memtables are flushed to disk. This is because the recent updates might be still residing in the Memtables. To implement the actual swap, we leverage the fact that SSTables are maintained as files on disk, stored in a directory named after the column family. Therefore, we move SSTable files from one directory to another. This does not cause disk I/O as we only update the *inodes* when moving files. Note that we do not simply drop the Original CF, but swap it with the Reconfigured CF. This is because the write requests that were issued since the reconfiguration has started are stored in the Original CF and need to be copied to the Reconfigured CF.

At the end of the Commit phase, the write lock is released at each node. At this point, all client facing requests are processed according to the new configuration. In our case, the new primary key is now in effect, and the read requests must use the new primary key.

Recovery phase: During this phase, the system catches up with the recent writes that are not transferred to Reconfigured CF in the Execute phase. Read/write requests are processed normally. The difference is that until the recovery is done, the read requests may return stale results. [2] Once SSTables are swapped in the Commit phase, the updated entries which need to be replayed are in the Original CF. The initiator notifies nodes to start the Recovery phase.

At each node, Parqua iterates through the SSTables of Original CF to recover the entries that were written during the reconfiguration. The SSTable is an immutable

2. This is acceptable as Cassandra only guarantees eventual consistency.

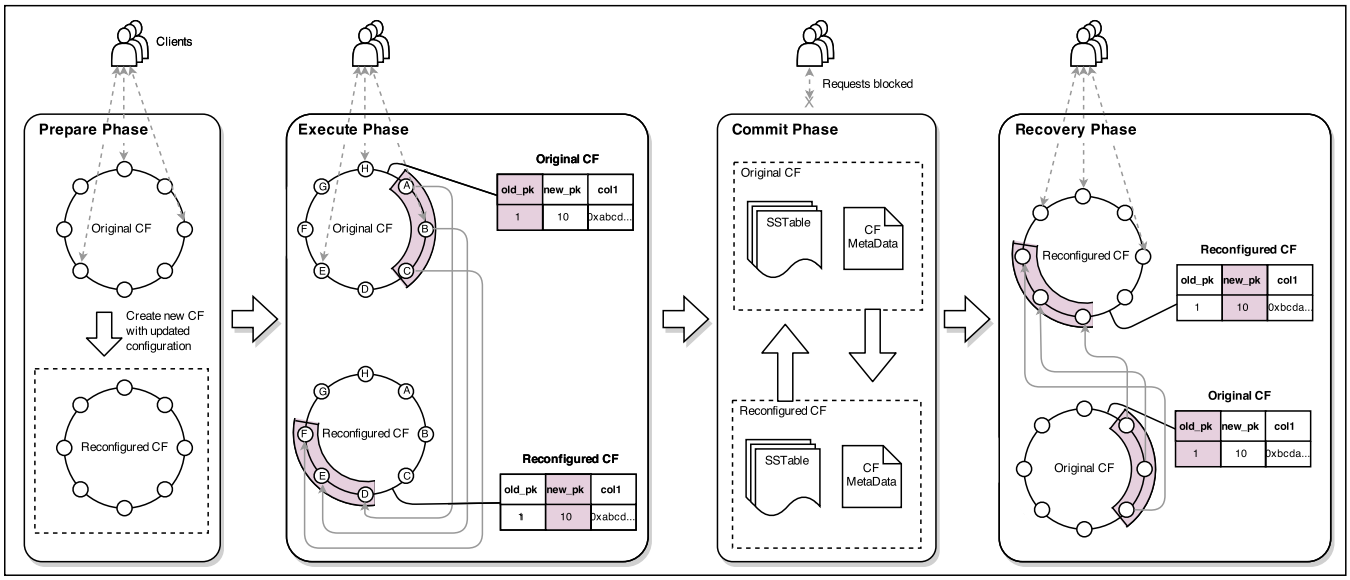


Figure 1: **Overview of Parqua phases. The gray solid lines represent internal entry transfers, and the gray dashed lines mean client requests. The phases progress from left to right.**

data structure such that a SSTable created at time t can only store updates written prior to time t. We leverage this fact to limit the amount of disk accesses required for recovery by only iterating the SSTables that are created after the reconfiguration has started. The iterated entries are routed to appropriate destinations in the same way as the Execute phase.

Since all writes in Cassandra carry a timestamp [1], Parqua can ensure that the recovery of an entry does not overshadow newer updates, thus guaranteeing the eventual consistency. For example, if an entry is updated at time t_1 during the Execute phase and again at t_2 in the Recovery phase where $t_1 < t_2$, the update originally issued at t_1 recovers at t_3 where $t_2 < t_3$. In this case, once the reconfiguration is over, a read request on this entry would return the correct result with the timestamp t_2, because Cassandra aggregates SSTables favoring the value with the highest timestamp.

3.3. Fault Tolerance of Parqua

Parqua can tolerate the failure of non-initiator nodes on the condition that there are enough replicas and appropriate consistency levels. If a non-initiator node fails in any of the phases, the Parqua guarantees the same fault tolerance model as the underlying distributed key-value store.

In a ring-based distributed key-value store, the key-value store is available upon failure of upto k nodes, if the consistency level of requests is less than or equal to $N-k$, where N is the replication factor. This is because there is always at least (consistency level) number of replicas available for any entry. Otherwise, the key-value store can still be recovered if $k \leq (N-1)$, since in this case there is at least one unfailed replica for all entries.

For instance, if a non-initiator node fails during the Execute phase, the entries that were stored in the failed node are not transferred to the new destination nodes. However, if the replication factor is 3 and the consistency level is 1 for read/write requests, there are still at least two replicas available for the any entries – including the entries from the crashed node – and the reconfiguration can continue without interrupted. This is the same guarantee offered by the underlying distributed key-value store.

4. Experimental Evaluation

In our experiments, we would like to answer the following questions:

- Does Parqua perform robustly under different workload patterns?
- How much does Parqua affect normal read and write operations of Cassandra, especially during the reconfiguration?
- Is Parqua scalable in terms of size of the cluster, database size, and the operation injection rate?

4.1. Setup

Data Set We used the Yahoo! Cloud Service Benchmark(YCSB) [15] to generate a dataset and workload. Each entry (key, values pair) has 10 columns with each column's size being 100 bytes, and an additional column that serves as the primary key. In all experiments, our default database size is 10 GB in all experiments.

Cluster The default Parqua cluster used 9 machines running 64 bit Ubuntu 12.04. We used Emulab cluster's d710 machines [4]. Each d710 machine has a 2.4 GHz quad-core

processor, 12 GB memory, 2 hard disks of capacities 250 GB and 500 GB, and 6 Gigabit Ethernet NICs.

Workload Generator We used YCSB as our workload generator. Our operations consist of 40% reads, 40% updates, and 20% inserts. With YCSB, we used the key access patterns of 'uniform', 'zipfian', and 'latest' – these model the pattern in which queries are distributed (uniform) or clustered across keys (zipfian) as well as time (latest). For zipfian distribution, the zipfian parameter of 1.50 was used. The default operation injection rate for our experiments was 100 operations per second, and the default key access pattern used was the uniform distribution.

We made a few minor modifications to YCSB in order to perform our experiments. First, to measure latency distributions accurately, we changed the granularity of the latencies histogram to 0.1 ms instead of the default setting of 1 ms. Second, we modified YCSB so that only one reconfiguration was executed at a time.

Cassandra Parqua is integrated into Apache Cassandra version 2.0.8. The Parqua system was written in Java and has about 2000 lines. We used the *simple replication strategy* with the default replication factor of 3, and the *Murmur3 hash-based partitioner* [23]. We also enabled the *virtual nodes* in which each peer is assigned 256 tokens, and used the *size-tiered compaction strategy*. In order to offer strong consistency under read/write operations, we used the write consistency level of ALL and the read consistency level of ONE.

Parqua In our reconfiguration experiments, we set the Parqua system to change the old partition key from *y_id* to the new partition key *field0*. Each plotted data point is an average of at least 3 trials, and is shown with standard deviation error bars.

4.2. Reconfiguration Time

In this experiment, we measured the time taken to complete the reconfiguration and the availability of queries during the reconfiguration for different workload distributions. Fig. 2 shows the contribution of the three major phases to the overall reconfiguration time. We observe that the Execute phase dominates for all workload types. This is expected due to the large volume of data being migrated during this phase. The duration of the Commit phase stays constant across different workloads (0.8 % - 1 %) – Parqua is able to obtain this advantage because it decouples data migration from the data querying.

In the Recovery phase, the read only workload finishes this phase very quickly because there are no entries to catch up in the Recovery phase. Also, in the same phase, we observe that uniform workload takes almost four times longer than zipfian and latest workloads. This is because uniform workload spreads writes over keys evenly, thus having more unique entries that need to be sent to different peers in the Recovery phase. However, since the overall reconfiguration time is dominated by the Execute phase (consisting 88 % - 98 % of overall reconfiguration time), this has a small effect on the overall reconfiguration time.

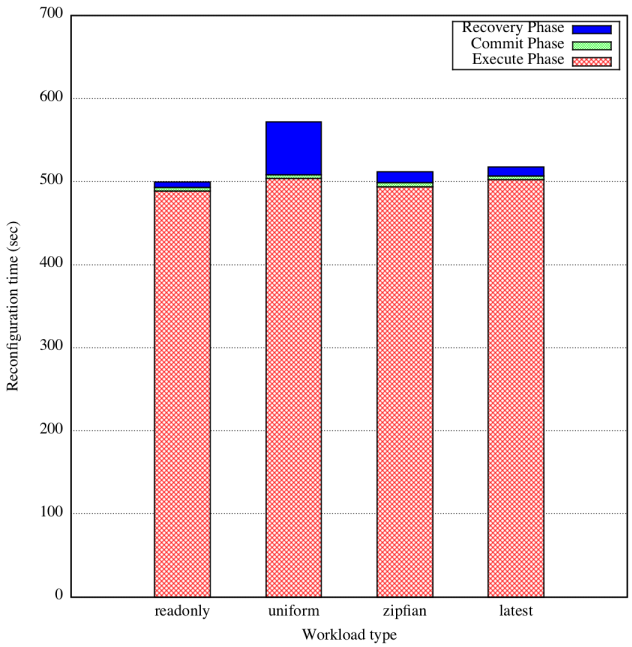


Figure 2: **Time taken for reconfiguration for different workload types and breakdown by individual phases.**

The overall reconfiguration time is small considering that the size of unique entries is 10 GB. Compared to our past work on Morphus [20], Parqua's reconfiguration time is 10 times faster (10 % of Morphus). This improvement is largely due to our design decision to migrate all replicas concurrently.

From this experiment, we conclude that Parqua offers predictable reconfiguration time under different workload patterns.

	Read (%)	Write (%)
Read only	99.17	-
Uniform	99.27	99.01
Latest	96.07	98.92
Zipfian	99.02	98.92
No reconfig (Uniform)	100.00	100.00
No reconfig (Latest)	99.52	100.00

TABLE 1: **Percentage of reads and writes that succeed during reconfiguration.**

4.3. Availability

Next, we measure the availability of Parqua system during the reconfiguration. In our system, the point at which the primary key actually changes is at the end of the Commit phase. Starting from this point onwards, any queries that were using the only the old primary key need to also include the new primary key. Therefore, we calculate the overall availability of our system by combining the availabilities of the system for queries with old primary key before the

Commit phase is finished, and the availability for queries with the new primary key after the Commit phase is finished.

In Table 1 we observe that the read and write availabilities for read only, uniform, and zipfian workloads are in the range of 99.02–99.27 % and 98.92 - 99.01 %, respectively. We point out that this slight degradation of the availability is far more preferable than the current solution of shutting down the database during the reconfiguration. We will explore this issue further in Section 4.4.

The lowest availability is the read availability of the latest workload at 96.07 %. This is because the multi-threaded YCSB workload generator assumes the most recent insert queries are already committed even before receiving the successful responses. Therefore, YCSB frequently tries and fails to read entries that are not yet inserted, causing degraded availability. This behavior is inherent to the architecture of YCSB, rather than Parqua. We can see a similar degradation of read availability under no reconfiguration for the latest workload. The reason for larger degradation under reconfiguration is due to increased average latency during reconfiguration as depicted in Section 4.4.

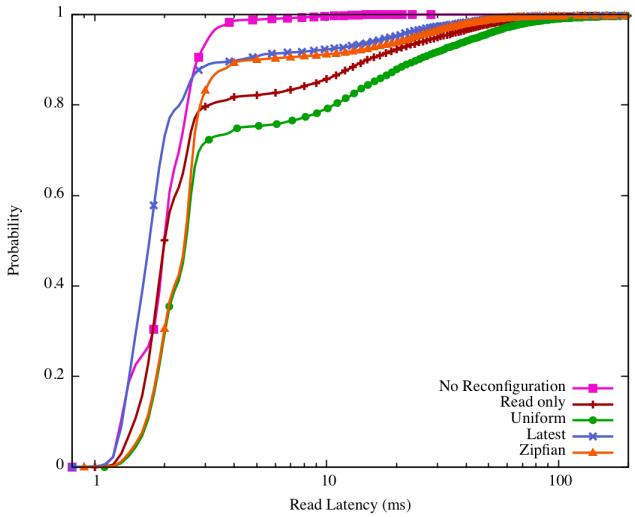


Figure 3: **CDF of read latencies for different workloads under reconfiguration, and the CDF of read latency under no reconfiguration for baseline measurement. The x-axis is read latency in logarithmic scale, while the y-axis is the cumulative probability.**

4.4. Read Latency

We now further explore the distributions of read latency during reconfiguration under Parqua.

4.4.1. Read Latency Over Time. In this section, we investigate the read latency characteristics for different workloads, during reconfiguration.

First, Fig. 4 shows the read latencies over time for four different workloads during the reconfiguration of Cassandra using Parqua. Fig. 4a shows the read latencies when no update/insert queries were issued. The reconfiguration starts at time 00:00:00 (hh:mm:ss), and the Execute phase ends at 00:08:08. The Commit phase is over at 00:08:12, and the reconfiguration ends at 00:08:18. The Recovery phase duration for read only workload is much shorter than that of other workloads, as explained in Section 4.2. Fig. 4 (b), (c), and (d) depict read latencies for different YCSB workloads: uniform, zipfian, and latest respectively.

There are two latency lines for this experiment – Original CF and Reconfigured CF (CF = Column Family). These refer to the queries using the old primary key and the new primary key respectively, with the switch-over happening at the atomic commit point. We only query using the primary key because using secondary index is not recommended for our use case, where the cardinality of the columns that participate in the reconfiguration is too high [9].

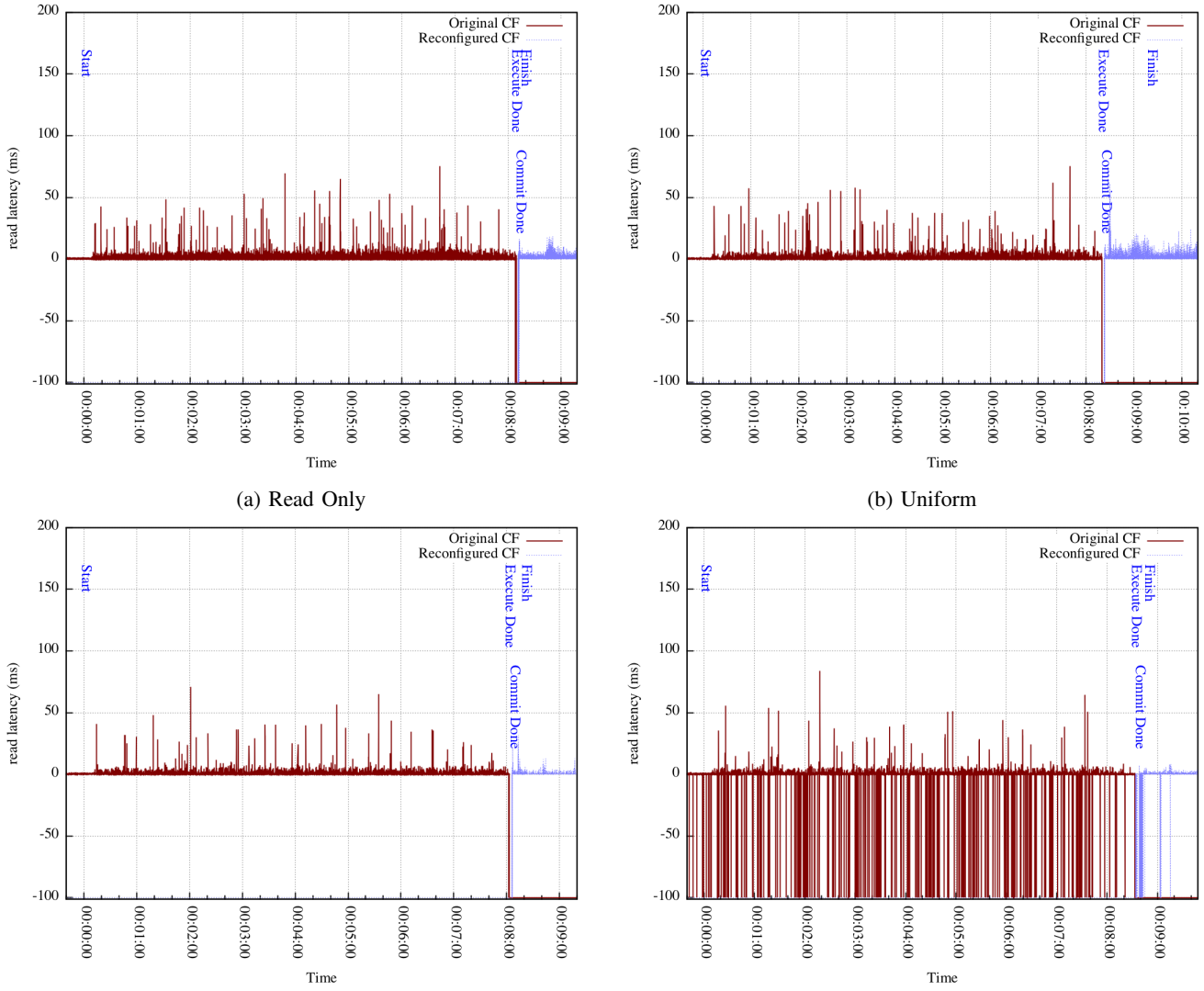
During the reconfiguration, we can see occasional latency spikes in the different workloads. This is due to increased disk activities during migration, where a lot of entries are read off the disk. We observe less frequent latency spikes in zipfian and latest workloads than in read only and uniform workloads. This is because of the skewed key access patterns under zipfian and latest workloads and the effect of caching for "popular" keys. This is discussed further in Section 4.4.2.

Negative values for read latency show failed reads (unavailability). We observe higher read unavailability in the Commit phase when SSTables of the Original CF and the Reconfigured CF are being swapped. In the Commit phase, each Cassandra peer swaps the physical SSTables of the Original CF and Reconfigured CF in its local file system, and reloads the column family definitions. Although reads are not explicitly blocked during the Commit phase, swapping physical SSTables and reloading the column family definitions cause some read operations to fail. This is because SSTable swap and schema reload does not happen exactly at the same time.

Fig. 4d shows many failed reads throughout the time. This is because the multi-threaded YCSB workload generator tries to read some of the most recent writes even before they are committed. Such read requests appear to fail, since the primary key being searched is not inserted yet. As explained in Section 4.3, this is inherent to the multi-threaded YCSB workload generator.

4.4.2. Read Latency CDF. Fig. 3 shows the CDF of read latencies under various workloads while reconfiguration is being executed. As a baseline, we also plot the CDF of read latency when no reconfiguration is being run. When measuring the latencies, we only considered latencies for successful reads. Note that overall availability of Parqua system is high as presented in Table 1.

First, the read only workload shows the same median (50th percentile) latency as the baseline, and only shows degraded tail latency above the 80th percentile. The uniform workload has a slightly higher latency than the read only workload, indicating that the injection of write operations adds a slight increase of latency. Compared to the uniform



(a) Read Only

(b) Uniform

(c) Zipfian

(d) Latest

Figure 4: **Read Latency for (a) Read only operations (no writes), and three read-write YCSB workloads: (b) Uniform, (c) Zipfian, and (d) Latest. Times shown are in hh:mm:ss. Failed reads are shown as negative latencies.**

workload, zipfian and latest workload performs better for the slowest 30th percentile of queries (beyond 70th percentile point). This is explained by the fact that the key access patterns of zipfian and latest workloads are concentrated at a smaller number of keys whereas uniform workload chooses keys uniformly. As a result, these frequently-accessed key-value pairs in the former two workloads are available in the disk cache, while the latter workload incurs a lot of disk seeks.

Next, when we compare latest and zipfian workloads, we observe the latest workload is slightly faster up to the 90th percentile level, and they share similar read latency above that percentile. This is because for latest workload, the most frequently accessed keys are among those that were inserted most recently. Therefore, these recently inserted entries are present in Memtables, and the read queries on such entries invoke fewer disk seeks and are answerable directly from memory. For the slowest reads however, the large number of disk seeks during the Execute phase and the Recovery phase makes the tail longer for Parqua, independent of workload pattern. Nevertheless, we point out that having a small (20%) fraction of reads answered slower is preferable to shutting down the entire database.

4.5. Write Latency

In our next set of experiments, we investigate the write latency characteristics of Parqua system. Similar to Section 4.4, we aim to observe the effect of the reconfiguration on normal write operations.

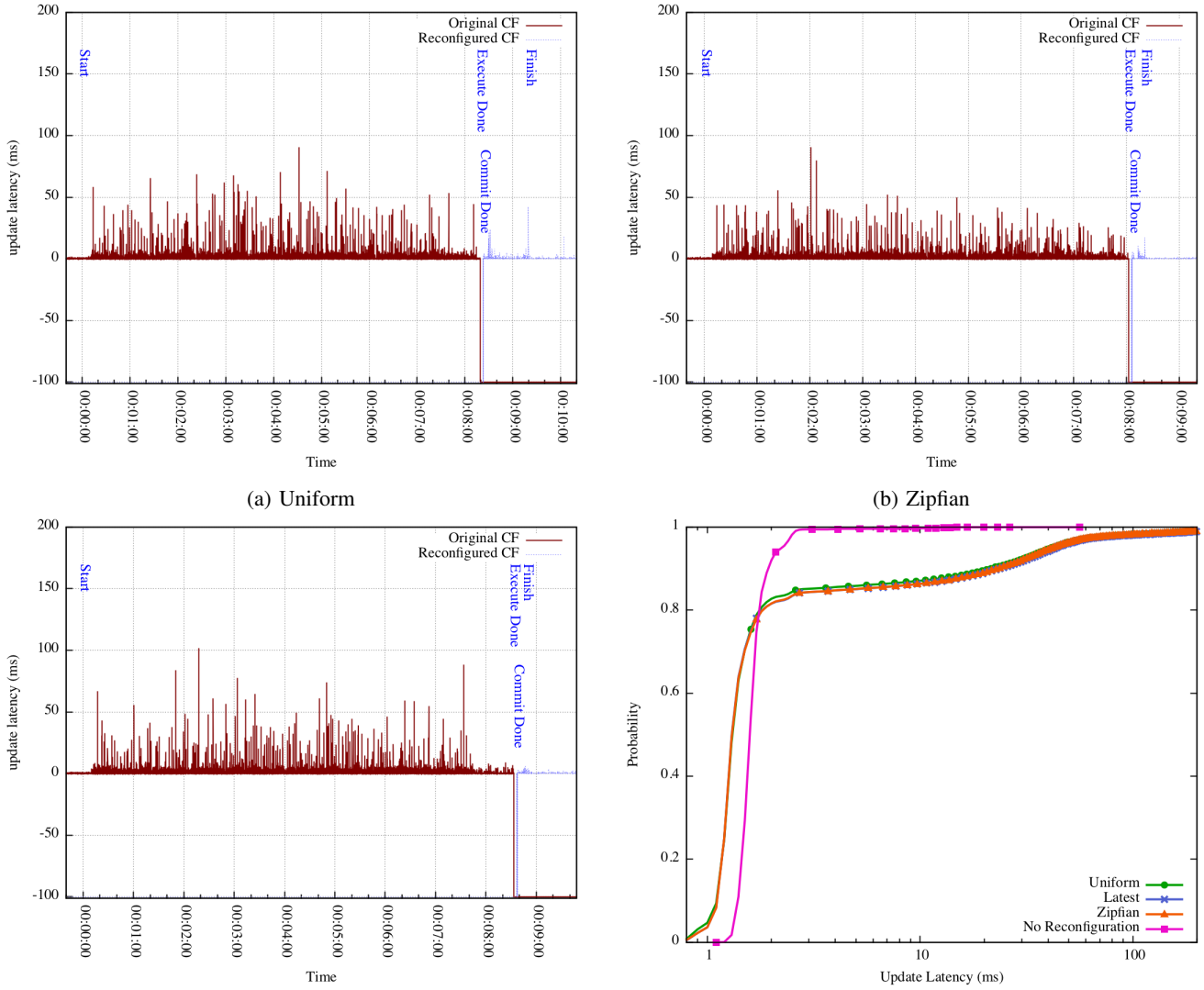


Figure 5: **Update latencies over time for three YCSB workloads: (a) Uniform, (b) Zipfian, and (c) Latest. (d) depicts CDF of write latencies for various workloads. Times shown are in hh:mm:ss. Failed inserts are shown as negative latencies.**

4.5.1. Write Latency Over Time. Fig. 5 (a), (b), and (c) depict the write latencies over time for different workloads under reconfiguration. This is for the same experiment as Fig. 4. We see that many operations fail in the Commit phase, similar to Section 4.4. As explained in Section 3.2, in the Commit phase the coordinator locks writes for the column family being reconfigured, and unlocks it when the primary key of that column family is updated. Once writes are unlocked, the query with the new primary key starts to succeed while the query with the old partition key fails. Similar to Section 4.4, we also observe the latency spikes in the Execute phase and the Recovery phase. However, unlike in Section 4.4, we do not see differing behaviors of latencies across workloads. This is because Cassandra is a write-optimized database, which does not incur a disk seek. After the reconfiguration is over, the write latency stabilizes and behaves similar to before the reconfiguration has begun.

4.5.2. Write Latency CDF. In Fig. 5d, we plot the CDF of update latencies for different workloads. We observe the three workloads perform similarly across different latency percentiles. This is because Cassandra's write path consists of appending to commit log and writing into Memtable, and there is little disk I/O involved. Compared to baseline, Parqua shows degraded tail latency above the 80th percentile. This is due to higher disk utilization level when the reconfiguration is taking place, thus commit log appending faces interferences and takes longer. The tail latency is aggravated by the use of consistency level of 'ALL', since the coordinator node has to wait for acknowledgements from

all of the replicas.

As a result, we conclude that Parqua exhibits slightly degraded write latency during the reconfiguration, especially at the tail. However, the write latency is not affected by various workload patterns, and recovers after the reconfiguration is completed.

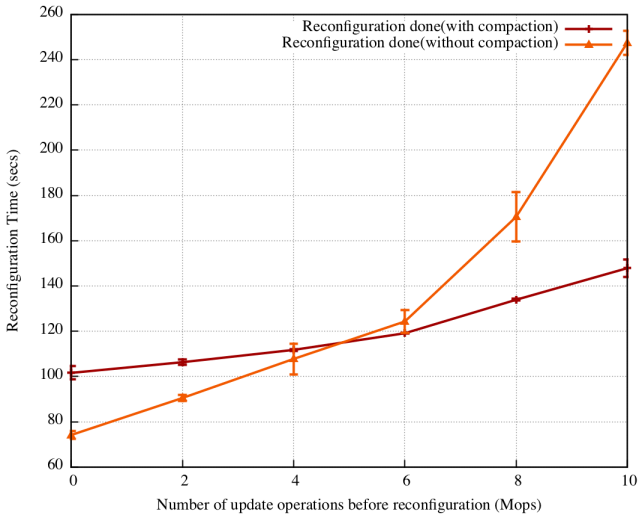


Figure 6: **Reconfiguration time for number of injected update operations under two different implementations of Parqua system.**

4.6. Scalability

Next, we measure how well Parqua scales with: (1) database size, (2) operation injection rate, (3) cluster size, and (4) replication factor. To evaluate our system's scalability, we measured the total reconfiguration times along with a breakdown by phase. In order to isolate the effects of injected operations, we do not inject operations for the experiments in Section 4.6.1, 4.6.3, and 4.6.4. We investigate the scalability of our system under operation injection in Section 4.6.2.

4.6.1. Database Size. Fig. 7a depicts the reconfiguration time as the database size is increased up to 30 GB. Since the replication factor was 3, 30 GB here means 90 GB overall amount of entries (without accounting for duplicate entries). In this plot, we observe the total reconfiguration time scales linearly with database size. This is expected as a bulk of the reconfiguration time is spent transferring data (the Execute phase), and this is three overlapping lines in the plot.

4.6.2. Operation Injection Rate. Fig. 7b shows the result of varying the operation injection rate from 0 ops/s to 1500 ops/s. (database size was fixed at 10 GB)

The reconfiguration time increases linearly with the operation injection rate. We present the explanation for completion time of each phase. First, the Recovery phase duration increases steadily with operation rate. This happens because as more operations are injected during reconfiguration, their replay during the Recovery phase takes longer. This is evident from the growing gap between the "Reconfiguration done" and the "Commit phase done" lines.

Second, the Commit phase duration (the time between the Execute phase and the Commit phase lines) stays almost at constant across the increasing operation rate. This is because the Commit phase swaps the SSTables of Original CF with Reconfigured CF and reloads the schema with the new primary key, and both operations are independent of operation rate.

Third, the Execute phase also increases steadily along with the operation rate. The increase is due to accumulation of injected operations. In the Execute phase, each node iterates the primary key ranges that it is responsible for, and sends the entries to the appropriate destination nodes. Therefore, if a new entry is injected in the Execute phase before its key is iterated, this entry would be transferred to other nodes when Parqua iterates over that key, thus increasing the overall amount of the transferred data. The rate of increase at the Execute phase is much slower than at the Recovery phase, because not all newly injected operations are migrated in the Execute phase.

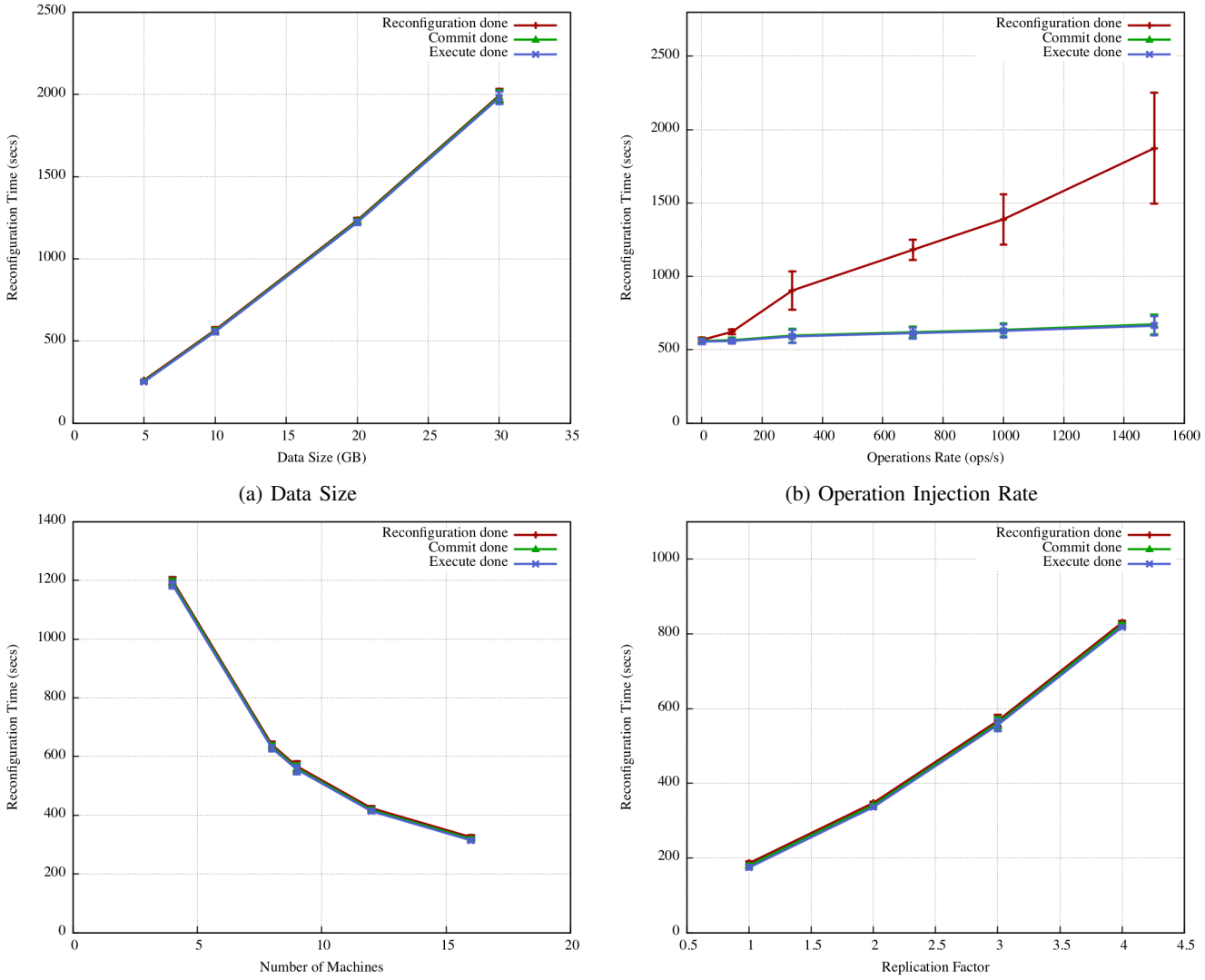
4.6.3. Replication Factor. Next, Fig. 7d shows the effect of increasing replication factor (number of replicas of each key) on the total reconfiguration time. We observe that the reconfiguration time increases as the replication factor increases. This is because a higher replication factor implies that more data exists in the underlying SSTables, and thus migration takes longer.

4.6.4. Cluster Size. Finally, we demonstrate the reconfiguration time as we scale the cluster size. Database size was fixed at 10 GB. In Fig. 7c, we observe that the reconfiguration time *decreases* as the number of Cassandra peers increases.The decrease occurs because as the number of machines increases, there is higher parallelism involved in the Execute phase. Observe that as the number of peers increases, the Commit phase and the Recovery phase durations stay constant whereas the Execute phase duration decreases.

We conclude that Parqua scales very well with cluster size – the larger the cluster, the faster is the reconfiguration time.

4.7. Effect of Compaction

Cassandra uses major compaction to periodically aggregate fragments of an entry (created due to updates). One available option in Parqua is to run Cassandra's major compaction before the Execute phase begins. The rationale behind this is that by compacting the fragmented entries first, we might be able to save the disk I/O time caused by on-demand aggregation of fragmented entries. In Fig. 6 we show the reconfiguration time for different number of update operations under these two different implementations of Parqua. In this experiment, we simulated fragmented entries by injecting update operations prior to reconfiguration while



(a) Data Size

(b) Operation Injection Rate

(c) Number of Cassandra Machines

(d) Replication Factor

Figure 7: **Morphus Scalability with: (a) Data Size, (b) Operation injection rate, measured in number of YCSB workload threads, (c) Number of machines, and (d) Replication Factor.**

disabling the automatic compaction. Also, for the purpose of our experiment, we minimized the effect of disk cache by flushing it every minute. We used 1 GB database size in order to observe the effect of increasing number of injected operations more easily. In Fig. 6, we observe that reconfiguration time is shorter for the implementation without compaction when no update operations are injected. However, as number of update operations increase, the reconfiguration time for the implementation without compaction increases rapidly and crosses over at 5 Mops (1 Mops = 10^6 ops) operations.

From this result, we conclude that the benefit of upfront major compaction heavily depends on the kind of workload that the database receives prior to the reconfiguration. We recommend executing major compaction for workloads that have frequent updates.

4.8. Migration Throttling

In our initial implementation of Parqua, we observed the read/write latencies of normal operations were affected by the reconfiguration. Profiling the query latency revealed that the normal requests were queued for a long time because Parqua's migration operations were flooding the queue. This can be explained by Cassandra's adoption of the staged event-driven architecture (SEDA) [29]. SEDA maintains a set of thread pools and queues each dedicated for specific tasks, which helps to achieve high overall throughput. In our case, Parqua's migration logic was sharing the same stage with normal requests, causing the queues to be overly crowded. To address this, we created a new SEDA thread pool that is dedicated exclusively for our Parqua operations. After this design change, we achieved 100-fold improvement

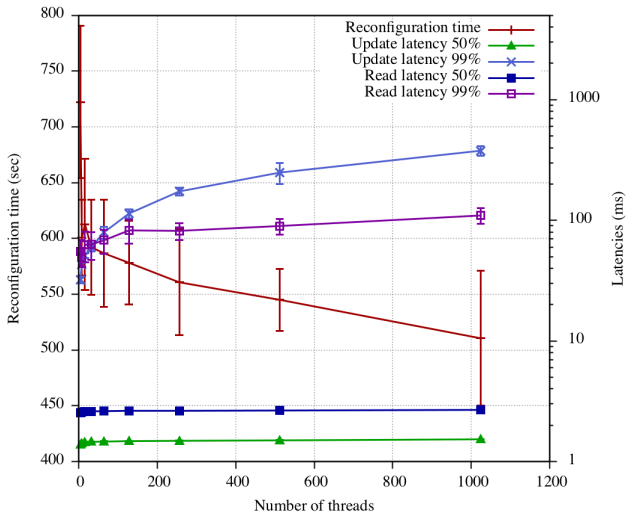


Figure 8: **Reconfiguration time and read/write latency over number of migration threads for reconfiguration.**

in tail latency.

Fig. 8 depicts the change in reconfiguration time and read/write latency under different number of threads in the Parqua migration thread pool. As the number of threads in the thread pool increases, reconfiguration time decreases. Reconfiguration time improves because of the increased parallelism under higher number of threads in the thread pool. The reconfiguration time plateaus as number of threads increases, as more threads compete for limited system resources.

However, this causes the Parqua to negatively affect the normal request latencies as Parqua's entry transfer competes with normal requests for other system resources (such as disk I/O and network). In Fig. 8, we observe the tail latency of reads and updates increases quickly at first. As number of threads increase, the update latency keep increases (note that the y-axis for the latency plots is in log scale), while read latency plateaus beyond 200 threads. The update latency grows much faster than the read latency, because most Parqua operations are "write" operations which share the same write path of normal requests in Cassandra. Thus, higher number of Parqua threads implies more contention of system resources for normal write requests.

5. Related Work

Google's Bigtable [13] and Amazon's Dynamo [18] were the first NoSQL databases. One key difference between them was that Bigtable was a sharded NoSQL database while Dynamo used a virtual ring-based design inspired from Chord [26]. Together, they inspired a long line of research in NoSQL databases both in academia and industry. The ring-based design was later used by second generation databases like Cassandra [22], Riak [6] and Voldemort [8]. Under the CAP [21] tradeoff, all of these systems are highly available even under network partition. Our work, Parqua, enables these systems to meet their availability guarantees even during reconfiguration.

Reconfiguration in databases has received considerable attention in the recent past. Our earlier work, Morphus [20] was the first to attempt live reconfiguration of a sharded NoSQL database like, MongoDB. Extending the Morphus design to virtual ring-based databases was a considerable challenge. This led to the design and implementation of Parqua. In the relational database space, Squall [19] attempts to live reconfigure H-Store [27], a partitioned main memory database. Squall design requires H-Store's sharded architecture, making it inapplicable to ring-based key-value stores. Online schema change [24] attempted by Rae et. al. resulted in lower availabilities compared to Parqua. For data migration, earlier work has explored stop-and-copy [14] and pre-copy-based [11], [17] approaches, both of which require locking [12] to transfer a consistent copy of data. Parqua needs locking for only a short duration, and provides high data availabilities.

6. Summary

In this paper, we introduced Parqua, a system which enables online reconfiguration in a ring-based distributed key-value store. We introduced the general system assumptions for Parqua, and proposed its detailed design. Next, we integrated Parqua in Cassandra, and implemented a reconfiguration that changes the primary key of a column family. We experimentally demonstrated Parqua achieves high nines of availability, and scales well with database size, cluster size, and operation rate. In fact, Parqua becomes faster as the cluster size increases.

References

[1] An Introduction to using Custom Timestamps in CQL3. http://planetcassandra.org/blog/an-introduction-to-using-custom-timestamps-in-cql3/. visited on 2015-04-25.

[2] Command to change shard key of a collection. https://jira.mongodb.org/browse/SERVER-4000. visited on 2015-1-5.

[3] DynamoDB. http://aws.amazon.com/dynamodb/. visited on 2015-5-5.

[4] Emulab. https://wiki.emulab.net/wiki/d710. visited on 2014-04-29.

[5] MongoDB. http://www.mongodb.org. visited on 2014-04-29.

[6] Riak. http://basho.com/riak/. visited on 2015-1-5.

[7] Troubles With Sharding - What Can We Learn From The Foursquare Incident? http://highscalability.com/blog/2010/10/15/troubles-with-sharding-what-can-we-learn-from-the-foursquare.html. visited on 2015-04-11.

[8] Voldemort. http://www.project-voldemort.com/voldemort/. visited on 2014-05-12.

[9] When to use an index in Cassandra. http://docs.datastax.com/en/cql/3.1/cql/ddl/ddl_when_use_index_c.html. visited on 2015-04-25.

[10] NoSQL market forecast 2013-2018, Market Research Media. http://www.marketresearchmedia.com/?p=568, 2012. visited on 2014-04-29.

[11] S. K. Barker, Y. Chi, H. Hacigümüs, P. J. Shenoy, and E. Cecchet. Shuttledb: Database-aware elasticity in the cloud. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.*, pages 33–43, 2014.

[12] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 169–179, New York, NY, USA, 2007. ACM.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[15] 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*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[16] 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. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8:1–8:22, Aug. 2013.

[17] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. In *Proceedings of the Very Large Database Endowment*, volume 4, pages 494–505. VLDB Endowment, May 2011.

[18] 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. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[19] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. E. Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of SIGMOD*, 2015.

[20] M. Ghosh, W. Wang, G. Holla, and I. Gupta. Morphus: Supporting online reconfigurations in sharded nosql systems. In *12th IEEE International Conference on Autonomic Computing (ICAC 15)*, Grenoble, France, 2015. IEEE.

[21] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[22] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[23] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[24] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in F1. In *Proceedings of the Very Large Database Endowment*, volume 6, pages 1045–1056. VLDB Endowment, Aug. 2013.

[25] Stackoverflow. Altering Cassandra column family primary key. http://stackoverflow.com/questions/18421668/alter-cassandra-column-family-primary-key-using-cassandra-cli-or-cql. visited on 2014-04-29.

[26] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[27] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[28] TechRepublic. The great primary key debate. http://www.techrepublic.com/article/the-great-primary-key-debate/. visited on 2014-04-29.

[29] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[30] W. G. Yee. Orbitz: Technical challenges and opportunities in a leading online travel business. https://sites.google.com/site/