Efficient Background Mechanisms for Data-Intensive Cloud Systems

Mainak Ghosh

January 30, 2016: Prelim Proposal

Abstract

The performance of data-intensive cloud computing systems depend heavily not only on foreground mechanisms (like query processing, computation, etc.) but also on background mechanisms like data placement, compaction, replication, and reconfiguration. While the former has been the focus of much research, this thesis focuses on background mechanisms. In this thesis, we claim that design and implementation of efficient new background mechanisms can improve performance of data-intensive cloud systems. We cover both cloud storage as well as cloud computation systems. For example, we propose reconfiguration as a new background process in NoSQL databases, maintaining availability and latency while reconfiguring the database. Similarly, we also explore compaction in log structured databases. In real-time analytics systems like Druid, we propose adaptive replication schemes that can reduce the storage space without loss in query throughput. In actor frameworks, we propose an adaptive replication strategy that can change the number of activations of a stateful actor based on read/write proportion.

1 Introduction

Over the past decade, data-intensive cloud systems have revolutionized the storage and processing of large amounts of data. Current systems can scale to exabytes of data and process them in time ranging from a few hours to less than a second depending on query complexity. Looking ahead, with the advent of new applications like Internet of Things (IoT), the total sensor data is predicted to reach 44 Zettabytes by year 2020 [42]. New research also shows that latency is important and related to user retention and revenue [43]. To meet these challenges of the next decade, big data systems research is formally trying to push the boundaries of storage and performance.

Performance of data-intensive cloud systems is largely measured by benchmarking the client response time (foreground process). We observe that in reality, it is heavily dependent on background mechanisms. For example, efficient data placement in storage and compute systems can improve query response and computation time respectively. But, it is affected by changing workloads, cluster size and network characteristics. Efficient background mechanisms can help storage adapt to these factors thereby improving system performance. Unfortunately, many present day systems employ ad-hoc techniques [11] in implementing these background mechanisms without solving the core algorithmic and system-level challenges. This leads to the central hypothesis of this thesis:

We claim that design and implementation of efficient new background mechanisms can improve performance of data-intensive cloud systems.

<table>
<thead>
<tr>
<th>Background Mechanism</th>
<th>Type of System</th>
<th>Performance Metric Affected</th>
<th>Applicable to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reconfiguration</td>
<td>Storage</td>
<td>Read/Write Latency</td>
<td>MongoDB, Cassandra</td>
</tr>
<tr>
<td>Compaction</td>
<td>Storage</td>
<td>Read Latency</td>
<td>Cassandra, Riak</td>
</tr>
<tr>
<td>Adaptive Replication</td>
<td>Computation</td>
<td>Query Throughput</td>
<td>Druid</td>
</tr>
<tr>
<td>Adaptive Replication</td>
<td>Programming</td>
<td>Read/Write Latency</td>
<td>Orleans</td>
</tr>
<tr>
<td>Network Topology</td>
<td>Computation</td>
<td>Computation Time</td>
<td>PowerGraph, Storm</td>
</tr>
</tbody>
</table>

Table 1: Background mechanisms considered in the proposal and their performance impact

Table 1 summarizes the set of example background mechanisms we improved and/or built from scratch to backup our hypothesis. We targeted background mechanisms in storage, computation and programming
frameworks. In Section 2, we will briefly discuss the projects and the challenges involved. Section 3 highlights the broad impact our solutions will have. In the subsequent sections, we will dive deeper into the projects. Finally, we will end with our execution plan and summary in Section 8 and 9 respectively.

2 Intellectual Merit

2.1 Reconfiguration in Distributed Databases [COMPLETE]

Distributed NoSQL databases have been widely adopted because of their high availability guarantees. In production deployments of NoSQL systems and key-value stores, reconfiguration operations have been a persistent and major pain point [14, 27]. Reconfiguration operations deal with changes to configuration parameters at the level of a database table or the entire database itself, in a way that affects a large amount of data all at once. Examples include schema changes like changing the shard/primary key that is used to split a table into blocks, or changing the block size itself, or increasing or decreasing the cluster size (scale out/in). Existing solutions naively shut down the database and then export and reimport the data into the new configurations which reduces availability. In Morphus and Parqua, we design two systems that allow reconfiguration to happen while database is still serving reads and writes. In Morphus, we also make two additional contributions: 1) use provably optimal algorithms to minimize the amount of network transfer required and 2) data migration traffic adapts itself to datacenter’s network topology.

Morphus has been published in proceedings of ICAC 2015 [71] and as journal in a special issue of IEEE Transactions on Emerging Topics in Computing [72]. The Parqua system has been accepted as a short paper in the proceedings of ICCAC 2015 [91].

2.2 Compaction in Log Structured Databases [COMPLETE]

Supporting fast reads and writes simultaneously on a large database can be quite challenging in practice [56, 69]. Since today’s workloads are write-heavy, many NoSQL databases [16, 26, 38, 79] choose to optimize writes over reads. Log structured storage allows fast write by appending write (create, update and delete) operations to a log. Reads require log merging and are therefore, slower in comparison. Compaction is an existing background process that proactively merges the log files in multiple iterations. Thus, it is I/O bound. The choice of log files to merge in each iteration is defined by the compaction strategy. The problem of compaction and the existing strategies used in databases like Cassandra lack theoretical insights. In this work, we make the following contributions: 1) formulate compaction as an optimization problem, 2) propose a set of heuristics and analyze their closeness to optimality, and 3) compare their performance using simulation.

The work has been published in the proceedings of ICDCS 2015 [70] (Theory track).

2.3 Adaptive Replication in Real-Time Analytics System [ONGOING]

Real-time analytics use high amount of I/O and CPU parallelism to meet the sub second latency requirements of individual applications. As data streams in, it is broken down into segments based on time of arrival. For high throughput, inter query parallelism is exploited by replicating these segments. Current replication schemes do not consider the popularity of segments. Thus the effective disk utilization is poor. In this work, we plan to adaptively replicate segments based on popularity. We plan to use forgetful datastructures to maintain the segment popularity. We plan to evaluate our scheme over many different realistic workloads.

2.4 Adaptive Replication in Actor Frameworks [FUTURE WORK]

Actor frameworks enable a programmer to write distributed applications without worrying about complexities that may arise due to data races. They provide neat abstractions for hassle-free development while the runtime takes care of systems level issues like cluster management, actor instantiation/replication on demand, etc. In production deployments, they have scaled linearly even when the number of actors in the systems reached a million. Unfortunately, actors with state are not easy to scale and turn out to be a performance bottleneck. While under heavy write load, a small number of actors is desired for smaller
reconciliation overhead. Conversely a heavy read load demands more actor instances. We plan to tackle this tradeoff by using an optimal algorithm [95] that adapts to changing workload pattern. We further plan to propose a new and efficient reconciliation strategy that is compatible with the adaptive replication strategy.

2.5 Topology-aware data placement in Supercomputers [FUTURE WORK]

Supercomputers have been traditionally used for computation intensive batch processing jobs in scientific computing community. For cost reasons, it has not been considered for the data-intensive cloud computing applications. We believe high throughput applications like graph and stream processing systems could be optimized for a supercomputer. The main challenge lies in embedding the communication graph inside a supercomputer’s 3d torus network topology. As an instance, we consider PowerGraph [73], a popular graph processing framework. To embed the communication graph, we need to modify PowerGraph’s partitioning strategy. Current partitioning algorithms try to minimize network communication as well as balance compute load. Adding network awareness can affect the other parameters. Finding the optimal tradeoff point is a challenging problem that we plan to solve in this work.

3 Broader Impact

Most of the problems that I have worked on, are born out of real-life issues faced by customers. In the next few sections, we cite the relevant issues to further highlight the importance of our proposed solutions in the overall scenario.

- **Reconfiguration in Distributed Databases:** Failure to reconfigure databases on time can lead to lower performance which impacts annual revenues. It has led to outages such as the one at Foursquare [46]. While such reconfigurations may not be very frequent operations, they are a significant enough pain point that they have engendered multiple JIRA (bug tracking) entries (e.g., [14]), discussions and blogs (e.g., [25]). Inside Google, resharding MySQL databases took two years and involved a dozen teams [64]. Thus, we believe that this is a major problem that directly affects the life of a system administrator – without an automated reconfiguration primitive, reconfiguration operations today are laborious and manual, consume significant amounts of time, and leave open the room for human errors during the reconfiguration. We have implemented our design in two popular nosql databases – 1) MongoDB and 2) Cassandra.

- **Compaction in Log Structured Databases:** Log structured databases like Cassandra have a wide user base [15] ranging from online course platforms like Coursera to movie streaming services like Netflix to space agencies like NASA. One of the selling points have been Cassandra's excellent read write performance. We envisage some of our proposed compaction heuristics once implemented in these systems will further improve the read latencies thus, making them popular in industry.

- **Adaptive Replication in Real-Time Analytics System:** Aggregate queries are frequently used to summarize large data. At the same time, they are also expected to reflect change in data in real time. Druid, a real-time analytics system is used in Netflix. Our system, Getafix, by adapting replication to workload change can potentially support high throughput in real-time analytics system like Druid. The general design can be easily extended to other systems too.

- **Adaptive Replication in Actor Frameworks:** Actor Frameworks have been used to design service around Halo 4, a popular game by 343 industries [1], and distributed graph processing engine [57]. Stateful actors are used to represent entities like graph partitions. Under varying read-write load, determining the optimal scaling strategy of these actors is difficult which leads to poor response time. Our work tries to address this problem by modifying the framework which in turn will impact all applications using it.

- **Topology-aware data placement in Supercomputers:** By running high throughput cloud computing systems in supercomputers, we hope to make its case as a viable alternative to datacenters. Currently, supercomputers are considered only for high performance computing jobs. We believe supercomputers’ tightly coupled network design can lend itself favorably for data intensive cloud computing applications. By modifying the data placement module in current cloud computing frameworks, we hope to enable better network utilization, thereby improving performance.
4 Reconfiguration

4.1 Motivation

Distributed NoSQL storage systems comprise one of the core technologies in today’s cloud computing revolution. These systems are attractive because they offer high availability and fast read/write operations for data. They are used in production deployments for online shopping, content management, archiving, e-commerce, education, finance, gaming, email and healthcare. The NoSQL market is expected to earn $14 Billion revenue during 2015-2020, and become a $3.4 Billion market by 2020 [48].

In today’s NoSQL deployments [26, 35, 59, 79], data-centric 1 global reconfiguration operations are quite inefficient. This is because executing them relies on ad-hoc mechanisms rather than solving the core underlying algorithmic and system design problems. The most common solution involves first saving a table or the entire database, and then re-importing all the data into the new configuration [11]. This approach leads to a significant period of unavailability. A second option may be to create a new cluster of servers with the new database configuration and then populate it with data from the old cluster [53, 66, 68]. This approach does not support concurrent reads and writes during migration, a feature we would like to provide.

Consider an admin who wishes to change the shard key inside a sharded NoSQL store like MongoDB [11]. The shard key is used to split the database into blocks, where each block stores values for a contiguous range of shard keys. Queries are answered much faster if they include the shard key as a query parameter (otherwise the query needs to be multicast). Today’s systems strongly recommend that the admin decide the shard key at database creation time, but not change it afterwards. However, this is challenging because it is hard to guess how the workload will evolve in the future. In reality, admins need to change the shard key prompted by either changes in the nature of the data being received, or due to evolving business logic, or by the need to perform operations like join with other tables, or due to prior design choices that are sub-optimal in hindsight. As a result, the reconfiguration problem has been a fervent point of discussion in the community for many years [4, 25].

In this work, we present two systems that support automated reconfiguration. Our systems, called Morphus and Parqua, allow reconfiguration changes to happen in an online manner, that is, by concurrently supporting reads and writes on the database table while its data is being reconfigured.

4.2 Morphus

4.2.1 Assumptions

Morphus assumes that the NoSQL system features: 1) master-slave replication, 2) range-based sharding (as opposed to hash-based) 2, and 3) flexibility in data assignment 3. Several databases satisfy these assumptions, e.g., MongoDB [35], RethinkDB [36], CouchDB [16], etc. To integrate our Morphus system we chose MongoDB due to its clean documentation, strong user base, and significant development activity. To simplify discussion, we assume a single datacenter, but our paper [72] present results for geo-distributed experiments. Finally, we focus on NoSQL rather than ACID databases because the simplified CRUD (Create, Read, Update, Delete) operations allow us to focus on the reconfiguration problem – addressing ACID transactions is an exciting avenue that our paper opens up.

Morphus solves three major challenges: 1) in order to be fast, data migration across servers must incur the least traffic volume; 2) degradation of read and write latencies during reconfiguration must be small compared to operation latencies when there is no reconfiguration; 3) data migration traffic must adapt itself to the datacenter’s network topology.

---

1 Data-centric reconfiguration operations deal only with migration of data residing in database tables. Non-data-centric reconfigurations are beyond our scope, e.g., software updates, configuration table changes, etc.

2 Most systems that hash keys use range-based sharding of the hashed keys, and so our system applies there as well. Our system also works with pure hash sharded systems, though it is less effective.

3 This flexibility allows us to innovate on data placement strategies. Inflexibility in consistent hashed systems like Cassandra [79] require a different solution, and was investigated in Parqua.
4.2.2 MongoDB System Model

We have chosen to incorporate Morphus into a popular sharded key-value store, MongoDB [35] v2.4. Beside its popularity, our choice of MongoDB is also driven by its clean documentation, strong user base, and significant development and discussion around it.

A MongoDB deployment consists of three types of servers. The _mongod_ servers store the data chunks themselves—typically, they are grouped into disjoint _replica sets_. Each replica set contains the same number of (typically 3) servers which are exact replicas of each other, with one of the servers marked as a primary (master), and others acting as secondaries (slaves). The configuration parameters of the database are stored at the _config servers_. Clients send CRUD (Create, Read, Update, Delete) queries to a front-end server, called _mongos_. The mongos servers also cache some of the configuration information from the config servers, e.g., in order to route queries they cache mappings from each chunk range to a replica set.

A single database table in MongoDB is called a _collection_. Thus, a single MongoDB deployment consists of several collections.

4.2.3 Reconfiguration Phases in Morphus

Morphus allows a reconfiguration operation to be initiated by a system administrator on any collection. Morphus executes the reconfiguration via five sequential phases, as shown in Figure 1.

First Morphus prepares for the reconfiguration by creating partitions (with empty new chunks) using the new shard key (Prepare phase). Second, Morphus isolates one secondary server from each replica set (Isolation phase). In the third Execution phase, these secondaries exchange data based on the placement plan decided by mongos. In the meantime, further operations may have arrived at the primary servers—these are now replayed at the secondaries in the fourth Recovery phase. When the reconfigured secondaries are caught up, they swap places with their primaries (Commit phase).

At this point, the database has been reconfigured and can start serving queries with the new configuration. However, other secondaries in all replica sets need to reconfigure as well—this slave catchup is done in multiple rounds, with the number of rounds equal to the size of the replica set.

We discuss the individual phases in detail in our paper [72].

Read-Write Behavior. The end of the Commit phase marks the switch to the new shard key. Until this
4.2.4 Algorithms for Efficient Shard Key Reconfigurations

A reconfiguration operation entails the data present in shards across multiple servers to be resharded. The new shards need to be placed at the servers in such a way as to: 1) reduce the total network transfer volume during reconfiguration, and 2) achieve load balance. This section presents optimal algorithms for this planning problem.

We present two algorithms for placement of the new chunks in the cluster. Our first algorithm is greedy and is optimal in the total network transfer volume. However, it may create bottlenecks by clustering many new chunks at a few servers. Our second algorithm, based on bipartite matching, is optimal in network transfer volume among all those strategies that ensure load balance.

**Greedy Assignment.** The greedy approach considers each new chunk independently. For each new chunk $NC_i$, the approach evaluates all the $N$ servers. For each server $S_j$, it calculates the number of data items $W_{NC_i,S_j}$ of chunk $NC_i$ that are already present in old chunks at server $S_j$. The approach then allocates each new chunk $NC_i$ to that server $S_j$ which has the maximum value of $W_{NC_i,S_j}$, i.e., $argmax_{S_j}(W_{NC_i,S_j})$. As chunks are considered independently, the algorithm produces the same output irrespective of the order in which chunks are considered by it.

**Lemma 4.1.** *The greedy algorithm is optimal in total network transfer volume.*

To illustrate the greedy scheme in action, Fig. 2 provides two examples for the shard key change operation. In each example, the database has 3 old chunks $OC_1-OC_3$ each containing 3 data items. For each data item,
we show the old shard key $K_0$ and the new shard key $K_n$ (both in the ranges 1-9). The new configuration splits the new key range evenly across 3 chunks shown as $NC_1 - NC_3$.

In Fig. 2a, the old chunks are spread evenly across servers $S_1 - S_3$. The edge weights in the bipartite graph show the number of data items of $NC_j$ that are local at $S_j$, i.e., $W_{NC_i,S_j}$ values. Thick lines show the greedy assignment.

However, the greedy approach may produce an unbalanced chunk assignment for skewed bipartite graphs, as in Fig. 2b. While the greedy approach minimizes network transfer volume, it assigns new chunks $NC_2$ and $NC_3$ to server $S_1$, while leaving server $S_3$ empty.

**Load Balance via Bipartite Matching.** Load balancing chunks across servers is important for several reasons: i) it improves read/write latencies for clients by spreading data and queries across more servers; ii) it reduces read/write bottlenecks; iii) it reduces the tail of the reconfiguration time, by preventing allocation of too many chunks to any one server.

Our second strategy achieves load balance by capping the number of new chunks allocated to each server. With $m$ new chunks, this per-server cap is $\lceil m/N \rceil$ chunks. We then create a bipartite graph with two sets of vertices — top and bottom. The top set consists of $\lceil m/N \rceil$ vertices for each of the $N$ servers in the system; denote the vertices for server $S_j$ as $S^k_j$ and bottom vertex $NC_j$ have an edge cost equal to $|NC_j| - W_{NC_i,S_j}$ i.e., the number of data items that will move to server $S_j$ if new chunk $NC_j$ were allocated to it.

Assigning new chunks to servers in order to minimize data transfer volume now becomes a bipartite matching problem. Thus, we find the minimum weight matching by using the classical Hungarian algorithm [29]. The complexity of this algorithm is $O((N.V + m).N.V.m)$ where $V = \lceil m/N \rceil$ chunks. This reduces to $O(m^3)$. The greedy strategy becomes a special case of this algorithm with $V = m$.

**Lemma 4.2.** Among all load-balanced strategies that assign at most $V = \lceil m/N \rceil$ new chunks to any server, the Hungarian algorithm is optimal in total network transfer volume.

Fig. 2b shows the outcome of the bipartite matching algorithm using dotted lines in the graph. While it incurs the same overall cost as the greedy approach, it additionally provides the benefit of a load-balanced new configuration, where each server is allocated exactly one new chunk.

While we focus on the shard key change, this technique can also be used for other reconfigurations like changing shard size, or cluster scale out and scale in. The bipartite graph would be drawn appropriately (depending on the reconfiguration operation), and the same matching algorithm used. For purpose of concreteness, the rest of the paper focuses on shard key change.

Finally, although we have used datasize (number of key-value pairs) as the main cost metric. Instead we could use traffic to key-value pairs as the cost metric, and derive edge weights in the bipartite graph (Fig. 2) from these traffic estimates. Hungarian approach on this new graph would balance out traffic load, while trading off optimality – further exploration of this variant is beyond our scope in this paper.

**4.2.5 Network-Awareness**

Datacenters use a wide variety of topologies, the most popular being hierarchical, e.g., a typical two-level topology consists of a core switch and multiple rack switches. Others that are commonly used in practice include fat-trees [49], CLOS [78], and butterfly [77].

Our first-cut data migration strategy discussed in Section 4.2.3 was chunk-based: it assigned as many sockets (TCP streams) to a new chunk $C$ at its destination server as there are source servers for $C$ i.e., it assign one TCP stream per server pair. Using multiple TCP streams per server pair has been shown to better utilize the available network bandwidth [60]. Further, the chunk-based approach also results in stragglers in the execution phase. Particularly, we observe that 60% of the chunks finish quickly, followed by a 40% cluster of chunks that finish late.

To address these two issues, we propose a weighted fair sharing (WFS) scheme that takes both data transfer size and network latency into account. Consider a pair of servers $i$ and $j$, where $i$ is sending some data to $j$ during the reconfiguration. Let $D_{i,j}$ denote the total amount of data that $i$ needs to transfer to $j$, and $L_{i,j}$ denote the latency in the shortest network path from $i$ to $j$. Then, we set $X_{i,j}$, the weight for the flow from server $i$ to $j$, as follows:

$$X_{i,j} \propto D_{i,j} \times L_{i,j}$$
In our implementation, the weights determine the number of sockets that we assign to each flow. We assign each destination server $j$ a total number of sockets $X_j = K \times \frac{\sum_i D_{i,j}}{\sum_{i,j} D_{i,j}}$, where $K$ is the total number of sockets throughout the system. Thereafter each destination server $j$ assigns each source server $i$ a number of sockets, $X_{i,j} = X_j \times \frac{C_{i,j}}{\sum_i C_{i,j}}$.

However, $X_{i,j}$ may be different from the number of new chunks that $j$ needs to fetch from $i$. If $X_{i,j}$ is larger, we treat each new chunk as a data slice, and iteratively split the largest slice into smaller slices until $X_{i,j}$ equals the total number of slices. Similarly, if $X_{i,j}$ is smaller, we use iterative merging of the smallest slices. Finally, each slice is assigned a socket for data transfer. Splitting or merging slices is only for the purpose of socket assignment and to speed up data transfer; it does not affect the final chunk configuration which was computed in the Prepare phase.

Our approach above could have used estimates of available bandwidth instead of latency estimates. We chose the latter because: i) they can be measured with a lower overhead, ii) they are more predictable over time, and iii) they are correlated to the effective bandwidth.

### 4.2.6 Evaluation

**Setup.** We use the dataset of Amazon reviews as our default collection [82]. Each data item has 10 fields. We choose `productID` as the old shard key, `userID` as the new shard key, while update operations use these two fields and a `price` field. Our default database size is 1 GB (we later show scalability with data size).

The default Morphus cluster uses 10 machines. These consist of one mongos (front-end), and 3 replica sets, each containing a primary and two secondaries. There are 3 config servers, each co-located on a physical machine with a replica set primary – this is an allowed MongoDB installation. All physical machines are d710 Emulab nodes [22] with a 2.4 GHz processor, 4 cores, 12 GB RAM, 2 hard disks of 250 GB and 500 GB, 64 bit CentOS 5.5, and connected to a 100 Mbps LAN switch.

We implemented a custom workload generator that injects YCSB-like workloads via MongoDB’s `pymongo` interface. Our default injection rate is 100 ops/s with 40% reads, 40% updates, and 20% inserts. To model realistic key access patterns, we select keys for each operation via one of three YCSB-like [62] distributions: 1) Uniform (default), 2) Zipf, and 3) Latest. For Zipf and Latest distributions we employ a shape parameter $\alpha = 1.5$. The workload generator runs on a dedicated pc3000 node in Emulab running a 3GHz processor, 2GB RAM, two 146 GB SCSI disks, 64 bit Ubuntu 12.04 LTS.

Morphus was implemented in about 4000 lines of C++ code. The code is publicly available at [http://dprg.cs.uic.edu/downloads](http://dprg.cs.uic.edu/downloads). Each plotted datapoint is an average of at least 3 experimental trials, shown along with standard deviation bars. Section 4.2.4 outlined two algorithms for the shard key change reconfiguration – Hungarian and Greedy. We implemented both into Morphus, and call these variants Morphus-H and Morphus-G respectively.

**Effect on Reads and Writes.** A key goal of Morphus is availability of the database during reconfiguration. To evaluate this, we generate read and write requests and measure their latency while a reconfiguration is

![CDF of Read Latency Distribution (log axis)](image1)

![CDF of Write Latency Distribution (log axis)](image2)
in progress. We use Morpus-G with chunk based migration scheme. We have run separate experiments for all the key access distributions and also for a read only workload.

Table 4a lists the percentage of read and write requests that succeeded during reconfiguration. The number of writes that fail is low: for the Uniform and Zipf workloads, fewer than 2% writes fail. We observe that many of the failed writes occur during one of the write throttling periods. Recall from Section 4.2.3 that there are as many write throttling periods as the replica set size, with one throttle period at the end of each reconfiguration round. The Latest workload has a slightly higher failure rate since a key that was attempted to be written is more likely to be attempted to be written or read again in the near future. Yet, the write failure rate of 3.2% and read failure rate of 2.8% is reasonably low.

Overall, the availability numbers are higher at two or three-9's for Uniform and Zipf workload, comparable to the scenario with no insertions. We conclude that unless there is temporal and spatial (key-wise) correlation between writes and reads (i.e., Latest workloads), the read latency is not affected much by concurrent writes. When there is correlation, Morpus mildly reduces the offered availability.

To flesh this out further, we plot in Fig. 3a the CDF of read latencies for the four settings, and when there is no reconfiguration (Uniform workload). Notice that the horizontal axis is logarithmic scale. We only consider latencies for successful reads. We observe that the 96th percentile latencies for all workloads are within a range of 2 ms. The median (50th percentile) latency for No Reconfiguration is 1.4 ms, and this median holds for both the Read only (No Write) and Uniform workloads. The medians for Zipf and Latest workloads are lower at 0.95 ms. This lowered latency is due to two reasons: caching at the mongod servers for the frequently-accessed keys, and in the case of Latest the lower percentage of successful reads. In Fig. 3b, we plot the corresponding CDF for write latencies. The median for writes when there is no reconfiguration (Uniform workload) is similar to the other distributions.

We conclude that under reconfiguration, the read and write availability provided by Morpus is high (close to two 9’s), while latencies of successful writes degrade only mildly compared to when there is no reconfiguration in progress.

**Effect of Network Awareness.** First, Fig. 5a shows the length of the Execution phase (using a 500 MB Amazon collection) for two hierarchical topologies, and five migration strategies. The topologies are: i) homogeneous: 9 servers distributed evenly across 3 racks, and ii) heterogeneous: 3 racks contain 6, 2, and 1 servers respectively. The switches are Emulab pc3000 nodes and all links are 100 Mbps. The inter-rack and intra-rack latencies are 2 ms and 1 ms respectively. The five strategies are: a) Fixed sharing, with one socket assigned to each destination node, b) chunk-based approach (Section 4.2.3), c) Orchestra [60] with \( K = 21 \), d) WFS with \( K = 21 \) (Section 4.2.5), and e) WFS with \( K = 28 \).

We observe that in the homogeneous clusters, WFS strategy with \( K = 28 \) is 30% faster than fixed sharing, and 20% faster than the chunk-based strategy. Compared to Orchestra which only weights flows by their data size, taking the network into account results in a 9% improvement in WFS with \( K = 21 \). Increasing \( K \) from

---

**Figure 4:** (a) Percentage of Reads and Writes that Succeed under Reconfiguration (b) Execution Phase Migration time for five strategies: (i) Fixed Sharing (FS), (ii) Chunk-based strategy, (iii) Orchestra with \( K = 21 \), (iv) WFS with \( K = 21 \), and (v) WFS with \( K = 28 \). (b) CDF of total reconfiguration time in chunk-based strategy vs. WFS with \( K = 28 \).
Figure 5: (a) Execution Phase Migration time for five strategies: (i) Fixed Sharing (FS), (ii) Chunk-based strategy, (iii) Orchestra with $K=21$, (iv) WFS with $K=21$, and (v) WFS with $K=28$. (b) CDF of total reconfiguration time in chunk-based strategy vs. WFS with $K=28$.

21 to 28 improves completion time in the homogeneous cluster, but causes degradation in the heterogeneous cluster. This is because a higher $K$ results in more TCP connections, and at $K=28$ this begins to cause congestion at the rack switch of 6 servers. Second, Fig. 5b shows that Morphus’ network-aware WFS strategy has a shorter tail and finishes earlier. Network-awareness lowers the median chunk finish time by around 20% in both the homogeneous and heterogeneous networks.

We conclude that the WFS strategy improves performance compared to existing approaches, and $K$ should be chosen high enough but without causing congestion.

Large Scale Experiment. In this experiment, we increase data and cluster size simultaneously such that the amount of data per replica set is constant. We ran this experiment on Google Cloud [24]. We used n1-standard-4 VMs each with 4 virtual CPUs and 15 GB of memory. The disk capacity was 1 GB and the VMs were running Debian 7. We generated a synthetic dataset by randomly dispersing data items among new chunks. Morphus-H was used for reconfiguration with WFS migration scheme and $K=\text{number of old chunks}$.

Fig. 4b shows a sublinear increase in reconfiguration time as data and cluster size increases. Note that x-axis uses log scale. In the Execution phase, all replica sets communicate among each other for migrating data. As the number of replica sets increases with cluster size, the total number of connections increases leading to network congestion. Thus, the Execution phase takes longer.

The amount of data per replica set affects reconfiguration time super-linearly. On the contrary, cluster size has a sublinear impact. In this experiment, the latter dominates as the amount of data per replica set is constant.

4.3 Parqua System Design

4.3.1 Assumptions

Unfortunately, the techniques of Morphus cannot be extended to ring-based key-value/NoSQL stores like Cassandra [79], Riak [37], Dynamo [21], and Voldemort [47]. 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 there. Thus, our optimal 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 4, enables online and efficient reconfigurations in virtual ring-based

4The Korean word for “change.”
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.

4.3.2 System Model

Parqua is applicable to any key-value/NoSQL store that satisfies the following assumptions. First, we assume a distributed key-value store that 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. 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 [59]. Fourth, 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.

4.3.3 System Design and Implementation

Figure 6: 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.

Parqua runs reconfiguration in four phases. The graphical overview of Parqua phases is shown in Fig. 6. Next, we discuss these individual phases in detail.

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.
Execute phase. The initiator node notifies all other nodes to start copying data from the Original CF to the Reconfigured CF. 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. 6, 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. To implement the SSTable 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.

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. At each node, Parqua iterates through the SSTables of Original CF to recover the entries that were written during the reconfiguration. We 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 [5], Parqua can ensure that the recovery of an entry does not overshadow newer updates, thus guaranteeing the eventual consistency.

4.3.4 Experimental Evaluation

Setup. We used the Yahoo! Cloud Service Benchmark (YCSB) [62] to generate the dataset, and used ‘uniform’, ‘zipfian’, and ‘latest’ key access distributions to generate CRUD workloads. Our default database size is 10 GB in all experiments. The operations consist of 40% reads, 40% updates, and 20% inserts.

<table>
<thead>
<tr>
<th></th>
<th>Read (%)</th>
<th>Write (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>99.17</td>
<td>-</td>
</tr>
<tr>
<td>Uniform</td>
<td>99.27</td>
<td>99.01</td>
</tr>
<tr>
<td>Latest</td>
<td>96.07</td>
<td>98.92</td>
</tr>
<tr>
<td>Zipfian</td>
<td>99.02</td>
<td>98.92</td>
</tr>
</tbody>
</table>

Table 2: Percentage of reads and writes that succeed during reconfiguration.

Availability. In this experiment, we measure the availability of our system during reconfiguration. The slight degradation in availability is due to the Commit phase when writes are rejected. The total unavailability time is a few seconds which is orders of magnitude better than the current state of the art.

The lowest availability is observed for the latest distribution. This is because YCSB does not wait for the database to acknowledge an insert of a key. As these keys are further read and updated because of the temporal nature of the distribution, the operations fail because the insert is still in progress.

5This is acceptable as Cassandra only guarantees eventual consistency.
Effect on Read Latency. Fig. 7a 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 using the Uniform key access distribution. We plot the latencies of successful reads only.

The median (50th percentile) latencies for read-only workload and the baseline are similar because they both use uniform distribution. Under reconfiguration, 20% of the reads take longer. With writes in the workload, the observed latencies for the uniform curve are higher overall.

Compared to other workloads, latest has the smallest median latency. Due to its temporal nature, recently inserted keys are present in Memtables, which is a data structure maintained in memory. As a result, reads are faster compared to other distributions which require disk accesses.

Overall, Parqua affects median read latency minimally across all the distributions. Our observations for write latency are similar. We refer the reader to our tech-report [92].

Scalability. Next, we measure how well Parqua scales with: (1) database size, (2) cluster size, (3) operation injection rate, and (4) replication factor. Due to lack of space, we omit the plots for the last two experiments and refer the reader to our tech-report [92]. To evaluate our system’s scalability, we measured the total reconfiguration times along with the time spent in each phase. We do not inject operations for the experiments presented next.

Fig. 7b depicts the reconfiguration time as the database size is increased up to 30 GB. Since we use a replication factor (number of copies of the same entry across the cluster) of 3 for fault-tolerance, 30 GB of data implies 90 GB of total data in the database. In this plot, we observe the total reconfiguration time scales linearly with database size. The bulk of the reconfiguration time is spent in transferring data in the Execute phase.

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, the same amount of data divided in smaller chunks gets transferred by a larger number of peers. Again, here the duration for the Execute phase dominated the reconfiguration time.

4.4 Related Work

Research in distributed databases has focused on query optimization and load-balancing [58], and orthogonally on using group communication for online reconfiguration [75], however, they do not solve the core algorithmic problems for efficient reconfiguration. Online schema change was targeted in [87], but the resultant availabilities were lower than those provided by Morphus and Parqua. In a parallel work, Elmore et. al. [67] have looked into the reconfiguration problem for a partitioned main memory database like H-Store. Data placement in parallel databases have used hash-based and range-based partitioning [63, 83], but they do not target optimality for reconfiguration.

The problem of live migration has been looked into in the context of databases. Albatross [66], Zephyr [68] and ShuttleDB [53] addresses live migration in multi-tenant transactional databases. Albatross and ShuttleDB uses iterative operation replay like Morphus, while Zephyr routes updates based on current data locations. Data migration in these systems happen between two different sets of servers while Morphus and Parqua achieve this inside the same replica sets. Also, they do not propose optimal solutions for any reconfiguration operation. Opportunistic lazy migration explored in Relational Cloud [65] entails longer completion
times. Tuba [51] looked into the problem of migration in a geo-replicated setting. They avoided write throttle by having multiple masters at the same time which is not supported by MongoDB and Cassandra.

Morphus’ techniques naturally bear some similarities with live VM migration. Pre-copy techniques migrate a VM without stopping the OS, and if this fails then the OS is stopped [61]. Like pre-copy, Morphus also replays operations that occurred during the migration. Pre-copy systems also use write throttling [54]. Pre-copy has been used in database migration [52].

For network flow scheduling, Chowdhury et.al [60] proposed a weighted flow scheduling which allocates multiple TCP connections to each flow to minimize migration time. Our WFS approach improves their approach by additionally considering network latencies. Morphus’ performance is likely to improve further if we also consider bandwidth. Hedera [50] also provides a dynamic flow scheduling algorithm for multi-rooted network topology. Even though these techniques may improve reconfiguration time, Morphus’ approach is end-to-end and is less likely to disrupt normal reads and writes which use the same network links.

5 Compaction

5.1 Motivation

NoSQL databases optimized for write-heavy workloads often use log structured storage for performance. A given server stores multiple keys. At that server, writes are quickly logged (via appends) to an in-memory data structure called a memtable. When the memtable becomes old or large, its contents are sorted by key and flushed to disk. This resulting table, stored on disk, is called an sstable.

Over time, at a server, multiple sstables get generated. Thus, a typical read path may contact multiple sstables, making disk I/O a bottleneck for reads. As a result, reads are slower than writes in NoSQL databases. To make reads faster, each server in a NoSQL system periodically runs a compaction protocol in the background. Compaction merges multiple sstables into a single sstable by merge-sorting the keys.

In order to minimally affect normal database CRUD (create, read, update, delete) operations in these systems, sstables are merged in iterations. A compaction strategy identifies the best candidate sstables to merge during each iteration. To improve read latency, an efficient compaction strategy needs to minimize the compaction running time. Compaction is I/O-bound because sstables need to be read from and written to disk. Thus, to reduce the compaction running time, an optimal compaction strategy should minimize the amount of disk bound data. In this work, we have the following contributions:

• Prove compaction problem as NP-Hard
• Propose some compaction strategies with approximation bounds
• Simulate the strategies with real life workload to evaluate performance.

5.2 Solution Outline

In this work, we formulate this compaction strategy as an optimization problem. Given a collection of $n$ sstables, $S_1, \ldots, S_n$, which contain keys from a set, $U$, a compaction strategy creates a merge schedule. A merge schedule defines a sequence of sstable merge operations that reduces the initial $n$ sstables into one final sstable containing all keys in $U$. Each merge operation reads atmost $k$ sstables from disk and writes the merged sstable back to disk ($k$ is fixed and given). The total disk I/O cost for a single merge operation is thus equal to the sum of the size of the input sstables (that are read from disk) and the merged sstable (that is written to disk). The total cost of a merge schedule is the sum of the cost over all the merge operations in the schedule. An optimal merge schedule minimizes this cost.

We propose the 4 heuristics. Theoretically, we show that the results generated by the heuristics are $O(\log n)$ approximation. In practice with real-life workloads however, we showed through simulation that the results were much closer to optimal.
5.3 Related Work

A practical implementation of compaction was first proposed in Bigtable [59]. It merges sstables when their number reaches a pre-defined threshold. They do not optimize for disk I/O. For read-heavy workloads, running compaction over multiple iterations is slow in achieving the desired read throughput. To solve this, Level-based compaction [31, 33] merges every insert, update and delete operation instead. They optimize for read performance by sacrificing writes. NoSQL databases like Cassandra [10] and Riak [37] implement both these strategies [32, 40]. Cassandra’s Size-Tiered compaction strategy [40], inspired from Google’s Bigtable, merges sstables of equal size. This approach bears resemblance to one of our proposed heuristic. For data which becomes immutable over time, such as logs, recent data is prioritized for compaction [19, 94]. Again, the goal here is to improve read throughput.

Our work looks at a major compaction operation. Mathieu et. al. [81] have also theoretically looked at compaction, however they focused on minor compaction and their problem is thus different from ours. The memtable and a subset of sstables are compacted at periodic intervals, and the resultant number of sstables left after each interval is bounded from above. An optimal merge schedule specifies the number of sstables to merge in an interval given the cardinality of current sstables and the memtable. On the contrary, in our case of major compaction, we merge all sstables at once by choosing a fixed number of sstables to merge in an iteration. Our goal is to create a single sstable at the end of the compaction run.

5.4 Open Problems

Many interesting theoretical questions still remain. The $O(\log n)$ approximation bound shown for the heuristic seems quite pessimistic. Under real-life workloads, the algorithms perform far better than $O(\log n)$. We do not know of any bad example for these two heuristics showing that the $O(\log n)$ bound is tight. This naturally motivates the question, if the right approximation bound is infact $O(1)$. Finally, it will be interesting to study the hardness of approximation for the BINARYMERGING problem.

6 Adaptive Replication

6.1 Real Time Analytics

6.1.1 Motivation

Data warehousing solutions have been providing business intelligence to companies from late 1980s [17] and it is expected to become a 20 billion dollar business by 2020 [18]. The range of solutions vary from traditional RDBMS provided by IBM [30] and HP [28] to Hadoop [6] provided by small scale startups like Cloudera [12], Splunk [41], etc. Queries can vary from simple aggregation to complex data mining. Aggregation queries are frequently used to calculate information like linkedin connections, facebook likes etc. They should change in real-time as new friends are made on Facebook and new connections in LinkedIn. Hadoop’s overheads makes it unsuitable for such time-sensitive application.

Google’s Dremel [84] was the first solution in this space and it inspired other systems like Impala [13], Druid [20]. To process large amount of data in real time, all of these systems use a high degree of CPU and I/O parallelism. To exploit parallelism, the large data is broken down into segments which align with time boundaries. Aggregate queries are then split up and routed to nodes having the relevant time segments. The aggregate results from each of the nodes are then further aggregated before returning to the client. For high throughput, inter-query parallelism is exploited by replicating the individual segments.

Currently systems like Druid use a tiered replication strategy. Each tier represents data popularity and has its own fixed replication factor. This is wasteful as unpopular batches take up disk space which could have been better utilized by more popular batches. Further, these tiers are defined by users manually. The key idea of Getafix is to improve system throughput by adaptively replicating segments based on popularity.

Thus, in this work, we have the following contributions:

1. We model the problem of replica allocation and provide a formal proof on the best allocation strategy which will minimize the total number of replicas while still giving the best throughput.
2. We propose Getafix, a system which automatically monitors the batch popularity to decide its replication factor.

3. We implement Getafix inside Druid and evaluate it with real life workloads.

6.1.2 Solution Outline

Model: The problem of replica allocation can be modelled as a modified bin packing problem [8]. We have \( n \) queries and \( m \) compute nodes. Each query \( i \) requires a set \( Q_i \) of data segments to compute aggregate on. Segments represent objects while compute nodes are bins in this problem. A single slot corresponds to the time taken to execute a query on one segment of data. Since we want to maximize throughput, and therefore minimize the overall completion time for \( n \) queries, we can easily calculate the number of slots required per bin, \( \left\lceil \frac{\sum_{i=1}^{n} |Q_i|}{m} \right\rceil \). It can be shown that best fit decreasing [7] algorithm minimizes the number of replicas required in this setting.

System Design: We implement our system, Getafix on top of a popular real time analytics store called, Druid. It has two main components:

- **Segment Popularity Tracker:** We plan to use a data structure to keep a running count of segment accesses. To maintain recent counts only, the datastructure will also age out old segment accesses. Periodically, we will use this datastructure to define the number of replicas for each segment.

- **Replica Management:** This module will refer the datastructure to perform the following actions: 1) add a segment replica to a compute node, 2) remove replicas of obsoleted segments.

6.1.3 Current Status:

We have completed:

1. Theoretical Formulation and Optimality Proof.
2. Ran proof of concept simulation to estimate the benefits of an adaptive replication scheme.
3. Have a basic design for our system Getafix on top of Druid.

We plan to complete:

1. The adaptive replication algorithm to find out the replication factor from the number of segment accesses.
2. Implement Getafix.
3. Run evaluation using real life workloads.

6.2 Actor Framework

6.2.1 Motivation

Cloud computing solutions have been widely adopted for solving problems at scale. Inspite of that, design of such a system is still considered the job of an expert. This is because parallel programs are notorious for race conditions that require knowledge of sophisticated concurrency control mechanisms that ultimately hurt scalability. Actor model provides a shared nothing abstraction for writing program logic thereby, providing good scalability. Actors encapsulate state and logic and talk to other actors through message passing. A programmer is entrusted with writing the semantic while the runtime takes care of deployment, instantiation, persistence, etc. The ease of use has made actor Frameworks like Erlang [23], Akka [3], Orleans [57] really popular.

A typical example application which would use an actor framework would be a web service like Amazon. In this example, shopping cart can be modelled as an actor. You can add items to the cart and also browse them. This is an example of a stateful actor where the stored items represent state. The system should scale with the number of web sessions. Actor frameworks like Orleans naively scale as read/write load increases.
Unfortunately the read write performance suffers as a result. Under heavy write volume, having a large number of actor instances can affect response time because of the reconciliation overheads. Conversely when there is a large read volume, too few instances can be a scalability bottleneck.

The key idea of this project is to explore an adaptive strategy for scaling stateful actors which can adjust as read/write proportion changes. We will explore the tradeoff with scalability and performance through new algorithms for scaling and reconciliation. We plan to incorporate the algorithms in Orleans, the actor framework from Microsoft Research.

6.3 Related Work

Adaptive Replication has been explored as a solution for problems in multiple research areas. In databases, systems adapt the number of replicas based on number of reads and writes [95]. In programming language design, synchronization bottlenecks [89] can be mitigated by reducing replication for the object under contention. Storage systems like Facebook’s f4 [85] use adaptive replication for “warm” BLOB data like photos, videos, etc.

7 Network Awareness

7.1 Graph Processing in Supercomputer

7.1.1 Introduction

Supercomputers have traditionally been used for scientific computing jobs [9] in biophysics, climate sciences, astronomy, etc. These are batch compute intensive jobs some time requiring multiple days and weeks to run. Despite their popularity in these areas, they are mostly ignored for the data intensive cloud computing applications. This is primarily because of two reasons – architecture and cost.

Unlike datacenter, compute nodes in supercomputers do not typically have disks attached with them. Storage is handled by a separate set of nodes which have a parallel file system like LustreFS [34] deployed in them. Even though, the aggregated I/O bandwidth supported is close to 1 TB/s, interactive applications may suffer from occasional latency spikes due to internal batching of access requests. To amortize the file system overheads, newer supercomputers like Gordon and Comet [39] attach SSDs to compute nodes. This brings supercomputer architecture even closer to datacenters making them amenable for high throughput cloud computing applications like graph and stream processing.

Titan, a powerful supercomputer at Oak Ridge National Laboratory, cost $97 million [44] which is atleast 100x more than setting up a datacenter. This is a huge deterrent to companies trying to optimize cost and performance. For supercomputers to become a viable alternative the performance gain has to be compelling. Thus, the goal of this work is to quantify the maximum performance gain that we can hope to get by deploying an optimized cloud computing system on a supercomputer.

We believe supercomputer’s 3D Torus architecture [45] can be leveraged by graph and stream processing systems. In both these systems, we embed the network communication graph into the underlying topology. Typical datacenters use tree topology which allows lesser freedom in embedding as opposed to the torus. Further in supercomputers, network bandwidth also decreases as hops increase between two nodes in the torus. Thus, a good embedding of the communication graph will minimize the aggregate hop distance. In this work, we modify Powergraph’s [73] partitioning algorithm to minimize the aggregate hop distance of its communication graph.

7.1.2 Solution Outline

Figure 8 shows a typical supercomputer architecture. Compute nodes are connected using direct links in a tight physical space in order to minimize network latency. The network bandwidth decreases as hops increase. Typically large graphs are partitioned to parallelize computation. An optimal partitioning algorithm tries to minimize the network communication required while ensuring load balance among compute nodes. Currently, these algorithms are not network aware, in the sense, the partitions may communicate along low bandwidth network links.
The key idea of this project is embedding this communication graph on top of the network topology in a way that reduces communication overhead, thus shortening iteration running time. Powergraph uses minimum vertex cuts to partition a large graph. The communication graph in Powergraph connects replicated vertices of the actual graph. At the end of each iteration, the replicas send their computed values to a master which aggregates and sends them back.

We have implemented a modified partitioning algorithm inside Powergraph. We are in the process of evaluating the performance gain by deploying the system in Bluewaters [9].

7.1.3 Related Work

Accumulo [2], an open source implementation of Bigtable [59] has been benchmarked [76] on MIT Super-Cloud [88]. A peak performance of 100,000,000 database inserts per second was observed. NoVoHT [55], a new key-value store specifically designed for supercomputers claims improved performance over LevelDB [31], a traditional key-value store. These are few example applications which further confirms our belief that a well designed cloud computing system optimized for supercomputers can outperform traditional systems making them a compelling alternative.

In the recent past, graph processing has received significant attention from the high performance community. GraphReduce [90] proposes to use GPUs in a scalable way for large scale computation. Exploring better partitioning strategies optimized for supercomputers have also been looked into by LeBeane et. al. [80]. Other works have focused on network-level optimizations [74, 86] to improve performance.

8 Execution Plan

The first half of the work mentioned in this document has been published, and for the rest we will be following this approximate timeline:

- Getafix system is currently a work-in-progress and we estimate to complete it in 4-6 months and submit to a conference like SOCC.
- The work in actor framework project is still in the ideation phase. We estimate to complete it in 6-9 months and submit to a conference like SIGMOD.
- We estimate 6-9 months for the bluewaters project too with the research suitable for publication in venues like ICDCS.
9 Summary

In this proposal, we have discussed projects (completed and future work) to support the central hypothesis that *efficient implementation of new and existing background mechanisms can help in performance optimization of data-intensive cloud systems*. In Morphus and Parqua, we have shown that reconfiguration which has been a persistent pain point in industry, can be done in an efficient way without taking the database offline. Our compaction work minimizes disk I/O and thereby improves completion time though the use of new heuristics with theoretical guarantees. The other proposed future work also look at new and existing background mechanisms which can potentially improve frontend performance.

References


