© 2018 Mainak Ghosh

EFFICIENT DATA RECONFIGURATION FOR TODAY'S CLOUD SYSTEMS

BY

MAINAK GHOSH

DISSERTATION
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Indranil Gupta, Chair
Professor Nitin Vaidya
Professor Luke Olson
Assistant Professor Aaron Elmore

Abstract

Performance of big data systems largely relies on efficient data reconfiguration techniques. Data reconfiguration operations deal with changing configuration parameters that affect data layout in a system. They could be user-initiated like changing shard key, block size in NoSQL databases, or system-initiated like changing replication in distributed interactive analytics engine. Current data reconfiguration schemes are heuristics at best and often do not scale well as data volume grows. As a result, system performance suffers.

In this thesis, we show that *data reconfiguration mechanisms can be done in the background by using new optimal or near-optimal algorithms coupling them with performant system designs*. We explore four different data reconfiguration operations affecting three popular types of systems – storage, real-time analytics and batch analytics. In NoSQL databases (storage), we explore new strategies for changing table-level configuration and for compaction as they improve read/write latencies. In distributed interactive analytics engines, a good replication algorithm can save costs by judiciously using memory that is sufficient to provide the highest throughput and low latency for queries. Finally, in batch processing systems, we explore prefetching and caching strategies that can improve the number of production jobs meeting their SLOs. All these operations happen in the background without affecting the fast path.

Our contributions in each of the problems are two-fold – 1) we model the problem and design algorithms inspired from well-known theoretical abstractions, 2) we design and build a system on top of popular open source systems used in companies today. Finally, using real-life workloads, we evaluate the efficacy of our solutions. Morphus and Parqua provide several 9s of availability while changing table level configuration parameters in databases. By halving memory usage in distributed interactive analytics engine, Getafix reduces cost of deploying the system by 10 million dollars annually and improves query throughput. We are the first to model the problem of compaction and provide formal bounds on their runtime. Finally, NetCachier helps 30% more production jobs to meet their SLOs compared to existing state-of-the-art.

I dedicate my thesis to my family

Acknowledgments

At the very start, I would like to thank my advisor, Professor Indranil Gupta (Indy). Over the years, he has advised me on challenging problems to work on, books to read, food to try, travel destinations. The fact that I could talk to him on a wide variety of topics (work and non-work) helped me a lot in settling down in this place away from home. Thanks Indy for guiding me through this long journey.

I would also like to thank my thesis committee members, Professor Nitin Vaidya, Professor Luke Olson and Professor Aaron Elmore. Your feedback has been invaluable and has helped in improving the thesis. Thanks to all the funding agencies who have supported my work. I would also like to thank the department for giving me this opportunity to pursue Ph.D. in this great institute. Thanks to all the professors whose courses I had the privilege to take and learn from. Thanks to all the admins for your tireless work to ensure a smooth graduate life here for me.

I have actively collaborated with researchers from industry and academia in my projects. I would like to take this opportunity to thank, Wenting Wang, Gopalakrishna Holla, Yosub Shin, Ashwini Raina, Le Xu, Xiaoyao Qian, Himanshu Gupta, Shalmoli Gupta, Nirman Kumar, Professor Chandra Chekuri, Professor Aaron Elmore, Virajith Jalaparti, Chris Douglas, Ashvin Agrawal, Avrilia Floratou, Ishai Menache, Joseph (Seffi) Naor, Sriram Rao, Kaushik Rajan. I cherish the stimulating discussions we had on the different projects we worked together. It was a learning experience and has helped me immensely in not only understanding the problem at hand but also appreciate the big picture.

I have had the opportunity to make some wonderful friends in Urbana-Champaign. I have had the pleasure of your company in restaurants, movies, travels, random discussions, celebrations. Thank you Sourabh, Ankita, Swarnali, Srijan, Sreeradha, Debapriya, Sangeetha, Subhro, Faria, Le, Muntasir. I greatly value our time spent. A big shout out to the vibrant communities I got to engage with – Distributed Protocols Research Group (DPRG), Bengali Student's Association (BSA), East Central Illinois Bengali Association (ECIBA) and ASHA for Education.

Next, I would like to thank my mother, Kakali Ghosh and my father, Mrinmoy Kumar Ghosh. You have cheered at all my milestones and helped me get up when I was feeling low. Thank you for teaching me to value relationships, family, friends, objects, and life itself. I try my best to follow your footsteps and be the person that you wanted me to be. At different points in my growing up years, you chose to let go of smaller pleasures in your life to help me succeed in my career and life. Pursuing a Ph.D. in the U.S.A. was one such moment

when you let me go despite realizing how far I would have to move. I am sorry that you had to sacrifice so much. I am not sure if I can ever make it up to you. Thank you for being the best parents.

Finally, I would like to thank my partner, my confidant, my best friend, Shalmoli. I am deeply indebted to the University for bringing us together. Shalmoli, you are the most amazing person I have met. In the last few years, you have made the highs more special and lows more bearable. Your enthusiasm and exuberance on different facets of life inspires me. Thanks for being the special person of my life.

Table of Contents

Chapter 1 Introduction 1
1.1 Background 1
1.2 Challenges 3
1.3 Contributions 5
1.4 Roadmap 6

Chapter 2 Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems 8
2.1 Introduction 8
2.2 System Design 10
2.3 Algorithms for Efficient Shard Key Reconfigurations 15
2.4 Network-Awareness 18
2.5 Evaluation 20
2.6 Related Work 30
2.7 Summary 31

Chapter 3 Parqua: Online Reconfigurations in Virtual Ring-Based NoSQL Systems 32
3.1 Introduction 32
3.2 System Model & Background 34
3.3 System Design and Implementation 35
3.4 Experimental Evaluation 38
3.5 Summary 50

Chapter 4 Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines 51
4.1 Introduction 51
4.2 Background 54
4.3 Static Version of Segment Replication Problem 56
4.4 Getafix: System Design 62
4.5 Evaluation 68
4.6 Discussion 77
4.7 Related Work 78
4.8 Summary 79

Chapter 5 Fast Compaction Algorithms for NoSQL Databases 80
5.1 Introduction 80
5.2 Problem Definition 82
5.3 Greedy Heuristics for BinaryMerging 84
5.4 Simulation Results 90
5.5 Conclusion 95

Chapter 6 Joint Network and Cache Management for Tiered Architectures 96
6.1 Introduction 96
6.2 Motivation 99
6.3 System Architecture 102
6.4 Implementation 105
6.5 Evaluation 108
6.6 Related Work 114
6.7 Conclusion 115

Chapter 7 Conclusion and Future Work 116

References 119

Chapter 1: Introduction

1.1 BACKGROUND

Data driven business intelligence (BI) has seen a steady adoption [1] across multiple sectors (telecommunication, financial services, healthcare, technology, etc) and among corporations, governments, non-profits, alike. Early BI tools were primitive and often required manual sifting through multiple long spreadsheets. One of the key challenges was lack of software and hardware capable of handling large volumes of data. The last decade has seen tremendous growth in both areas which has enabled organization to make business critical decisions in real time. Today, big data has impacted gene sequencing for faster drug discovery, making smart energy efficient homes, self driving cars, solving food wastage, help farming with precision tools, etc.

Tools for data driven business intelligence can be broadly classified into two categories – storage and computation. While the former is required to store data generated in a company, the latter is used for running analysis on the stored data. Computation systems are designed to be massively parallel, fault tolerant, handle stragglers, etc. Storage systems provide high availability, fault tolerance, consistency.

Google's Mapreduce [2] was one of the early computation systems that has inspired today's systems like Hadoop [3] and Spark [4]. These systems are popularly called batch processing systems. They handle large amount of data at rest and can run analytics jobs which can span from few hours to few days. One of the first open-source batch processing systems, Hadoop captures 21% [5] of the total analytics markets with 14K companies using and in some cases actively contributing to the project. Spark is an academic project from UC Berkeley which has henceforth been commercialized and has seen steady adoption in internet companies like Facebook, Twitter, etc.

Along with Mapreduce, Google also built storage systems like Bigtable [6] and Google File System [7] which stored petabytes of data across many servers. Later, GFS inspired open source development of HDFS [8]. Similarly, Bigtable and Dynamo [9], a storage system from Amazon around the same time, inspired some of the present day open source NoSQL databases like HBase [8], Cassandra [10], MongoDB [11] etc. HDFS is the de facto standard for companies storage needs and works as the storage substrate for the popular batch processing system Hadoop. NoSQL databases have been adopted in internet companies like Facebook, Google, telecommunication companies like Verizon, retail chains like Walmart, Amazon, etc.

Apart from batch analytics, there is an increased focus on analysis in real-time. Recent

studies have also shown that lack of realtime responses in ecommerce sites, can lead to reduced profitability anywhere between 25% to 125% [12]. Hadoop and Spark do not satisfy these requirements as they were not built for response latencies in the order of seconds. The real time analytics space has developed under two verticals – streaming analytics and distributed OLAP. Systems like Samza [13], Heron [14] fall into the streaming space while Apache Kylin [15], SQL Server [16] are some examples of OLAP systems. In the intersection lies distributed interactive analytics engine. Some popular systems in this space are Druid [17], Presto [18]. This area is expected to grow to 13.7 billion dollars by 2021 [19]. Companies currently maintain separate pipelines for batch and realtime analytics.

In the next decade, we expect data volumes to reach hundreds of zettabytes [20, 21]. Simultaneously, companies will continue to add newer pipelines to satisfy specialized application use-cases. Machine learning systems [22] are an example. Often many of these pipelines end up sharing the same data. Data Lakes [23] have seen adoption because they allow easy data sharing as well as independent scale out capabilities.

One of the major challenges in storage in this upcoming era is managing data reconfiguration operations at scale. Reconfiguration operations handle how data is laid out in a cluster. For example, shard key in a NoSQL database is a popular configuration that defines how to partition data in a table. Changing shard key is a reconfiguration operation that involves repartitioning and reassigning data to different physical machines. Reconfiguration operations are necessary in a system to improve performance. For example, changing shard key helps in improving read/write latency in NoSQL databases. They can be user-initiated like a sys-admin in a company or system initiated. They are commonly used in both compute and storage systems. Since reconfiguration operations change properties associated with data, these operations are I/O bound, often saturating network or disk.

Type of System	Reconfiguration	Performance Metric Affected	Applicable to	Chapter
Storage	Table-level Configuration Change	Read/Write Latency	NoSQL Databases like MongoDB and Cassadra	Chapter 2 Chapter 3
	Compaction	Read Latency	NoSQL Databases like Cassandra	Chapter 5
Real Time Analytics	Replication	Query Throughput	Distributed Interactive Analytics Engine like Druid	Chapter 4
Batch Analytics	Prefetching and Caching	Meeting Job SLOs	Distributed File System like HDFS	Chapter 6

Table 1.1: Reconfiguration Operations Explored in this Thesis. We picked operations that straddle different systems from storage to real-time analytics to batch processing system. All of them affect a key system performance metric.

Table 1.1 lists four popular data reconfiguration operations commonly seen in storage and computation systems of today and the performance metrics they affect. We observe that current state-of-the-art solutions for these operations are heuristic at best and often involve manual work from a sys-admin. In our thesis, we claim *data reconfiguration mechanisms can be done in the background by using new optimal or near-optimal algorithms coupled with performant system designs.*.

For each of the reconfiguration operations mentioned in Table 1.1, we make the following two-fold contributions – 1) we model the problem and design algorithms inspired from well-known theoretical abstractions, 2) we design and build a system on top of popular open source systems used in companies today. We have the following goals for system design – 1) reconfiguration should happen in the background without affecting the normal query path, 2) minimizing total data transfer required, 3) minimizing impact on query latency. Finally, using real-life workloads, we evaluate the efficacy of our solutions. Our evaluation results vindicate our thesis and shows that one can build efficient data reconfiguration strategies by merging theory with practical design principles.

In Section 1.2, we describe the challenges involved in each of the reconfiguration operation listed in Table 1.1. In Section 1.3, we outline the contributions we made to solve these challenges.

1.2 CHALLENGES

1.2.1 Changing table-level configuration in NoSQL Databases

Distributed NoSQL databases like MongoDB [11], HBase [24], Cassandra [10] use configurations like shard key, block size, etc. at the level of a table to ensure good read and write performance. Choosing good configuration parameter is highly workload dependent. For example, if your workload constantly updates an inventory table by the product group id (all hand towels fall under the same group), it should be selected as a shard key. If at a later point, you decide to update individual products (hand towel brands) separately, changing the shard key to product id is prudent from performance point of view. Currently, reconfiguration in these databases is done by a sys-admin and involves taking the table offline, and manually exporting and re-importing the data to a new table with an updated schema. This brute force approach has a high network overhead when the data size is large (in the order of zettabytes). It will slow the reconfiguration down and end up hurting data availability, one of the key selling points for these systems.

1.2.2 Selective Replication in Distributed Interactive Analytics Engines

Companies are increasing running their batch processing pipelines on top of a tiered architecture. Data resides in a lake like S3 [25] often shared by multiple internal organizations. Batch jobs with strict SLOs are submitted to frontend Yarn [26] cluster. Network bandwidth connecting the backend and frontend cluster in public clouds like AWS, Azure is limited by provider enforced limits and highly variable [27]. Further the front end storage space is orders of magnitude smaller than the backend storage. Naive fetching of data during job execution in such a setting can lead to SLO violations. This in turn leads to significant financial losses. State-of-the-art caching algorithms [28] can help as lot of jobs share the same data in our representative workload. Unfortunately, we see low correlation between the number of jobs that read a file and the rate at which it is read. Caching algorithms account primarily for access count and are oblivious to network bandwidth considerations. Using prefetching with caching can help alleviate these scenarios. This solution requires solving a joint optimization problem with limits on network and storage.

1.2.3 Compaction in Log Structured Databases

In write-optimized databases like Cassandra [10], Riak [29], Bigtable [6] data is stored in a log structured file system. Writes are fast in these systems because delta updates are appended to a log, sorted string table (SSTable). Reads have to merge these delta updates to materialize the entry corresponding to the key. This is typically slow as data must be read across multiple files residing in slower disks. Compaction is the process of proactively and periodically merging SSTables to improve read performance. Compaction is I/O bound as it reads and writes files to disk. Performance of these algorithms is critical to improve read performance in log structured databases. Current compaction algorithm are heuristics at best with little formal analysis. As data sizes grow, these algorithms will soon become a bottleneck and lack of a formal understanding will hamper future efforts for improvement.

1.2.4 Prefetching and Caching in Batch Analytics

While the last two operations dealt with storage system, the next two handles reconfiguration in analytics system. Real-time analytics is an exciting new space that is defined by fast real-time data ingestion, small query response time (in the order of seconds) and high throughput. Distributed interactive analytics engines like Druid [17] and Pinot [30] are popular in this space. These systems are massively parallel. Data is replicated to reduce query hotspots. Further, for fast query response time, in memory computation is preferred

over out-of-core options like SSDs. A good replication strategy is thus characterized by high memory utilization. State-of-the-art replication schemes [31] use heuristics that can over-replicate, often adversely affecting performance. Blindly reducing replication will also impact performance. We need a replication scheme that can find the "knee" of the curve in terms of replication.

1.3 CONTRIBUTIONS

In this thesis, we make the following contributions:

1.3.1 Table-level Configuration Change with Morphus and Parqua

Network traffic volume directly affects reconfiguration completion time in NoSQL databases. We model this as the well-known bipartite matching problem and provably show that minimum cost matching is optimal in minimizing the amount of data transfer required. Our Morphus system uses these algorithmic insights to build a system which can perform data migration with little impact on availability. While Morphus works on a database which uses range partitioning, Parqua solves the problem in a hash partitioned system. Our experiment with real world datasets show that both Morphus and Parqua provides several 9s of availability even when reconfiguration is in progress.

Morphus has been published in proceedings of ICAC 2015 [32] and as journal in a special issue of IEEE Transactions on Emerging Topics in Computing [33]. The Parqua system has been accepted as a short paper in the proceedings of ICCAC 2015 [34].

1.3.2 Selective Replication using Getafix

In distributed interactive analytics engines, increasing data replication has a diminishing returns in terms of query performance gain for a given workload and cluster size. Finding the right replication factor which does not hurt performance is an important problem in these systems. Replication is directly correlated with the amount of memory needed in the frontend cluster which impacts deployment cost. We model the problem of query scheduling in distributed interactive analytics engines as a colored variant of the well known balls and bins problem. Under simplifying assumptions, we provably show that a popular bin packing heuristic, best fit, can minimize the amount of replication required without hurting performance. Deriving insights from this theory, we built our system, Getafix which dynamically changes data replication in the background as queries are coming in. In our evaluation,

Getafix reduces memory required by 2.15x compared to Scarlett, a popular replication strategy used today.

The work has been published in the proceedings of Eurosys 2018 [35].

1.3.3 Compaction in Log Structured Databases

Compaction in log structured databases is a popular technique to improve read performance. Considering its importance, very little analysis exists on how good the present day heuristics are. We know compaction is I/O bound as it merges files residing in disks. We model compaction as a set merge problem where cost of a merge operation is directly proportional to the number of keys read and written to disk after the merge. This problem turns out to be a generalization of well known Huffman Coding problem. Based on this insight, we devise heuristics inspired from the optimal greedy solution for Huffman coding. We prove $O(\log n)$ approximation bounds for all the heuristics. Further, through simulation using realistic workloads we show that the worst case bound does not happen in practice and the heuristics perform much better.

The work has been published in the proceedings of ICDCS 2015 [36] (Theory track).

1.3.4 Prefetching and Caching in Batch Systems with NetCachier

Variable network bandwidth between compute and backend storage in public clouds severely hampers jobs in production clusters from meeting their SLOs. Since productions jobs are periodic in nature, prefetching the input files for them before the job starts can help them meet their SLOs. Limits on frontend cache space and network bandwidth means one must solve a joint optimization problem such that the number of jobs meeting their SLOs is maximized. As it turns out this problem is NP-Hard and even hard to approximate. Thus, we choose to use simple heuristics to build NetCachier. Given a workload of jobs which is known upfront, NetCachier picks while files to prefetch, how much bandwidth to allocate while fetching and whether to cache it in the compute tier for later use or not. Netcachier improves number of jobs meeting SLOs by 80% compared to naive strategy used today.

The work is currently under submission.

1.4 ROADMAP

The dissertation is organized as follows: Chapter 2 and Chapter 3 discusses the Morphus and Parqua systems respectively. In these chapters, we discuss the algorithms and

techniques used to solve shard key change based reconfigurations in a range based partitioning (MongoDB [11]) and a hash based partitioning system (Cassandra [10]) respectively. Workload-aware dynamic replication strategies in our Getafix system is explained in Chapter 4. Chapter 5 presents our results on better compaction algorithms which can be used in write-optimized databases. We show how one can meet more job SLOs using smart prefetching algorithms in our NetCachier system in Chapter 6. Finally, we end with our conclusion and some future directions of work in Chapter 7.

Chapter 2: Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems

In this chapter, we discuss the problem of changing configuration parameters like shard key, block size etc. at the level of a database table in a sharded NoSQL distributed database like MongoDB. We motivate the problem in Section 2.1. We highlight our system techniques in Section 2.2 and algorithmic contributions in Section 2.3. Since reconfiguration involves significant amount of data transfer, Morphus uses network level optimizations described in Section 2.4. Finally, we present results from our evaluation in Section 2.5.

2.1 INTRODUCTION

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 [37].

In production deployments of NoSQL systems and key-value stores, *reconfiguration operations* have been a persistent and major pain point [38, 39]. (Even traditional databases have considered reconfigurations to be a major problem over the past two decades [40, 41].) 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 which is used to split a table into blocks, or changing the block size itself, or increasing or decreasing the cluster size (scale out/in).

In today's sharded NoSQL deployments [24, 11, 6, 42], such 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 [43]. 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 [44, 45, 46]. This approach does not support concurrent reads and writes during migration, a feature we

[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.

would like to provide.

Consider an admin who wishes to change the shard key inside a sharded NoSQL store like MongoDB [43]. 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.

As a result, many admins start their databases with a system-generated UID as the shard key, while others hedge their bets by inserting a surrogate key with each record so that it can be later used as the new primary key. The former UID approach reduces the utility of the primary key to human users and restricts query expressiveness, while the latter approach would work only with a good guess for the surrogate key that holds over many years of operation. In either case, as the workload patterns become clearer over a longer period of operation, a new application-specific shard key (e.g., username, blog URL, etc.) may become more ideal than the originally-chosen shard key.

Broadly speaking, 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. Failure to reconfigure databases can lower performance, and also lead to outages such as the one at Foursquare [47]. As a result, the reconfiguration problem has been a fervent point of discussion in the community for many years [48, 49]. 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., [38]), discussions and blogs (e.g., [49]). Inside Google, resharding MySQL databases took two years and involved a dozen teams [50]. 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.

In this chapter, we present a system that supports automated reconfiguration. Our system, called Morphus, allows 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.

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 [11], RethinkDB [51], CouchDB [52], 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 later present 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 chapter 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.

We solve these challenges via these approaches:

- Morphus performs automated reconfiguration among cluster machines, without needing additional machines.
- Morphus concurrently allows efficient read and write operations on the data that is being migrated.
- To minimize data migration volume, Morphus uses new algorithms for placement of new data blocks at servers.
- Morphus is topology-aware by using new network-aware techniques which adapt data migration flow as a function of the underlying network latencies.
- We integrate Morphus into MongoDB [43] focusing on the shard key change operation.
- Our experiments using real datasets and workloads show that Morphus maintains several 9s of availability during reconfigurations, keeps read and write latencies low, and scales well with both cluster and data sizes.

2.2 SYSTEM DESIGN

We now describe the design of our Morphus system.

[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 [42] require a different solution, which is the target of a different project of ours.

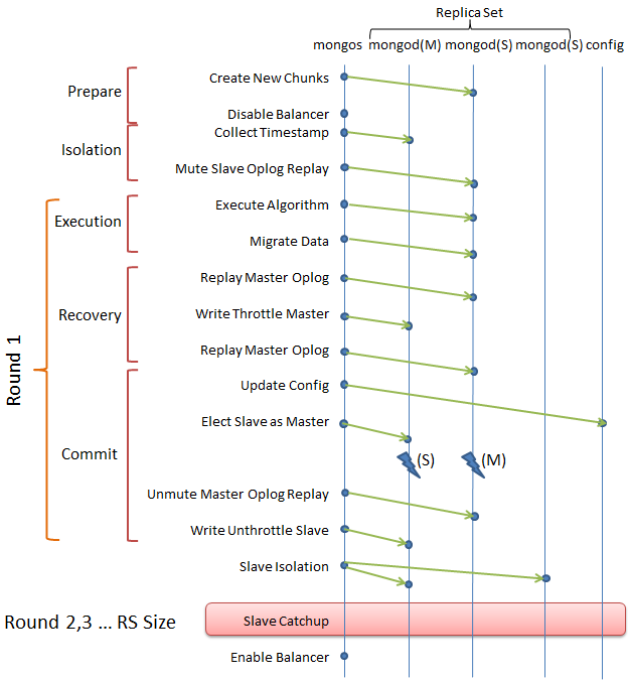


Figure 2.1: Morphus Phases. Arrows represent RPCs. M stands for Master, S for Slave.

2.2.1 MongoDB System Model

We have chosen to incorporate Morphus into a popular sharded key-value store, MongoDB [11] 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.

2.2.2 Reconfiguration Phases in Morphus

Section 2.3 will describe how new chunks can be allocated to servers in an optimal way. Given such an allocation, we now describe how to i) migrate data, while ii) concurrently supporting operations on this data.

Overview: 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 3.1.

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.

Next we discuss the individual phases in detail.

Prepare: The first phase is the Prepare phase, which runs at the mongos front-end. Reads and writes are not affected in this phase, and can continue normally. Concretely, for the shard key change reconfiguration, there are two important steps in this phase:

- **Create New Chunks:** Morphus queries one of the mongod servers and sets split points for new chunks by using MongoDB's internal splitting algorithm (for modularity).
- **Disable Background Processes:** We disable background processes of the NoSQL system which may interfere with the reconfiguration transfer. This includes the MongoDB Balancer, a background thread that periodically checks and balances the number of chunks across replica sets.

Isolation: In order to continue serving operations while the data is being reconfigured, Morphus first performs reconfiguration transfers only among secondary servers, one from each replica set. It prepares by performing two steps:

- **Mute Slave Oplog Replay:** Normally, the primary server forwards the operation log (called *oplog*) of all the write operations it receives to the secondary, which then replays it. In the Isolation phase, this oplog replay is disabled at the selected secondaries, but only for the collection being reconfigured – other collections still perform oplog replay.

We chose to keep the secondaries isolated, rather than removed, because the latter would make Recovery more challenging by involving collections not being resharded.

- **Collect Timestamp:** In order to know where to restart replaying the oplog in the future, the latest timestamp from the oplog is stored in memory by mongos.

Execution: This phase is responsible for making placement decisions (which we will describe in Section 2.3) and executing the resultant data transfer among the secondaries. In the meantime, the primary servers concurrently continue to serve client CRUD operations. Since the selected secondaries are isolated, a consistent snapshot of the collection can be used to run the placement algorithms (Section 2.3) at a mongos server.

Assigning a new chunk to a mongod server implies migrating data in that chunk range from several other mongod servers to the assigned server. For each new chunk, the assigned server creates a separate TCP connection to each source server, "pulls" data from the appropriate old chunks, and commits it locally. All these migrations occur in parallel. We call this scheme of assigning a single socket to each migration as "chunk-based". The chunk-based strategy can create stragglers, and Section 2.4.1 addresses this problem.

Recovery: At the end of the Execution phase, the secondary servers have data stored according to the new configuration. However, any write (create, update or delete) operations that had been received by a primary server, since the time its secondary was isolated, now need to be communicated to the secondary.

Each primary forwards each item in the oplog to its appropriate new secondary, based on the new chunk ranges. This secondary can be located from our placement plan in the Execution phase, if the operation involved the new shard key. If the operation does not involve the new shard key, it is multicast to all secondaries, and each in turn checks whether it needs to apply it. This mirrors the way MongoDB typically routes queries among replica sets.

However, oplog replay is an iterative process – during the above oplog replay, further write operations may arrive at the primary oplogs. Thus, in the next iteration this delta oplog will need to be replayed. If the collection is hot, then these iterations may take very long to converge. To ensure convergence, we adopt two techniques: i) cap the replay at 2 iterations, and ii) enforce a write throttle before the last iteration. The write throttle is required because of the atomic commit phase that follows right afterwards. The write throttle rejects any writes received during the final iteration of oplog replay. An alternative was to buffer these writes temporarily at the primary and apply them later – however, this would have created another oplog and reinstated the original problem. In any case, the next phase (Commit) requires a write throttle anyway, and thus our approach dovetails with the Commit phase. Read operations remain unaffected and continue normally.

Commit: Finally, we bring the new configuration to the forefront and install it as the default configuration for the collection. This is done in one atomic step by continuing the write throttle from the Recovery phase.

This atomic step consists of two substeps:

- **Update Config:** The mongos server updates the config database with the new shard key and chunk ranges. Subsequent CRUD operations use the new configuration.
- **Elect Slave As Master:** Now the reconfigured secondary servers become the new primaries. The old primary steps down and Morphus ensures the secondary wins the subsequent leader election inside each replica set.

To end this phase, the new primaries (old secondaries, now reconfigured) unmute their oplog and the new secondaries (old primaries for each replica set, not yet reconfigured) unthrottle their writes.

Read-Write Behavior: The end of the Commit phase marks the switch to the new shard key. Until this point, all queries with old shard key were routed to the mapped server and all queries with new shard key were multicast to all the servers (normal MongoDB behavior). After the Commit phase, a query with the new shard key is routed to the appropriate server (new primary). Queries which do not use the new shard key are handled with a multicast, which is again normal MongoDB behavior.

Reads in MongoDB offer per-key sequential consistency. Morphus is designed so that it continues to offer the same consistency model for data under migration.

Slave Isolation & Catchup: After the Commit phase, the secondaries have data in the old configuration, while the primaries receive writes in the new configuration. As a result, normal oplog replay cannot be done from a primary to its secondaries. Thus, Morphus isolates all the remaining secondaries simultaneously.

The isolated secondaries catch up to the new configuration via *(replica set size - 1)* sequential *rounds*. Each round consists of the Execution, Recovery and Commit phases shown in Figure 3.1. However, some steps in these phases are skipped – these include the leader election protocol and config database update. Each replica set numbers its secondaries and in the ith round ($2 \leq i \leq$ *replica set size*), its ith secondary participates in the reconfiguration. The group of ith secondaries reuses the old placement decisions from the first round's Execution phase – we do so because secondaries need to mirror primaries.

After the last round, all background processes (such as the Balancer) that had been previously disabled are now re-enabled. The reconfiguration is now complete.

Fault-Tolerance: When there is no reconfiguration, Morphus is as fault-tolerant as MongoDB. Under ongoing reconfiguration, when there is one failure, Morphus provides similar fault-tolerance as MongoDB – in a nutshell, Morphus does not lose data, but in some cases

the reconfiguration may need to be restarted partially or completely.

Consider the single failure case in detail. Consider a replica set size of $rs \geq 3$. Right after isolating the first secondary in the first round, the old data configuration is still present at $(rs-1)$ servers: current primary and identical $(rs-2)$ idle secondaries. If the primary or an idle secondary fails, reconfiguration remains unaffected. If the currently-reconfiguring secondary fails, then the reconfiguration can be continued using one of the idle secondaries (from that replica set) instead; when the failed secondary recovers it participates in a subsequent reconfiguration round.

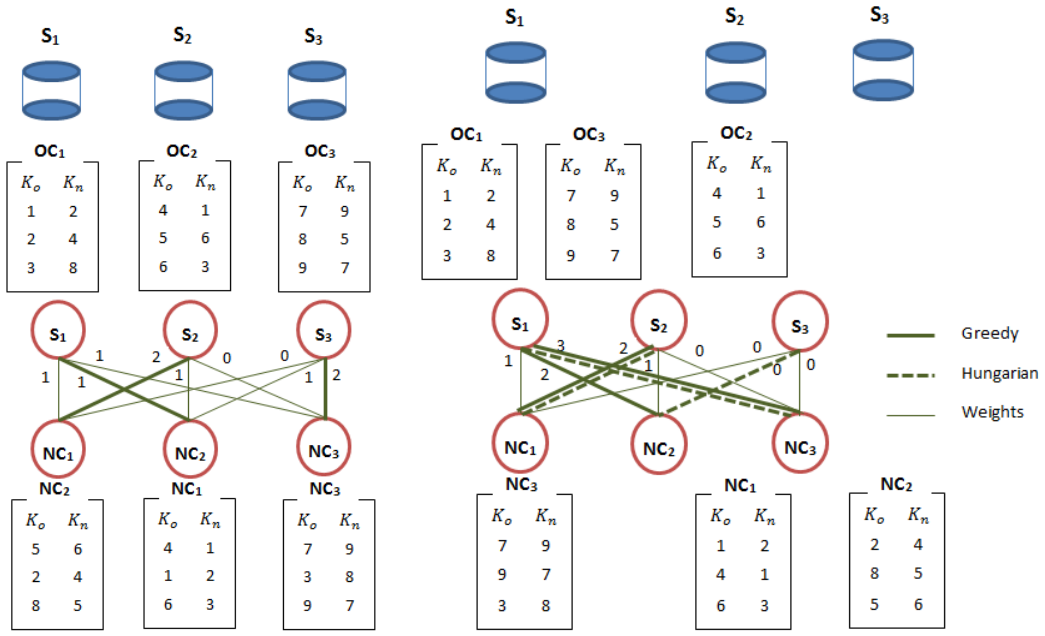
In a subsequent round (≥ 2), if one of the non-reconfigured replicas fails, it recovers and catches up directly with the reconfigured primary. Only in the second round, if the already-reconfigured primary fails, does the entire reconfiguration need to be restarted as this server was not replicated yet. Writes between the new primary election (Round 1 Commit phase) up to its failure, before the second round completes, may be lost. This is similar to the loss of a normal MongoDB write which happens when a primary fails before replicating the data to the secondary. The vulnerability window is longer in Morphus, although this can be reduced by using a backup Morphus server.

With multiple failures, Morphus is fault-tolerant under some combinations. For instance, if all replica sets have at least one new-configuration replica left, or if all replica sets have at least one old-configuration replica left. In the former case, failed replicas can catch up. In the latter case, reconfiguration can be restarted using the surviving replicas.

2.3 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.



(a) Balanced Old Configuration.

(b) Unbalanced Old Configuration.

Figure 2.2: Greedy and Hungarian strategy for shard key change using: (a) Balanced, (b) Unbalanced old chunk configuration. S_1 - S_3 represent servers. OC_1 - OC_3 and NC_1 - NC_3 are old and new chunks respectively. K_o and K_n are old and new shard keys respectively. Edges are annotated with W_{NC_i,S_j} weights.

2.3.1 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., $argmax_{S_*}(W_{NC_i,S_j})$. As chunks are considered independently, the algorithm produces the same output irrespective of the order in which chunks are considered by it.

The calculation of W_{NC_i,S_j} values can be performed in parallel at each server S_j, after servers are made aware of the new chunk ranges. A centralized server collects all the W_{NC_i,S_j} values, runs the greedy algorithm, and informs the servers of the allocation decisions.

Lemma 2.1 *The greedy algorithm is optimal in total network transfer volume.*

Proof: The proof is by contradiction. Consider an alternative optimal strategy A that assigns at least one new chunk NC_i to a server S_k different from $S' = argmax_{S_*}(W_{NC_i,S_j})$, such that $W_{NC_i,S'} > W_{NC_i,S_k}$ – if there is no such NC_i, then A produces the same total network transfer volume as the greedy approach. By instead changing A so that NC_i is

re-assigned to S', one can achieve a reconfiguration that has a lower network transfer volume than A, a contradiction.

For each of the m new chunks, this algorithm iterates through all the N servers. Thus its complexity is $O(m.N)$, linear in the number of new chunks and cluster size.

To illustrate the greedy scheme in action, Fig. 2.2 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_o 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. 2.2a, 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_i 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. 2.2b. 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.

2.3.2 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 $S_j^1 - S_j^{\lceil m/N \rceil}$. The bottom set of vertices consist of the new chunks. All edges between a top vertex S_j^k and a bottom vertex NC_i have an edge cost equal to $|NC_i| - W_{NC_i,S_j}$ i.e., the number of data items that will move to server S_j if new chunk NC_i 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 [53]. 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 of Section 2.3.1 becomes a special case of this algorithm with $V = m$.

Lemma 2.2 *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.*

Proof: The proof follows from the optimality of the Hungarian algorithm [53].

Fig. 2.2b 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 chapter 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. 2.2) 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 chapter.

2.4 NETWORK-AWARENESS

In this section, we describe how we augment the design of Section 2.2 in order to handle two important concerns: awareness to the topology of a datacenter, and geo-distributed settings.

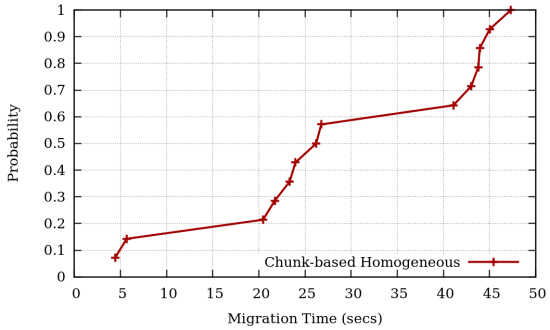


Figure 2.3: Execution phase CDF for chunk-based strategy on Amazon 500 MB database in tree network topology with 9 mongod servers spread evenly across 3 racks.

2.4.1 Awareness to Datacenter Topology

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 [54], CLOS [55], and butterfly [56].

Our first-cut data migration strategy discussed in Section 2.2 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 [57]. Further, the chunk-based approach also results in *stragglers* in the execution phase as shown in Figure 2.3. 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:

$$X_{i,j} \propto D_{i,j} \times L_{i,j} \tag{2.1}$$

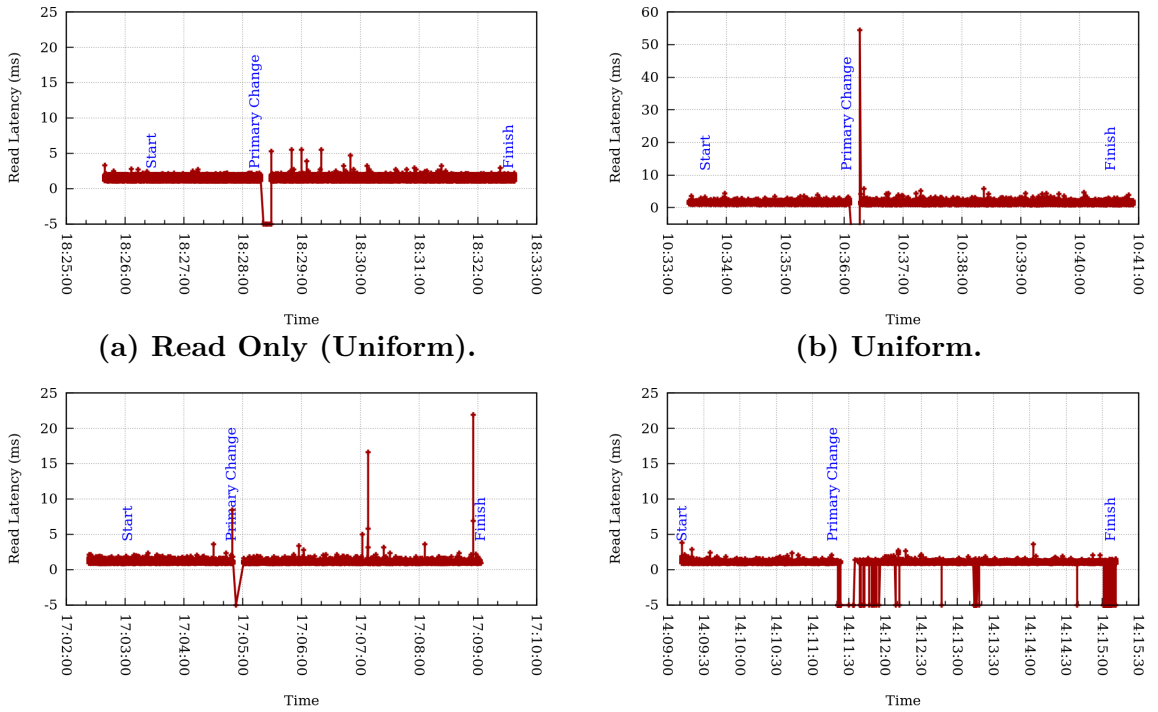
In our implementation, the weights determine the number of sockets that we assign to each flow. We assign each destination server j a total number of sockets $X_j = K \times \frac{\sum_i D_{i,j}}{\sum_{i,j} D_{i,j}}$, where K is the total number of sockets throughout the system. Thereafter each destination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

2.4.2 Geo-Distributed Settings

So far, Morphus has assumed that all its servers reside in one datacenter. However, typical NoSQL configurations split servers across geo-distributed datacenters for fault-tolerance. Naively using the Morphus system would result in bulk transfers across the wide-area network and prolong reconfiguration time.



(a) Read Only (Uniform). **(b) Uniform.**

(c) Zipf. **(d) Latest.**

Figure 2.4: Read Latency for: (a) Read only operations (no writes), and three read-write workloads modeled after YCSB: (b) Uniform, (c) Zipf, and (d) Latest. Times shown are hh:mm:ss. Failed reads are shown as negative latencies. Annotated "Primary Change" point marks the start of the leader election protocol in the first round.

To address this, we localize each stage of the data transfer to occur within a datacenter. We leverage MongoDB's server tags [58] to tag each replica set member with its datacenter identifier. Morphus then uses this information to select replicas, which are to be reconfigured together in each given round, in such a way that they reside within the same datacenter. If wide-area transfers cannot be eliminated at all, Morphus warns the database admin.

One of MongoDB's invariants for partition-tolerance requires each replica set to have a voting majority at some datacenter [58]. In a three-member replica set, two members (primary and secondary-1) must be at one site while the third member (secondary-2) could be at a different site. Morphus obeys this requirement by selecting that secondary for the first round which is co-located with the current primary. In the above example, Morphus would select the secondary-1 replicas for the first round of reconfiguration. In this way, the invariant stays true even after the leader election in the Commit phase.

2.5 EVALUATION

Our experiments are designed to answer the following questions:

- How much does Morphus affect read and write operations during reconfiguration?
- For shard key change, how do the Greedy and Hungarian algorithms of Section 2.3 compare?
- How does Morphus scale with data size, operation injection rate, and cluster size?
- How much benefit can we expect to get from the network-aware (datacenter topology and geo-distributed) strategies?

2.5.1 Setup

Data Set: We use the dataset of Amazon reviews as our default collection [59]. 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).

Cluster: 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 [60] 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.

Workload Generator: 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 [61] 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 default settings: Morphus was implemented in about 4000 lines of C++ code. The code is publicly available at `http://dprg.cs.uiuc.edu/downloads`. A demo of Morphus can be found at `https://youtu.be/0rO2oQyyg0o`. For the evaluation, each plotted datapoint is an average of at least 3 experimental trials, shown along with standard deviation bars.

2.5.2 Read Latency

Fig. 2.4 shows the timelines for four different workloads during the reconfiguration, lasting between 6.5 minutes to 8 minutes. The figure depicts the read latencies for the reconfigured

database table (collection), with failed reads shown as negative latencies. We found that read latencies for collections not being reconfigured were not affected and we do not plot these.

Fig. 2.4a shows the read latencies when there are no writes (Uniform read workload). We observe unavailability for a few seconds (from time t =18:28:21 to t =18:28:29) during the Commit phase when the primaries are being changed. This unavailability lasts only about 2% of the total reconfiguration time. After the change, read latencies spike slightly for a few reads but then settle down. Figs. 2.4b to 2.4d plot the YCSB-like read-write workloads. We observe similar behavior as Fig. 2.4a.

	Read	Write
Read Only	99.9	-
Uniform	99.9	98.5
Latest	97.2	96.8
Zipf	99.9	98.3

Table 2.1: Percentage of Reads and Writes that Succeed under Reconfiguration. (Figs. 2.4 and 2.6)

Fig. 2.4d indicates that the Latest workload incurs a lot more failed reads. This is because the keys that are being inserted and updated are the ones more likely to be read. However, since some of the insertions fail due to the write throttles at various points during the reconfiguration process (t =14:11:20 to t =14:11:30, t =14:13:15 to t =14:13:20, t =14:15:00 to t =14:15:05 from Fig. 2.6c), this causes subsequent reads to also fail. However, these lost reads only account for 2.8% of the total reads during the reconfiguration time – in particular, Table 2.1 (middle column) shows that 97.2% of the reads succeed under the Latest workload. The availability numbers are higher at three-9's for Uniform and Zipf workload, and these are comparable to the case when there are 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, Morphus mildly reduces the offered availability.

To flesh this out further, we plot in Fig. 2.5 the CDF of read latencies for these 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

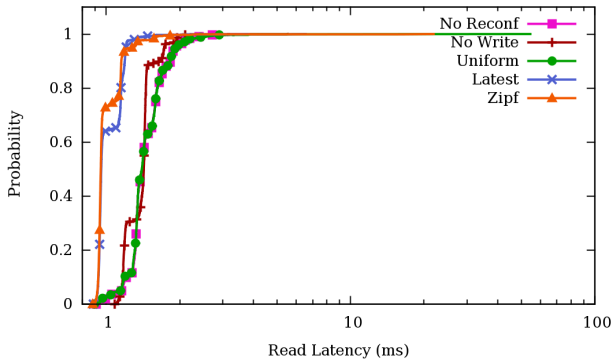


Figure 2.5: CDF of Read Latency. Read latencies under no reconfiguration (No Reconf), and four under-reconfiguration workloads.

successful reads.

We conclude that under reconfiguration, the read availability provided by Morphus is high (two to three 9's of availability), while the latencies of successful read operations do not degrade compared to the scenario when there is no reconfiguration in progress.

2.5.3 Write Latency

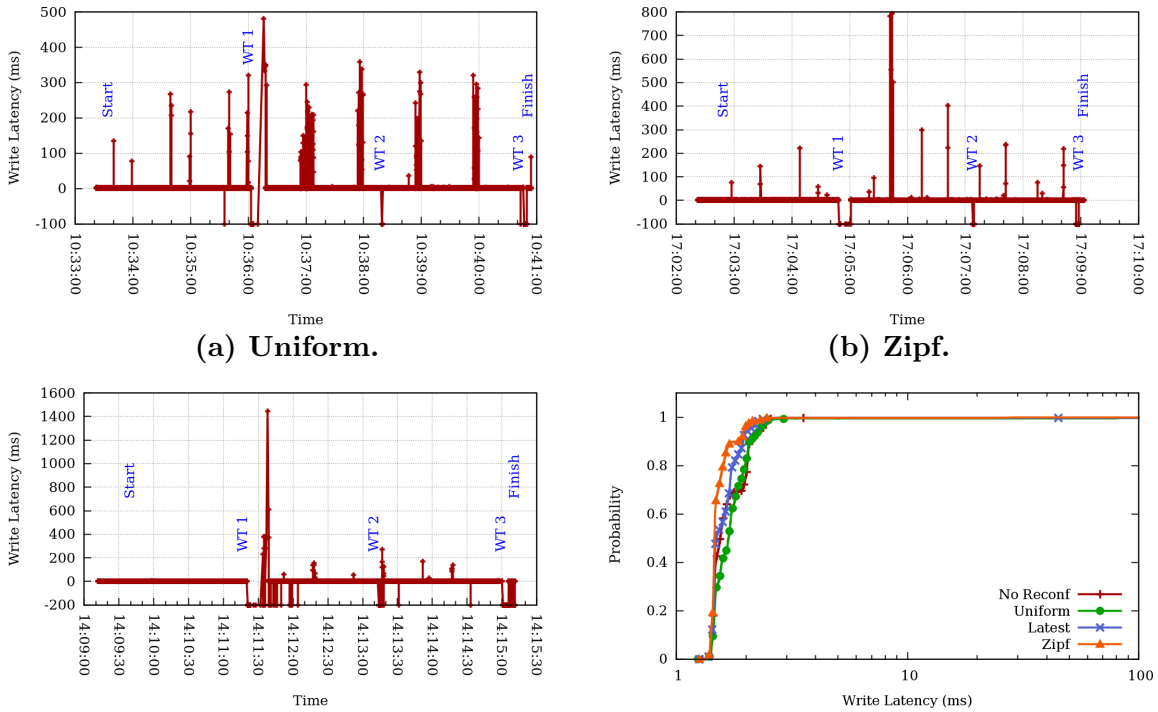
We next plot the data for write operations, i.e., inserts, updates and deletes.

Figs. 2.6a to 2.6c show writes in the same timelines as Figs. 2.4b to 2.4d. We observe that many of the failed writes occur during one of the write throttling periods (annotated as "WT"). Recall from Section 2.2.2 that there are as many write throttling periods as the replica set size, with one throttle period at the end of each reconfiguration round.

Yet, the number of writes that fail is low: Table 2.1 (last column) shows that for the Uniform and Zipf workloads, fewer than 2% writes fail. The Latest workload again has a slightly higher failure rate since a key that was attempted to be written (unsuccessfully) is more likely to be attempted to be written again in the near future. Yet, the write failure rate of 3.2% is reasonably low.

To flesh this out further, the CDF of the write latencies (ignoring failed writes) is shown in Fig. 2.6d. The median for writes when there is no reconfiguration (Uniform workload) in progress is 1.45 ms. The Zipf and Latest workloads have a similar median latency. Uniform has a slightly higher median latency at 1.6 ms – this is because 18% of the updates experience high latencies. This is due to Greedy's skewed chunk assignment plan (discussed and improved in Section 2.5.4). Greedy assigns a large percentage of new chunks to a single replica set, which thus receives most of the new write traffic. This causes MongoDB's periodic write journal flushes [4] to take longer. This in turn delays the new writes arriving around

[4]MongoDB maintains a write-ahead (journal) log for durability which is periodically flushed to disk.



(a) Uniform. **(b) Zipf.**

(c) Latest. **(d) CDF of Write Latency Distribution.**

Figure 2.6: Write Latency for three read-write workloads modeled after YCSB: (a) Uniform, (b) Zipf, and (c) Latest. Times shown are hh:mm:ss. Failed writes are shown as negative latencies. Annotations marked "WT" indicate the start of each write throttle phase. (d) CDF of Write Latency Distribution for no reconfiguration (No Reconf) and three under-reconfiguration workloads.

the journal flush timepoints. Many of the latency spikes observed in Figs. 2.6a to 2.6c arise from this journaling behavior.

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

2.5.4 Hungarian vs. Greedy Reconfiguration

Section 2.3 outlined two algorithms for the shard key change reconfiguration – Hungarian and Greedy. We implemented both these techniques into Morphus – we call these variants as Morphus-H and Morphus-G respectively. For comparison, we also implemented a random chunk assignment scheme called Morphus-R.

Fig. 2.7 compares these three variants under two scenarios. The uncorrelated scenario (left pair of bars) uses a synthetic 1 GB dataset where for each data item, the value for its new shard key is selected at random and uncorrelated to its old shard key's value. The plot shows

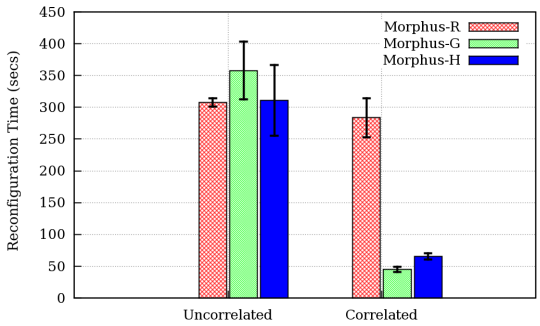
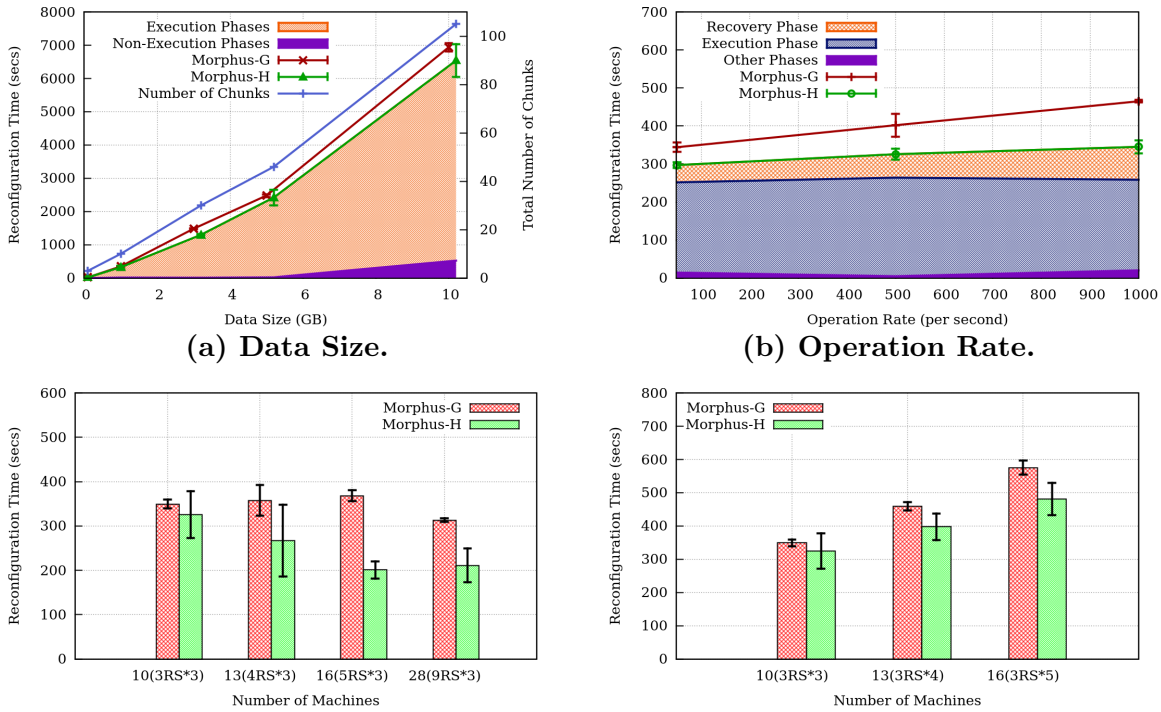


Figure 2.7: Greedy (Morphus-G) vs. Hungarian (Morphus-H) Strategies for shard key change. Uncorrelated: random old and new shard key. Correlated: new shard key is reverse of old shard key.

the total reconfiguration time including all the Morphus phases. Morphus-G is 15% worse than Morphus-H and Morphus-R. This is because Morphus-G ends up assigning 90% new chunks to a single replica set which results in stragglers during the Execution and Recovery phases. The underlying reason for the skewed assignment can be attributed to MongoDBs split algorithm which we use modularly. The algorithm partitions the total data size instead of total record count. When partitioning the data using the old shard key, this results in some replica sets getting a larger number of records than others. Morphus-R performs as well as Morphus-H because by randomly assigning chunks to servers, it also achieves load balance

The correlated scenario in Fig. 2.7 (right pair of bars) shows the case where new shard keys have a reversed order compared to old shard keys. That is, with M data items, old shard keys are integers in the range $[1, M]$, and the new shard key for each data item is set as $= M-$old shard key. This results in data items that appeared together in chunks continuing to do so (because chunks are sorted by key). Morphus-R is 5x slower than both Morphus-G and Morphus-H. Randomly assigning chunks can lead to unnecessary data movement. In the correlated case, this effect is accentuated. Morphus-G is 31% faster than Morphus-H. This is because the total transfer volume is low anyway in Morphus-G due to the correlation, while Morphus-H additionally attempts to load-balance.

We conclude that i) the algorithms of Section 3 give an advantage over random assignment, especially when old and new keys are correlated, and ii) Morphus-H performs reasonably well in both the correlated and uncorrelated scenario and should be preferred over Morphus-G and Morphus-R.



(a) Data Size. **(b) Operation Rate.**

(c) Number of Replica Sets. **(d) Size of Replica Set.**

Figure 2.8: Morphus Scalability with: (a) Data Size, also showing Morphus-H phases, (b) Operation injection rate, also showing Morphus-H phases, (c) Number of replica sets, and (d) Replica set size.

2.5.5 Scalability

We explore scalability of Morphus along three axes – database size, operation injection rate, and size of cluster. These experiments use the Amazon dataset.

Database Size: Fig. 2.8a shows the reconfiguration time at various data sizes from 1 GB to 10 GB. There were no reads or writes injected. For clarity, the plotted data points are perturbed slightly horizontally.

Firstly, Fig. 2.8a shows that Morphus-H performs slightly better than Morphus-G for the real-life Amazon dataset. This is consistent with our observations in Section 2.5.4 since the Amazon workload is closer to the uncorrelated end of the spectrum.

Secondly, the total reconfiguration time appears to increase superlinearly beyond 5 GB. This can be attributed to two factors. First, reconfiguration time grows with the *number of chunks* – this number is also plotted, and we observe that it grows superlinearly with datasize. This is again caused by MongoDB's splitting code [5]. Second, we have reused MongoDB's data transfer code, which relies on cursors (i.e., iterators), which are not the best approach for bulk transfers. We believe this can be optimized further by writing a

[5]Our results indicate that MongoDB's splitting algorithm may be worth revisiting.

module for bulk data transfer – yet, we reiterate that this is orthogonal to our contributions: even during the (long) data transfer time, reads and writes are still supported with several 9s of availability (Table 1). Today's existing approach of exporting/reimporting data with the database shut down, leads to long unavailability periods – at least 30 minutes for 10 GB of data (assuming 100% bandwidth utilization). In comparison, Morphus is unavailable in the worst-case (from Table 1) for 3.2% $\times$ 2 hours = 3.84 minutes, which is an improvement of about 10x.

Fig. 2.8a also illustrates that a significant fraction of the reconfiguration time is spent migrating data, and this fraction grows with increasing data size – at 10 GB, the data transfer occupies 90% of the total reconfiguration time. This indicates that Morphus' overheads fall and will become relatively small at large data sizes.

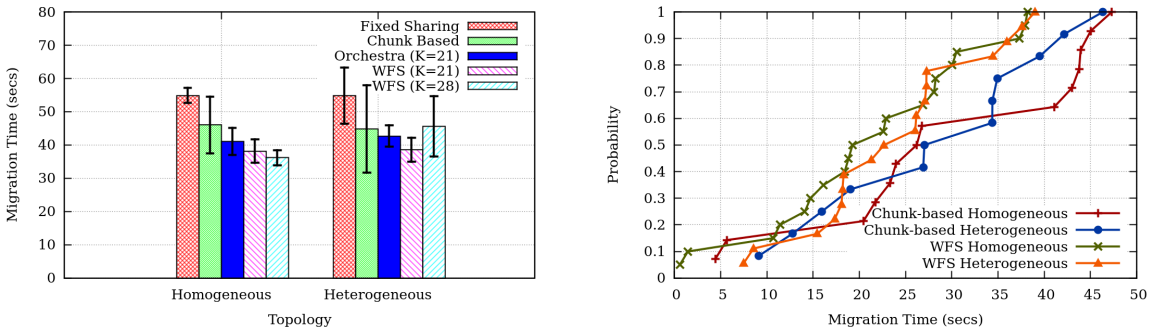
We conclude that the reconfiguration time incurred by Morphus scales linearly with the number of chunks in the system and that the overhead of Morphus falls with increasing data size.

Operation Injection Rate: An important concern with Morphus is how fast it plays "catch up" when there are concurrent writes during the reconfiguration. Fig. 2.8b plots the reconfiguration time against the write rate on 1 GB of Amazon data. In both Morphus-G and Morphus-H, we observe a linear increase. More concurrent reads and writes slow down the overall reconfiguration process because of two reasons: limited bandwidth available for the reconfiguration data transfers, and a longer oplog that needs to be replayed during the Recovery Phase. However, this increase is slow and small. A 20-fold increase in operation rate from 50 ops/s to 1000 ops/s results in only a 35% increase in reconfiguration time for Morphus-G and a 16% increase for Morphus-H.

To illustrate this further, the plot shows the phase breakdown for Morphus-H. The Recovery phase grows as more operations need to be replayed. Morphus-H has only a sublinear growth in reconfiguration time. This is because of two factors. First Morphus-H balances the chunks out more than Morphus-G, and as a result the oplog replay has a shorter tail. Second, there is an overhead in Morphus-H associated with fetching the oplog via the MongoDB cursors (iterators) – at small write rates, this overhead dominates but as the write rate increases, the contribution of this overhead drops off. This second factor is present in Morphus-G as well, however it is offset by the unbalanced distribution of new chunks.

We conclude that Morphus catches up quickly when there are concurrent writes, and that its reconfiguration time scales linearly with write operation injection rate.

Cluster Size: We investigate cluster size scalability along two angles: number of replica sets, and replica set size. Fig. 2.8c shows that as the number of replica sets increases from 3 to 9 (10 to 28 servers), both Morphus-G and Morphus-H eventually become faster with



(a) **(b)**

Figure 2.9: (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 $K = 28$.

scale. This is primarily because of increasing parallelism in the data transfer, while the amount of data migrating over the network grows much slower – with N replica sets, this latter quantity is approximately a fraction $\frac{N-1}{N}$ of data. While Morphus-G's completion time is high at a medium cluster size (16 servers) due to its unbalanced assignment, Morphus-H shows a steady improvement with scale and eventually starts to plateau as expected.

Next, Fig. 2.8d shows the effect of increasing replica set size. We observe a linear increase for both Morphus-G and Morphus-H. This is primarily because there are as many rounds inside a reconfiguration run as there are machines in a replica set.

We conclude that Morphus scales reasonably with cluster size – in particular, an increase in number of replica sets improves its performance.

2.5.6 Effect of Network Awareness

Datacenter Topology-Awareness: First, Fig. 2.9a 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 2.4.1), c) Orchestra [57] with $K = 21$, d) WFS with $K = 21$ (Section 2.4.1), 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 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. 2.9b shows that compared to Fig. 2.3 (from Section 2.4), 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 WFS strategy improves performance compared to existing approaches, and K should be chosen high enough but without causing congestion.

Geo-Distributed Setting: Table 2.2 shows the benefit of the tag-aware approach of Morphus (Section 2.4.2). The setup has two datacenters with 6 and 3 servers, with intra- and inter-datacenter latencies of 0.07 ms and 2.5 ms respectively (based on [62]) and links with 100 Mbps bandwidth. For 100 ops/s workload on 100 MB of reconfigured data, tag-aware Morphus improves performance by over 2x when there are no operations and almost 3x when there are reads and writes concurrent with the reconfiguration.

	Without Read/Write	With Read/Write
Tag-Unaware	49.074s	64.789s
Tag-Aware	21.772s	23.923s

Table 2.2: Reconfiguration Time in the Geo-distributed setting.

2.5.7 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 [63]. 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 similar to the one used in Section 2.5.4. Morphus-H was used for reconfiguration with WFS migration scheme and K = number of old chunks.

Fig. 2.10 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

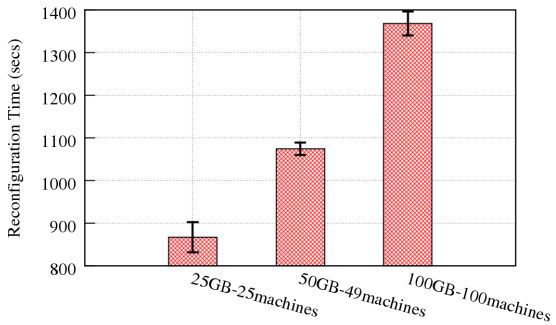


Figure 2.10: Running Morphus-H with WFS (K = number of old chunks) for reconfiguring database of size (25GB, 50GB, 100GB) running on a cluster size (25 machines (8 replica sets * 3 + 1 mongos), 49 machines (16 replica sets) and 100 machines (33 replica sets)).

amount of data per replica set is constant.

2.6 RELATED WORK

Research in distributed databases has focused on query optimization and load-balancing [64], and orthogonally on using group communication for online reconfiguration [65], however, they do not solve the core algorithmic problems for efficient reconfiguration. Online schema change was targeted in [66], but the resultant availabilities were lower than those provided by Morphus. 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 [68, 69], but they do not target optimality for reconfiguration.

The problem of live migration has been looked into in the context of databases. Albatross [45], Zephyr [46] and ShuttleDB [44] 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 achieves this inside the same replica sets. Also, they do not propose optimal solutions for any reconfiguration operation. Opportunistic lazy migration explored in Relational Cloud [70] entails longer completion times. Tuba [71] looked into the problem of migration in a geo-replicated setting. They avoided write thottle by having multiple masters at the same time. MongoDB does not support multiple masters in a single replica set, which dictated Morphus's current design.

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 [72]. Like pre-copy, Morphus also replays operations that occurred during the migration. Pre-copy

systems also use write throttling [73]. Pre-copy has been used in database migration [74].

For network flow scheduling, Chowdhury et.al [57] 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 [75] 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.

2.7 SUMMARY

This chapter described optimal and load-balanced algorithms for online reconfiguration operation, and the Morphus system integrated into MongoDB. Our experiments showed that Morphus supports fast reconfigurations such as shard key change, while only mildly affecting the availability and latencies for read and write operations. Morphus scales well with data size, operation injection rate, and cluster size.

Chapter 3: Parqua: Online Reconfigurations in Virtual Ring-Based NoSQL Systems

In this chapter, we discuss the problem of changing configuration parameters like shard key, block size etc. at the level of a database table in a virtual-ring based NoSQL distributed database like Cassandra. Virtual-ring based databases pose its unique set of challenges because of which techniques in Morphus are not applicable here. We outline them in in Section 3.1. We define the system model in Section 3.2. We describe the key technique in Parqua in Section 3.3. Finally, we summarize the results from our evaluation in Section 3.4.

3.1 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 [76] from Amazon. Dynamo, and subsequent open-source variants of it including Facebook's Apache Cassandra [42], Basho's Riak [29], and LinkedIn's Voldemort [77] 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 [11], BigTable [6], 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 [37].

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 [78].

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 require-

ment 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 [47], JIRA (bug tracking) issues that are hotly debated [38], and many blogs [48, 49]. The manual approach to resharding took over two years at Google [50].

In our past work [79], 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. 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.

[1] The Korean word for "change."

3.2 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.

3.2.1 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 [80] 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 [6]. 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.

3.2.2 Cassandra Background

We incorporated our design in Apache Cassandra, which is a popular ring-based distributed key-value store [42]. Cassandra borrows the architecture designs heavily from Distributed Hash Tables (DHTs) such as Chord [80].

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.

3.3 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.

3.3.1 Parqua: Overview

Parqua runs reconfiguration in four phases. The graphical overview of Parqua phases is shown in Fig. 3.1. 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.

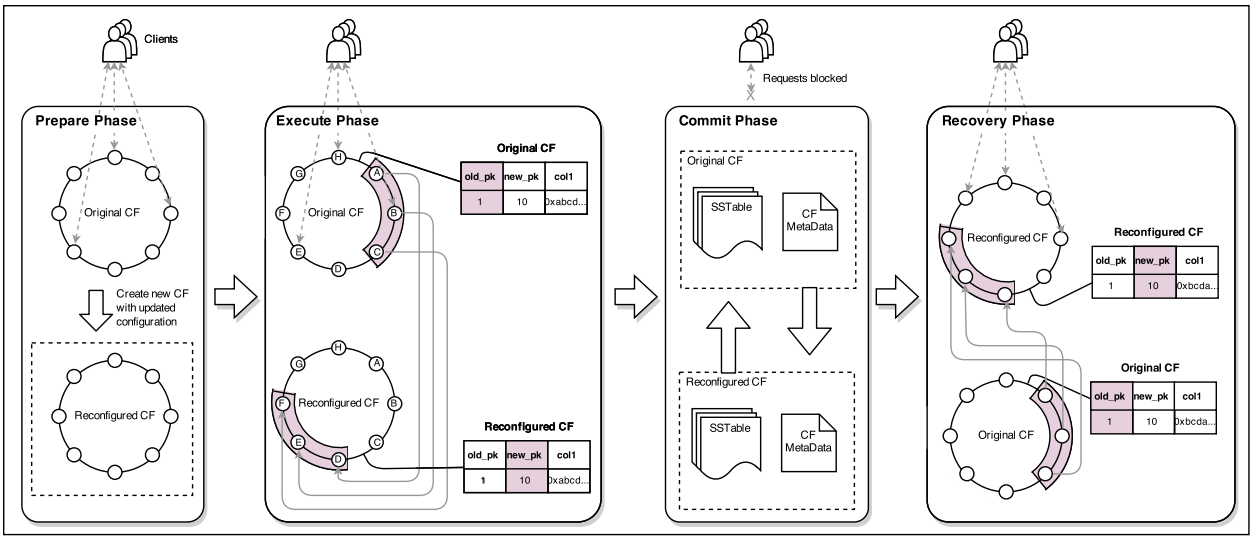


Figure 3.1: Overview of Parqua phases. The gray solid lines represent internal entry transfers, and the gray dashed lines mean client requests. The phases progress from left to right.

Next, we discuss these individual phases in detail.

3.3.2 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. 3.1, 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

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

that were written during the reconfiguration. The SSTable is an immutable 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 [81], 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.

3.3.3 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.

3.4 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?

3.4.1 Setup

Data Set: We used the Yahoo! Cloud Service Benchmark(YCSB) [61] 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 [60]. 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* [42]. 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.

3.4.2 Reconfiguration Time

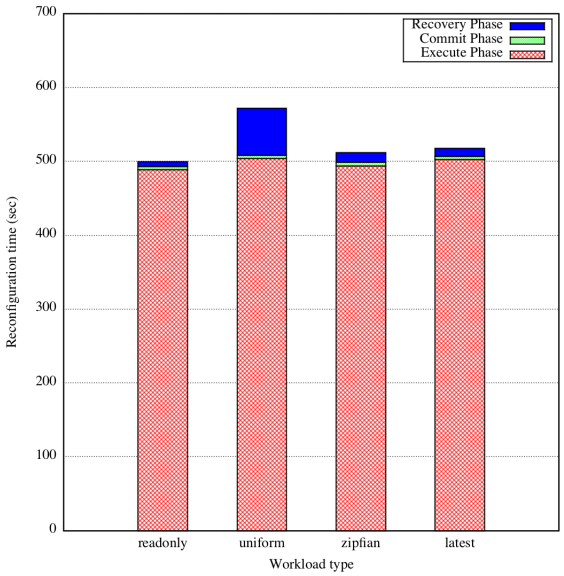


Figure 3.2: Time taken for reconfiguration for different workload types and breakdown by individual phases.

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. 3.2 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.

The overall reconfiguration time is small considering that the size of unique entries is 10 GB. Compared to our past work on Morphus [79], 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 3.1: Percentage of reads and writes that succeed during reconfiguration.

3.4.3 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 3.1 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 3.4.4.

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 3.4.4.

3.4.4 Read Latency

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

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

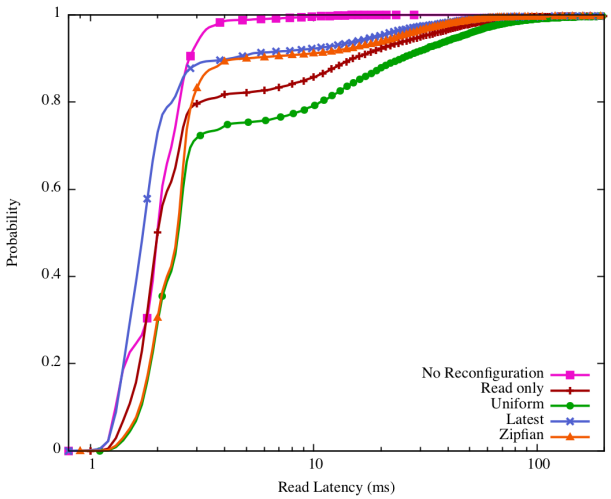


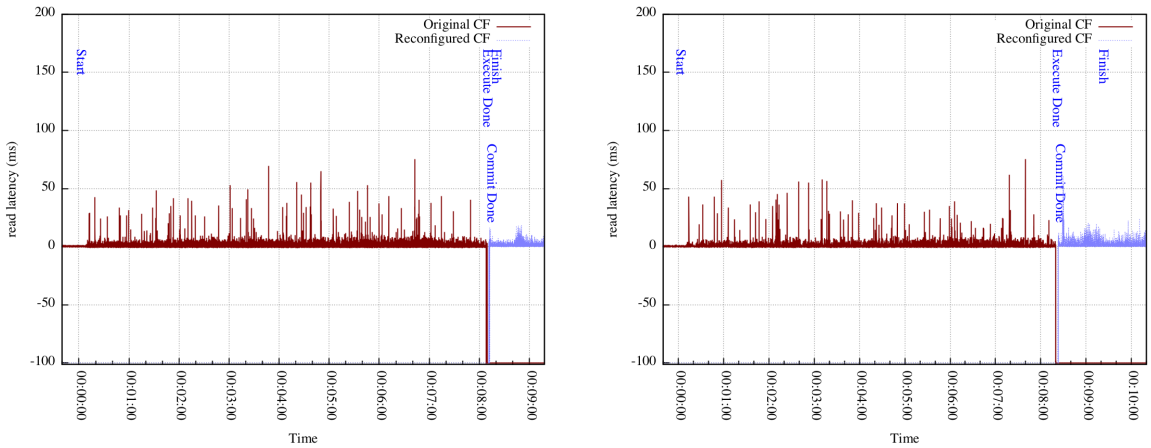
Figure 3.3: CDF of read latencies for different workloads under reconfiguration, and the CDF of read latency under no reconfiguration for baseline measurement. The x-axis is read latency in logarithmic scale, while the y-axis is the cumulative probability.

First, Fig. 3.4 shows the read latencies over time for four different workloads during the reconfiguration of Cassandra using Parqua. Fig. 3.4a 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 3.4.2. Fig. 3.4 (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 [82].

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.

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



(a) Read Only. **(b) Uniform.**

(c) Zipfian. **(d) Latest.**

Figure 3.4: 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.

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. 3.4d 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 3.4.3, this is inherent to the multi-threaded YCSB workload generator.

Read Latency CDF: Fig. 3.3 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 3.1.

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 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.

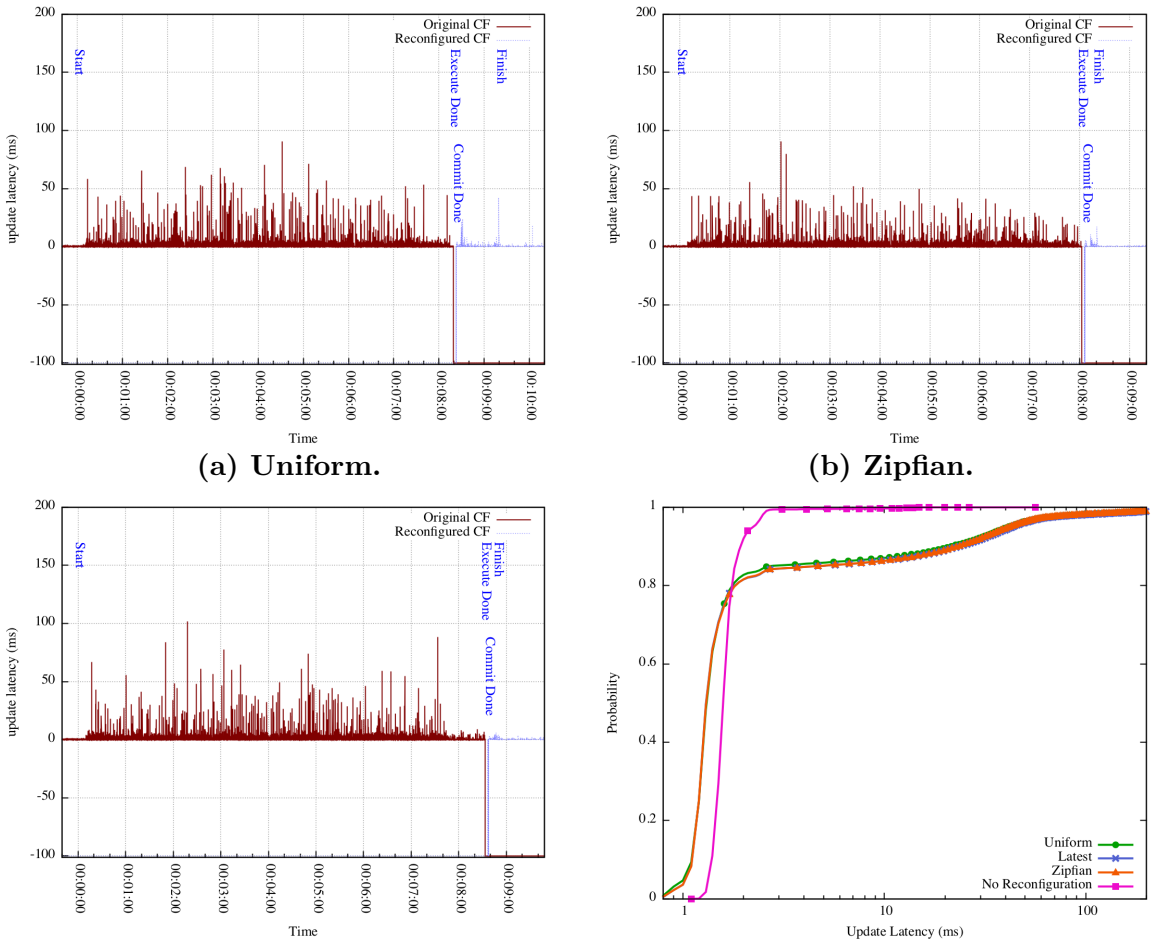
Next, when we compare latest and zipfian workloads, we observe the latest workload is slightly faster up to the 90th percentile level, and they share similar read latency above that percentile. This is because for latest workload, the most frequently accessed keys are among those that were inserted most 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.

3.4.5 Write Latency

In our next set of experiments, we investigate the write latency characteristics of Parqua system. Similar to Section 3.4.4, we aim to observe the effect of the reconfiguration on normal write operations.

Write Latency Over Time: Fig. 3.5 (a), (b), and (c) depict the write latencies over time for different workloads under reconfiguration. This is for the same experiment as Fig. 3.4. We see that many operations fail in the Commit phase, similar to Section 3.4.4. As explained in Section 3.3.2, 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 3.4.4, we also observe the latency spikes in the Execute phase and the Recovery phase. However, unlike in Section 3.4.4, we do not see differing behaviors of



(a) Uniform. **(b) Zipfian.**

(c) Latest. **(d) CDF of update latency Distribution.**

Figure 3.5: 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.

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.

Write Latency CDF: In Fig. 3.5d, 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.

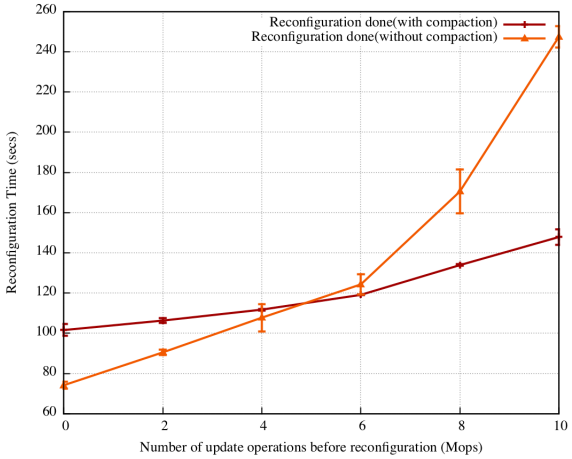


Figure 3.6: Reconfiguration time for number of injected update operations under two different implementations of Parqua system.

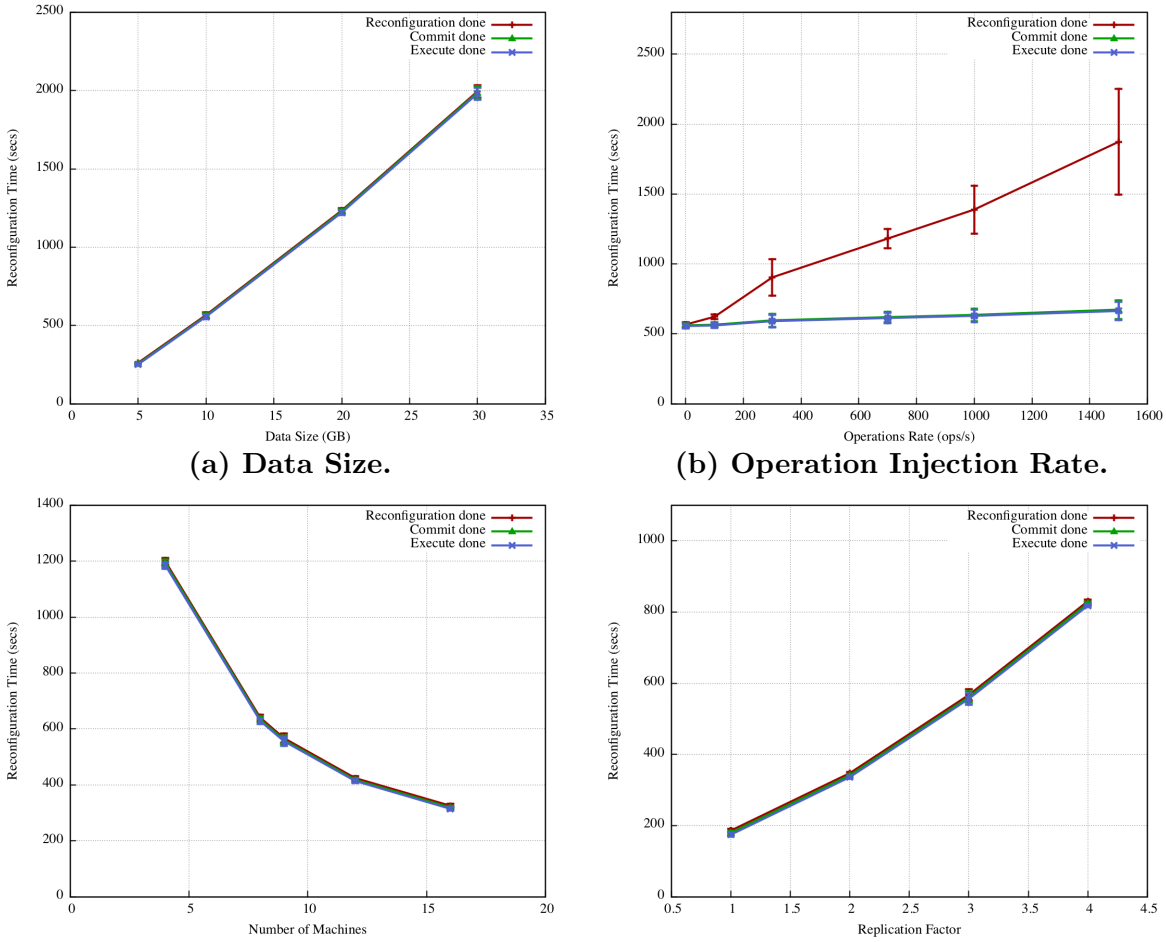
3.4.6 Scalability

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.

Database Size: Fig. 3.7a 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.

Operation Injection Rate: Fig. 3.7b 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.



(a) Data Size. **(b) Operation Injection Rate.**

(c) Number of Cassandra Machines. **(d) Replication Factor.**

Figure 3.7: 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.

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.

Replication Factor: Next, Fig. 3.7d 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.

Cluster Size: Finally, we demonstrate the reconfiguration time as we scale the cluster size. Database size was fixed at 10 GB. In Fig. 3.7c, 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.

3.4.7 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. 3.6 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 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. 3.6, 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.

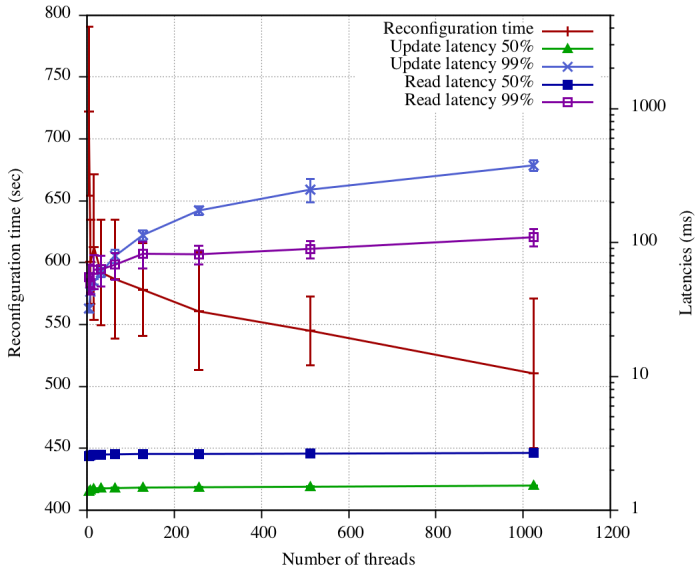


Figure 3.8: Reconfiguration time and read/write latency over number of migration threads for reconfiguration.

3.4.8 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) [83]. 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 in tail latency.

Fig. 3.8 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. 3.8, 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.

3.5 SUMMARY

In this chapter, 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.

Chapter 4: Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines

In this chapter, we discuss the problem of dynamically changing replication of data based on popularity of access in distributed interactive analytics engine like Druid. We motivate the need for smarter replication strategy in distributed interactive analytics engine in Section 4.1. In Section 4.2.2, we analyze workload traces from Yahoo! to characterize a typical production workload. We present our algorithmic contributions in Section 4.3 and system design in Section 4.4. We evaluate Getafix using workloads derived from Yahoo! production clusters and present the results in Section 4.5.

4.1 INTRODUCTION

Real-time analytics is projected to grow annually at a rate of 31% [84]. Apart from stream processing engines, which have received much attention [85, 86, 14], real time analytics now also includes the burgeoning area of interactive data analytics engines such as Druid [17], Redshift [87], Mesa [88], Presto [18] and Pinot [30]. These systems have seen widespread adoption [89, 90] in companies which require applications to support sub-second query response time. Applications span usage analytics, revenue reporting, spam analytics, ad feedback, and others [91]. Typically large companies have their own on-premise deployments while smaller companies use a public cloud. The internal deployment of Druid at Yahoo! (now called Oath) has more than 2000 hosts, stores petabytes of data and serves millions of queries per day at sub-second latency scales [91].

In interactive data analytics engines, data is continuously ingested from multiple pipelines including batch and streaming sources, and then indexed and stored in a data warehouse. This data is immutable. The data warehouse resides in a backend tier, e.g., HDFS [8] or Amazon S3 [25]. As data is being ingested, users (or programs) submit queries and navigate the dataset in an interactive way.

The primary requirement of an interactive data analytics engine is fast response to queries. Queries are run on multiple compute nodes that reside in a frontend tier (cluster). Compute nodes are expected to serve 100% of queries directly from memory [1]. Due to limited memory, the compute nodes cannot store the entire warehouse data, and thus need to smartly fetch and cache data locally. Therefore, interactive data analytics engines need to navigate the tradeoff between memory usage and query latency.

[1] While SSDs could be used, they increase latency, thus production deployments today are almost always in-memory.

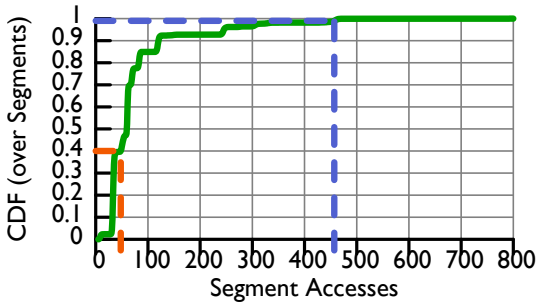


Figure 4.1: CDF of segment popularity collected from Yahoo! production trace.

Interactive analytics engines employ two forms of parallelism. First, data is organized into data blocks, called *segments*–this is standard in all engines. For instance, in Druid, hourly data from a given source constitutes a segment. Second, a query that accesses multiple segments can be run in parallel on each of those segments, and then the results are collected and aggregated. Query parallelization helps achieve low latency. Because a query (or part thereof) running at a compute node needs to have its input segment(s) cached at that node's memory, *segment placement* is a problem that needs careful solutions. Full replication is impossible due to the limited memory.

This chapter proposes new intelligent schemes for placement of data segments in interactive analytics engines. The key idea is to exploit the strong evidence [31] that at any given point of time, some data segments are more popular than others. When we analyzed traces from Yahoo!'s Druid cluster, we found that the top 1% of data is an order of magnitude more popular than the bottom 40%–in Figure 4.1, the bottom 40% popular segments account for 6% of the total accesses while the top 1% account for 43%. Today's deployments either uniformly replicate all data, or require system administrators to manually create storage tiers with different replication factors in each tier. Only the latter approach can account for popularity, but it is manual, laborious, and cannot adapt in real time to changes in query patterns.

Figure 4.2 shows the query latency for two cluster sizes (15, 30 compute nodes) and query rates (1500, 2500 qps). For each configuration (cluster size / query rate pair), as the replication factor (applied uniformly across segments) is increased, we observe the curve hits a "knee", beyond which further replication yields marginal latency improvements. The knee for 15 / 2500 is 9 replicas, and for the other two is 6 replicas. Our goal is to achieve the knee of the curve for individual segments (which is a function of their respective query loads), in an adaptive way.

Popularity is often confused with recency. Systems like Druid [17] approximate popularity by over-replicating data that was ingested recently (few hours to days). While there is

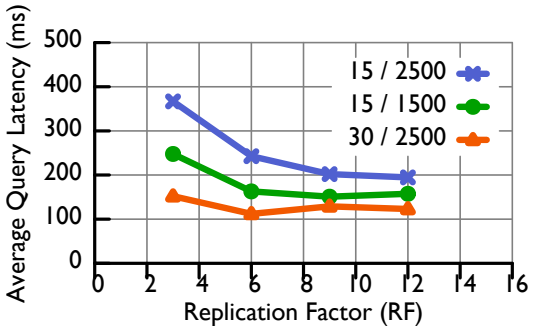


Figure 4.2: Average Query Latency observed with varying replication factors for different (cluster size / query injection rate) combinations.

correlation with recency, popularity needs to be treated as a first class citizen. Figure 4.3 shows our analysis of Yahoo!'s production Druid cluster. We find that some older data can be popular (transiently or persistently). For instance, in Figure 4.3 recent segments (B1) have a 50% chance of co-occurring with segments that are up to 5 months old (A1)–we explain this plot in detail later (§4.2.2). Purely using recency may result in popular old data becoming colocated with recent data at a compute node, overloading that node with many queries and prolonging query completion times. Another approach to approximating popularity is to use concurrent accesses, as in Scarlett [31]. We experimentally compare our work against Scarlett.

We present a new system called Getafix [2] that adaptively decides replication level and replica placement for segments. Getafix's goal is to significantly reduce usage of the most critical resource, namely memory [3], without affecting query latency. Getafix is built atop intuition arising from our optimal solution to the static version of the replication problem. Our static solution is provably optimal in *both* makespan (runtime of the query set) as well as memory costs. In the dynamic scenario, Getafix makes replication decisions by continually measuring query injection rate, segment popularity, and current cluster state.

We implemented Getafix and integrated it into Druid [17], one of the most popular open-source interactive data analytics engines in use today. We present experimental results using workloads from Yahoo! Inc.'s production Druid cluster. We compare Getafix to two known baselines: 1) base Druid system with uniform replication, and 2) ideas adapted from Scarlett [31], which solves replication in batch systems like Hadoop [3], Dryad [92], etc. Compared to these, Getafix achieves comparable query latency (both average and tail), while saving memory by 1.45-2.15× in a private cloud and cutting memory dollar costs in a public cloud by as much as $1.15K per hour (for a 100 TB dataset, thus an annual cost

[2]In "Asterix" comics, Getafix is the name of the village druid who brews magic potions.

[3]In this work, we deal with memory used to store segments. We ignore working memory required for executing queries as they are typically small for the queries Druid supports.

savings of $10 M).

The main contributions of this chapter are:

- We present workload characteristics of segment popularity in interactive data analytics engines (§4.2.2).
- We formulate and optimally solve the static version of the segment replication problem, for a given set of queries accessing a given set of segments (§4.3).
- We design the Getafix system to handle dynamic query and segment workloads (§4.4). We implement Getafix into Druid, a modern interactive data analytics engine.
- We evaluate Getafix using workload derived from Yahoo! production clusters (§4.5).

4.2 BACKGROUND

4.2.1 System Model

We present a general architecture of an interactive data analytics engine. To be concrete, we borrow some terminology from a popular system in this space, Druid [17].

An interactive data analytics engine receives data from both batch and streaming pipelines. The incoming data from batch pipelines is directly stored into a backend storage tier, also called *deep storage*. Data from streaming pipelines is collected by a *realtime node* for a pre-defined time interval and/or till it reaches a size threshold. The collected events are then indexed and pushed into deep storage. This chunk of events is identified by the time interval it was collected in (e.g., hourly, or minute-ly), and is called a *segment*. A segment is an immutable unit of data that can be queried, and also placed at and replicated across compute nodes. (By default the realtime node can serve queries accessing a segment until it is handed off to a dedicated compute node.)

Compute nodes residing in a frontend cluster are used to serve queries by loading appropriate segments from the backend tier. These compute nodes are called *historical nodes (HNs)*, and we use these terms interchangeably.

The *coordinator node* handles data management. Upon seeing a segment being created, it selects a suitable compute node (HN) to load the new segment. The coordinator can ask multiple HNs to load the segment thereby creating *segment replicas*. Once loaded, the HNs can start serving queries which access this segment.

Clients send queries to a frontend router, also called *broker*. A broker node maintains a view of which nodes (historical/realtime) are currently storing which segments. A typical query accesses multiple segments. The broker routes the query to the relevant HNs in parallel, collates or aggregates the responses, and sends it back to the client.

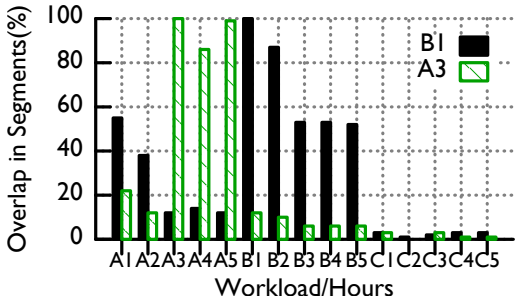


Figure 4.3: Measures overlap in segment accesses across different hours of Yahoo! production trace. Each trace identified with an id (A/B/C: see Table 4.1) and the ith hour.

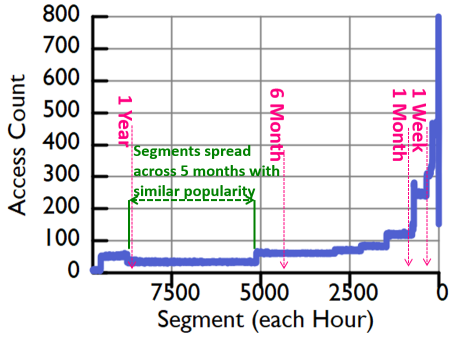


Figure 4.4: Popularity of Segments collected from Yahoo! production trace. X axis represents segments ordered in increasing order of creation time. Y-axis plots the number of accesses each segment saw in a 5 hour trace from Yahoo!.

In Druid, all internal coordination like segment loading between coordinator and HN is handled by a Zookeeper [93] cluster. Druid also uses MySQL [94] for storing metadata from segments and failure recovery. As a result, the coordinator, broker, and historical nodes are all stateless. This enables fast recovery by spinning up a new machine.

4.2.2 Workload Insights

Name	Month	Total Segments	Total Accesses
A	October 2016	0.6K	65K
B	January 2017	9.3K	0.8M
C	February 2017	1.3K	64K

Table 4.1: Druid traces from Yahoo! production clusters.

We analyze Yahoo!'s production Druid cluster workloads, spanning several hundreds of machines, and many months of segments (segments are hourly). Each of the three workload

traces shown in Table 4.1 spans 5 hours, but at different times over 2 years. Total segments reflect the working set size, and total accesses reflect workload size (query-segment pairs). We draw two useful observations:

Segment Access is skewed, and recent segments are generally more popular: Figure 4.1 plots the CDF of the access counts for trace B (other traces yielded similar trends and are not shown). The popularity is skewed: the top 1% of segments are accessed an order of magnitude more than the bottom 40% segments. While this skew has been shown in batch processing systems [31], we are the first to confirm it for interactive analytics systems. The skewed workload implies that some segments are more important and selective replication is needed.

Figure 4.4 shows the number of times a segment was accessed in the trace B. That is, the 4000th data point shows the total access count for the segment created 4000 hours before this trace was captured. We observe that segments are most popular 3 to 8 hours after creation, and this popularity is about 2× more than that of segments that are a week old. However, a few very old segments continue to stay popular (e.g., bumps at about a year ago) [4].

Some (older) segments continue to stay popular: Figure 4.3 shows the level of overlap between segments accessed during an hour of the Yahoo! trace (shown on the horizontal axis), vs. a reference hour (B1, A3). Here, "overlap" is defined as Jaccard Similarity Coefficient [95] – the size of the intersection divided by the size of the union, across the two sets of segment accesses.

First, we observe a 50% overlap of segments in A1 with B1 and 40% between A2 and B1. This large overlap across traces collected 5 months apart confirms that some select old segments may be popular (for a while) even in the future long after they are created.

Second, the high overlap among the segments in hours A3 through A5 and B1 through B5 indicates that segments generated nearby in time are highly likely to be queried together, and the length of such a temporal locality is at least 3 hours. This gives any replication policy ample time to adjust replication levels.

4.3 STATIC VERSION OF SEGMENT REPLICATION PROBLEM

We formally define the static problem (§4.3.1), and our solution (§4.3.2).

[4]This is sometimes due to interesting events such as Thanksgiving or holiday weeks.

4.3.1 Problem Formulation

Given m segments, n historical nodes (HNs), and k queries that access a subset of these segments, our goal is to find a segment allocation (segment assignment to HNs) that both: 1) minimizes total runtime (makespan), and 2) minimizes the total number of segment replicas. For simplicity we assume: a) each query takes unit time to process each segment it accesses, b) initially HNs have no segments loaded, and c) HNs are homogeneous in computation power. Our implementation (§4.4) relaxes these assumptions.

Consider the query-segment pairs in the given static workload, i.e., all pairs (Q_i, S_j) where query Q_i needs to access segment S_j. Spreading these query-segment pairs uniformly across all HNs, in a load-balanced way, automatically gives a time-optimal schedule: no two HNs finish more than 1 time unit apart from each other. A load balanced assignment is desirable as it *always* achieves the minimum runtime (makespan) for the set of queries. However, arbitrarily (or randomly) assigning query-segment pairs to HNs may not minimize the total amount of replication across HNs.

Consider an example with 6 queries accessing 4 segments. The access characteristics C for the 4 segments are: $\{S_1{:}6,\ S_2{:}3,\ S_3{:}2,\ S_4{:}1\}$. In other words, 6 queries access segment S_1, 3 access S_2 and so on. A possible time-optimal (balanced) assignment of the query-segment pair could be: bin $HN_1 = \{S_1{:}3,\ S_2{:}1\}$, $HN_2 = \{S_2{:}2,\ S_3{:}1,\ S_4{:}1\}$, $HN_3 = \{S_1{:}3,\ S_3{:}1\}$. However, this assignment is not optimal in replication factor (and thus storage). The total number of replicas stored in the HNs in this assignment is 7. The minimum number of replicas required for this example is 5. An allocation that achieves this minimum is: $HN_1 = \{S_1{:}4\}$, $HN_2 = \{S_2{:}3,\ S_4{:}1\}$, $HN_3 = \{S_1{:}2,\ S_3{:}2\}$ (Figure 4.5).

Formally, the input to our problem is: 1) segment access counts $C = \{c_1, \ldots c_m\}$ for k queries accessing m segments, and 2) n HNs each with capacity $\lceil \frac{\sum_i c_i}{n} \rceil$ (in our chapter, "capacity" always means "compute capacity"). We wish to find: *Allocation* $X = \{x_{ij} = 1, \text{if segment } i \text{ is replicated at HN } j\}$, such that it minimizes $\sum_i \sum_j x_{ij}$.

We solve this problem as a *colored* variant of the traditional bin packing problem [96]. A query-segment pair is treated as a *ball* and a HN represents a *bin*. Each segment is represented by a *color*, and there are as many balls of a color as there are queries accessing it. The number of distinct colors assigned to a bin (HN) is the number of segment replicas this HN needs to store. The problem is then to place the balls in the bins in a load-balanced way that minimizes the number of "splits" for all colors, i.e., the number of bins each color is present in, summed up across all colors. This number of splits is the same as the total number of segment replicas. Unlike traditional bin packing which is NP-hard, this version of the problem is solvable in polynomial time.

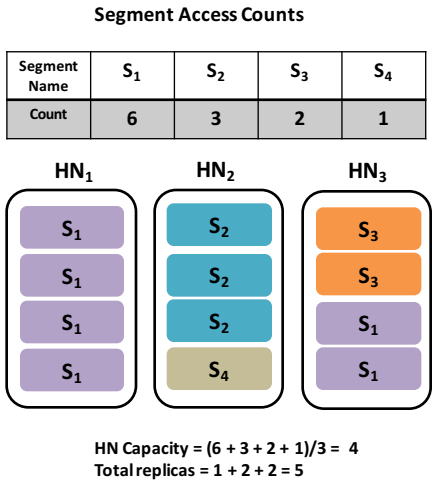


Figure 4.5: Problem depicted with balls and bin. Query-segment pairs are balls and historical nodes represent bins. All balls of same color access the same segment. HN capacity refers to compute capacity. Optimal assignment shown.

4.3.2 Solution

Algorithm 4.1 depicts our solution to the problem. The algorithm maintains a priority queue of segments, sorted in decreasing order of popularity (i.e., number of queries accessing the segment). The algorithm works iteratively: in each iteration it extracts the next segment S_j from the head of the queue, and allocates the query-segment pairs corresponding to that segment to a HN, selected based on a heuristic called ChooseHistoricalNode. If the selected HN's current capacity is insufficient to accommodate all the pairs, then the remaining available compute capacity in that HN is filled with a subset of it. Subsequently, the segment's count is updated to reflect remaining unallocated query-segment pairs, and finally, the segment is re-inserted back into the priority queue at the appropriate position.

The total number of iterations in this algorithm equals the total number of replicas created across the cluster. The algorithm takes time $O((n+m)\cdot log(m))$, i.e., in each iteration either you finish a color or you fill up a bin. This upper bound is loose and in practice it is significantly faster.

The ChooseHistoricalNode problem bears similarities with segmentation in traditional operating systems [97]. We explored three strategies to solve ChooseHistoricalNode: First Fit, Largest Fit, and Best Fit. Of the three, we only describe Best Fit here as it gives an optimal allocation.

Best Fit for ChooseHistoricalNode: In each iteration, we choose the next HN that would have the least compute capacity (space, or number of slots for balls) remaining after accommodating all the queries for the picked segment (head of queue). Ties are broken by

```
   input: C: Access counts for each segment
          nodelist: List of HNs
 1 Algorithm ModifiedFit(C, nodelist)
 2     n ← Length(nodelist)
 3     capacity ← ⌈(Σ_{C_i∈C} |C_i|) / n⌉
 4     binCap ← InitArray(n, capacity)
 5     priorityQueue ← BuildMaxHeap(C)
 6     while !Empty(priorityQueue) do
 7         (segment, count) ← Extract(priorityQueue)
 8         (left, bin) ← ChooseHistoricalNode
 9                       (count, binCap)
10         LoadSegment(nodelist, bin, segment)
11         if left > 0 then
12             Insert(priorityQueue, (segment, left))
13         end
14     end
```

Algorithm 4.1: Generalized Allocation Algorithm.

picking the lower HN id. If none of the nodes have sufficient capacity to fit all the queries for the picked segment, we default to Largest Fit for this iteration, i.e., we choose the HN with the largest available capacity (ties broken by lower HN id), fill it as much as possible, and re-insert unassigned queries for the segment back into the sorted queue.

We call this algorithm ModifiedBestFit. Consider our running example (Figure 4.5) where C is $\{S_1{:}6, S_2{:}3, S_3{:}2, S_4{:}1\}$. The algorithm assigns S_1 to HN_1 and S_2 to HN_2. Next, it picks segment S_1 (again tie broken with S_3) and assigns it to HN_3 because it has sufficient space to fit all the balls. The final assignment produced is optimal in both makespan and replication factor.

4.3.3 Optimality Proof

We now formally prove that ModifiedBestFit minimizes the amount of replication among all load balanced assignments.

Balls and Bin Problem: For ease of exposition, we restate the problem using the balls and bins abstraction. We have m balls of p colors ($p \leq m$) and n bins. The bins have capacity $\lceil \frac{m}{n} \rceil$. There are many load balanced assignments possible for the balls in the bins. The cost of each bin (in a given assignment) is calculated by counting the number of unique color balls in it. The sum of bin costs gives the cost of the assignment. This cost is equivalent to the number of replicas created by our algorithm in §4.3.1. We claim that ModifiedBestFit

minimizes the cost for a load balanced assignment of balls in bins.

Lemma 4.1 *Using* ModifiedBestFit *algorithm, no pair of HNs (bins) can have more than 1 segment (color) in common.*

Proof: Assume there is a pair of bins b_1 and b_2 that have 2 colors in common, c_1 and c_2. Either of c_1 or c_2 must have been selected first to be placed. W.l.o.g. assume c_1 was selected first (in the ordering of colors during the assignment). Since c_1 is split across b_1 and b_2, it must have filled one of the bins. However, this means that c_2 could not have been in bin b_1 as it is selected only afterwards. This contradicts our assumption.
Next, we define an important operation called *swap*.

Swap Operation: A 2-way swap operation takes an equal number of balls from 2 bins and swaps them. A k-way swap similarly creates a chain (closed loop) of k swaps across k bins.

Lemma 4.2 *A k-way swap, involving k HNs, is equivalent to k 2-way swaps.*

Proof: We prove this by induction.

Base Step: Trivially true when $k = 2$.

Induction Step: Assume a k-way swap is equivalent to k 2-way swaps. Let us add another $(k+1)th$ node HN_{k+1} to a k-way chain $HN_1, HN_2, \ldots, HN_k$ to make a $(k+1) - way$ swap chain. However, this can be written as a series of 2-way swaps: i) a k-way swap, executed as $(k-1)$ 2-way swaps among $HN_1, HN_2, \ldots, HN_k$ (as in the induction step, but skipping the last swap), followed by ii) a 2-way swap between HN_k and HN_{k+1}, and then iii) a 2-way swap between node HN_{k+1} and HN_1. This creates a chain of $(k+1)$ 2-way swaps.

Lemma 4.3 *No sequence of 2-way swaps, applied to the* ModifiedBestFit *algorithm's output, can further reduce the number of segment replicas (color splits).*

Proof:

Let's define *successful swap* as a swap which reduces the assignment cost (sum of unique colors across all bins). Note that for a successful 2-way swap, a prerequisite is the existence of at least one common color across both bins in the successful swap.

We prove this by contradiction. Lets say a successful swap is possible. From Lemma 4.1, we know that there is at most one common color between any pair of bins. (Note that by definition, a swap must move back an equal number of balls from b_2 to b_1.) This means that there exist 2 such bins whose common color ball can be moved completely to one of the bins without causing additional color splits due to the balls moved back from b_2 to b_1.

Lets assume that bins b_1 and b_2 have common balls of green color in them. Bin b_1 has n_1 green color balls and bin b_2 has n_2 balls of the same color. W.l.o.g. assume all the green color balls from bin b_1 are moved to b_2, in order to consolidate balls (and therefore lower the number of color splits). An equal number of balls need to be moved back. Three cases arise:

- $n_1 > n_2$**:** In the original assignment order of balls into bins, consider the first instance when green color balls were assigned to either bin b_1 or bin b_2. Since $n_1 > n_2$, then it must be true that bin b_1 must have filled with color green before color green hit b_2 – this can be proved by contradiction. If b_2 had filled first instead, either: 1) all (n_1+n_2) balls would have fit in b_2 (which did not occur), or 2) b_2's n_2-sized hole must have been larger than b_1's n_1-sized hole (which is not true). Essentially bin b_1 was selected first because it had the largest hole (this is Best Fit, and since none of the holes are large enough to accommodate all green color balls, we pick the largest hole).

  Next, in the swapping operation, we swap n_1 green color balls from b_1 to b_2. Thus we need to find n_1 balls from b_2 to swap back. When n_1 balls of green color were put into b_1, it is not possible that b_2 had n_1 or more empty slots available (otherwise b_2 would have been picked for n_1 instead of b_1). This means that to find n_1 balls to swap back from b_2, we have to pick from balls that arrived *before* color green did. But by definition, any such color (say, red) would have had at least (n_1+n_2) balls (due to the priority order), and because b_2 still has holes when green color arrives later, any such previously red-colored balls would have been wholly put into b_2. However, picking this color for swapping would cause a further split (in color red) as we can only move back $n_1(< n_1 + n_2)$ balls from b_2 to b_1. This means that the swap cannot be successful.

- $n_1 < n_2$**:** Analogous to Case 1, we can show that bin b_2 filled first with color green before bin b_1 did. To find n_1 balls to move back from bin b_2 to b_1, we have to choose among balls that arrived before color green in bin b_2, since green color was the last to arrive at b_2 (i.e., filled it out). But any such previous color red must have at least $(n_1 + n_2)$ balls in b_2 (due to the priority order), and choosing red would create an additional color split (in color red). This cannot be a successful swap.

- $n_1 = n_2$**:** W.l.o.g., assume b_1 was filled first with n_1 green color balls, then after some intermediate bins were filled, n_2 green color balls were put into b_2. All such intermediate bins must also have had exactly n_1-sized holes (due to the priority order, Best Fit strategy, and presence of n_2 color green balls in the queue). Bin b_2 cannot get any of these intermediate balls as it cannot have more than n_1 slots when b_1 was filled with green color (otherwise it would have been picked instead of b_1). For our

swap operation, this means one can only choose to swap back a color red (from b_2 to b_1) that was put into b_2 *before* b_1 was filled with green color. However, this means color red must have had at least $(n_1 + n_2)$ balls put into b_2 (due to the priority order), and moving back only some of these balls will cause an additional split (for red). This cannot be a successful swap.

Since a k-way swap is equivalent to k 2-way swaps (Lemma 4.2), no swap strategy can further reduce the number of segment replicas, computed by ModifiedBestFit.

Theorem 4.4 *Given a set of queries,* ModifiedBestFit *minimizes both total number of segment replicas and makespan.*

Proof: By Lemma 4.3, ModifiedBestFit generates load balanced allocation that minimizes the sum of unique color balls across all bins, which in turn minimizes replication. Load balanced allocation of query-segment pairs implies the completion time is minimized.

4.4 GETAFIX: SYSTEM DESIGN

The Getafix system is intended to handle dynamically arriving segments as well as queries. Figure 4.6 shows the general architecture. Most of Getafix's logic resides in the Coordinator. The coordinator manages the segment replicas, runs the ModifiedBestFit algorithm (§4.4.1) to create a logical plan for segment allotment, and then translates the logical plan into a physical one for replication. Additionally, it balances segment load among HNs (§4.4.3) and handles heterogeneity in a deployed cluster (§4.4.4). We modified the broker code to implement different query routing strategies (§4.4.2).

4.4.1 Segment Replication Algorithm

For the dynamic scenario, Getafix leverages the static solution from §4.3.2, by running it in periodic rounds. At the end of each round, it collects query load statistics, then runs the algorithm. The algorithm returns a *segment placement plan*, a one-to-many mapping of segment to HNs where they should be placed for the current round. The placement plan dictates whether a segment needs to be loaded to a HN or removed. In this way, the placement plan implicitly controls the number of replicas for a segment in each round. While it may appear that reducing replication factor reduces query parallelism, our scheme is in fact auto-replicative, which means that popular segments will be replicated more.

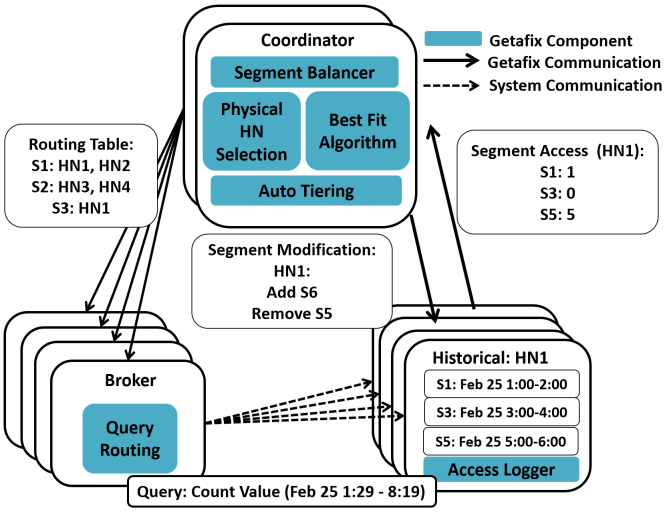


Figure 4.6: Getafix Architecture.

Getafix tracks popularity by having HNs track the total access time for each segment it hosts, during the round. Total access time is the amount of time queries spend computing on a segment. When the round ends, HNs communicate their segment access times to the coordinator and reset these counters. The coordinator calculates popularity via an exponentially weighted moving average. Popularity for segment s_j at round $(K+1)$ is calculated as:

$$\begin{aligned} \textsc{Popularity}(s_j, K+1) = \frac{1}{2} \times \textsc{Popularity}(s_j, K) \\ + \textsc{AccessTime}(s_j, K+1) \end{aligned} \tag{4.1}$$

Next, the coordinator runs ModifiedBestFit using Popularity(.) values. Since the static algorithm assumes logical nodes, these need to be mapped to physical HNs. We describe two mapping approaches later (§4.4.5 and §4.4.4). The round duration cannot be too long (or else the system will adapt slowly) or too short (or else the system may not have time to collect statistics and may thrash). Our implementation sets the round duration to 5 s, which allows us to catch popularity changes early but not react too aggressively. This duration can be chosen based on the segment creation frequency.

4.4.2 Query Routing

Query routing decides which HNs should run an incoming query (accessing multiple segments). We explore two types of routing schemes:

Allocation Based Query Routing (ABR): Apart from segment placement, ModifiedBestFit also provides sufficient information to build a query routing table. Concretely, ModifiedBestFit

proportionally allocates the total CPU time among each replica of a segment. In our running example (Figure 4.5), segment S_1 requires 6 CPU time units of which 4 should get handled by the replica in HN_1 and 2 by the replica in HN_3. This means that 67% of the total CPU resource required by S_1 should be allocated to HN_1, and 33% to HN_3. Hence Getafix creates a routing table that captures exact *query proportions*. The full routing table for this example is depicted in Table 4.2.

	HN_1	HN_2	HN_3
S_1	67	0	33
S_2	0	100	0
S_3	0	0	100
S_4	0	100	0

Table 4.2: Routing Table for Figure 4.5. Each entry represents percentage of queries accessing segment S_i to be routed to HN_j.

Brokers receive queries from clients. After each round the coordinator sends the routing table to the brokers. For a received query, the broker estimates its runtime (based on historical runtime data) and routes it to a HN probabilistically according to the routing table.

Load Based Query Routing (LBR): In ABR, routing table updates happen periodically. Because queries complete much faster than a round duration, ABR lags in adapting to fast changes in workload. With Load Based Routing (LBR), each broker keeps an estimate of every HN's current load. Load is calculated as the number of open connections between the broker and HN. An incoming query (or part thereof), which needs to access a segment, is routed to the HN that: a) has the segment already replicated at it, and b) is the least loaded among all such HNs. Although brokers do not have a global view of the HN load and do not use sophisticated queue estimation techniques [98], this scheme works well in our evaluations (§4.5.5) because of its small overhead.

4.4.3 Balancing Segment Load

For skewed segment access distributions (Figure 4.1), the output of ModifiedBestFit could produce imbalanced assignment of segments to HNs. We wish to minimize the maximum memory used by any HN in the system in order to achieve segment balancing. Additionally, we observed that less-loaded HNs (e.g., those with fewer segments) could be idle in some scenarios (e.g., if some segments became unpopular). In traditional systems, such imbalances require continuous intervention by human operators. We describe an automated segment balancing strategy that avoids this manual work, and both reduces the max memory

and increases overall CPU utilization across HNs.

Our algorithm is greedy in nature and run after every ModifiedBestFit round. We define *segment load* of a HN as the number of segments assigned to that HN. Starting with the output of ModifiedBestFit, the Coordinator first considers those HNs whose segment load is higher than the system-wide average. For each such HN, it picks its k least-popular replicas, where k is the difference between the HN's segment load and the system-wide average. These are added to a global *re-assign* list. Next, the coordinator sorts the replicas in the re-assign list in order of increasing query load. Query load of a segment replica in a HN is the value of the corresponding routing table entry. It picks one replica at a time from this list and assigns it to the HN that satisfies all the following conditions: 1) it does not already host a replica of that segment, 2) the *query load imbalance* after the re-assignment will be $\leq$ parameter γ, and 3) it has the least segment load of all such HNs. We calculate:

$$query\ load\ imbalance = 1 - \frac{min(QueryLoad(HN_i))}{max(QueryLoad(HN_i))} \tag{4.2}$$

In our evaluation (§4.5.3), we found that a default $\gamma = 20\%$ gives the best segment balance with minimal impact on query load balance.

4.4.4 Handling Cluster Heterogeneity

ModifiedBestFit assumes a homogeneous cluster consisting of machines with equal compute capacity (§4.3.1). We now relax that assumption and present modifications for heterogeneous settings.

Capacity-Aware ModifiedBestFit: Instead of assuming equal capacity in Algorithm 4.1 (line 3), we distribute the total query load proportionally among HNs based on their estimated *compute capacities*. To estimate the capacity of a HN, Getafix calculates the CPU time (in microseconds) spent on processing queries at that HN (disk IO is ignored). This data is collected by the coordinator. Finer-grained capacity estimation techniques could be used instead [99].

Stragglers: Some nodes may become stragglers due to bad memory, slow disk, flaky NIC, background tasks, etc. Capacity-Aware ModifiedBestFit approach handles stragglers implicitly. Straggler nodes will report low query CPU times as they would be busy doing I/O and/or waiting for available cores. Capacity-Aware ModifiedBestFit will assign lesser capacity to these node. Lesser capacity will ensure popular segments are not assigned to these HN.

Avoiding Manual Tiering: Today system administrators manually configure clusters into tiers by grouping machines with similar hardware characteristics into a single tier. They use hardcoded rules for placing segments within these tiers, with recent (popular) segments assigned to the hot tier. Eschewing this manual approach, Getafix continuously tracks changes in segment popularity and cluster configuration, to automatically move popular replicas to powerful HNs, thereby creating its own tiers. Thus, Getafix can help avoid laborious sysadmin activity and cut opex (operational expenses) of the cluster.

4.4.5 Minimizing Network Transfers

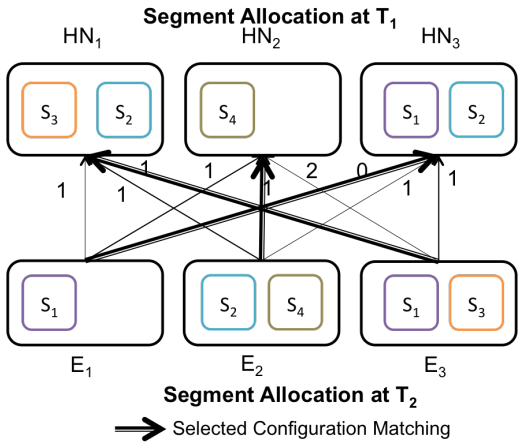


Figure 4.7: Physical HN Mapping problem from Figure 4.5 represented as a bipartite graph.

Although Auto-Tiering can improve query performance in a heterogeneous setting, it is unaware of underlying network bandwidth constraints. Network bandwidth between HNs and deep storage in today's public clouds is often subject to provider-enforced limits [100]. We next discuss approaches that make Getafix network aware.

Consider the example shown in Figure 4.7. In the configuration at time T_1 (top part of figure), HN_1 has segments S_2 and S_3, HN_2 has S_4 only, and HN_3 has segments S_1 and S_2. At time T_2, ModifiedBestFit expects the following configuration: $E_1 = \{S_1\}$, $E_2 = \{S_2, S_4\}$, $E_3 = \{S_1, S_3\}$. If each HN_i chooses to host the segments in E_i, then the algorithm needs to fetch 3 segments in total. However the minimum required is 2, given by the following assignment: E_1 to HN_3, E_2 to HN_2, E_3 to HN_1.

We model this problem as a bipartite graph shown in Figure 4.7 where vertices on the bottom represent expected configurations (E_j) and vertices on the top represent HNs (HN_i) with the current set of replicas. An $HN_i - E_j$ edge represents the network cost to transfer all of E_j's segments to HN_i (except those already at HN_i). Network transfer is minimized by

finding the minimum cost matching in this bipartite graph. We use the classical Hungarian Algorithm [101] to find the minimum matching. It has a complexity of $O(n^3)$ where n is the number of HNs. This is acceptable because interactive data analytics engine clusters only have a few hundred nodes. The coordinator uses the results to set up data transfers for the segments to appropriate HNs.

4.4.6 Bootstrapping of Segment Loading

To be able to serve queries right away, we preload segments at creation time. Concretely whenever a new segment is introduced from external datasources (created at a realtime node), Getafix immediately and eagerly replicates once at a random HN, independent of whether queries are requesting to access it. This cuts down segment loading time for the first few queries to touch a new segment.

Later, our replication may create more replicas (depending on segment popularity). This is preferable than letting the realtime nodes handle queries for fresh segments (the approach used in today's Druid system), which overloads the realtime node. This early bootstrapping also allows segment count calculation to start early.

For cases where a query fails due to the segment not being present on any of the HNs, Getafix re-runs the query. This could happen, for instance, if the segment was unpopular for a long duration and was garbage collected from the HNs. Just like a fresh segment, this segment is first loaded to a random HN. Unlike Druid which silently ignores the segment and returns an incomplete result, we incur slightly elevated latency but always return a complete and correct answer.

4.4.7 Deleting Unnecessary Segments

The replication count for a given segment (output by ModifiedBestFit) may go down from one round to the next. This may occur because incoming queries are no longer accessing this segment. For instance, in Figure 4.7, segment S_1 is not needed in HN_1 and HN_3 after configuration change. We delete such segments to reduce memory usage. This is in line with Getafix's goal of eagerly and aggressively reducing memory, without filling out memory. This is unlike traditional cache replacement algorithms like LFU [97], etc. which only kicks in when the memory is filled. When the workload is heavy and memory is filled, Getafix garbage collection defaults to LFU.

However, we avoid deletion of all replicas of a given segment at once. In cases where ModifiedBestFit cuts the number of replicas to zero, we still retain a single replica at

one (random) HN. This is in anticipation that the segment may become popular again in the future and hence, avoid the additional network I/O.

4.4.8 Garbage Collection

When memory resources are running low but new segments need to be loaded to HNs, Getafix runs a garbage collector. It uses LFU to remove unpopular segment replicas, but instead of absolute access frequency, Getafix uses POPULARITY(.) values. Garbage collection may completely remove a segment from all HNs, unlike deletion.

4.4.9 Fault-Tolerance

Brokers, HNs, and coordinator, are all stateless entities and after a failure can be spun up within minutes. The only state that we maintain are the segment popularity estimates used by MODIFIEDBESTFIT. We periodically checkpoint this state to a MySQL table every 1 minute. This data is not very large, and involves a few bytes per segment in the working set. In MySQL we use batch updates instead of incremental updates, since MySQL is optimized for bulk writes.

4.5 EVALUATION

We evaluate Getafix on both a private cloud (Emulab [60]) and a public cloud (AWS [102]). We use workload traces from Yahoo!'s production Druid cluster. We summarize our results here:

- Compared to the best existing strategy (Scarlett), Getafix uses 1.45 - 2.15× less memory, while minimally affecting makespan and query latency.
- Compared to uniform replication (a common strategy used today in Druid) Getafix improves average query latency by 44 - 55% while using 4 - 10× less memory.
- Capacity-Aware MODIFIEDBESTFIT improves tail query latency by 54% when 10% of the nodes are slow and by 17 - 22% when there is a mix of nodes in the cluster. We also save 17 - 27% in total memory used for the second case. In addition, it can automatically tier a heterogeneous cluster with an accuracy of 75%.

4.5.1 Methodology

Experimental Setup. We run our experiments in two different clusters:

- **Emulab**: We deploy Druid on dedicated machines as well as on Docker [103] containers (to constrain disk for GC experiment). We use d430 [104] machines each with two 2.4 GHz 64-bit 8-Core processor, 64 GB RAM, connected using a 10Gbps network. We use NFS as the deep storage.
- **AWS**: We use m4.4xlarge [99] instances (16 cores, 64 GB memory), S3 [25] as the deep storage, and Amazon EBS General Purpose SSD (gp2) volumes [105] as node local disks. EBS volumes can elastically scale out when the allocation gets saturated.

Workloads. Data is streamed via Kafka into a Druid realtime node. Typically, Druid queries summarize results collected in a time range. In other words, each query has a start time and an interval. We pick start and interval times based on production workloads– concretely we used a trace data set from Yahoo! (similar to Figure 4), and derive a representative distribution. We then used this distribution to set start times and interval lengths.

We generate a query mixture of timeseries, top-K and groupby. Each query type has its own execution profile. For example, groupby queries take longer to execute compared to top-K and timeseries. There can be considerable deviation in runtime among groupby queries themselves based on how many dimensions are queried. Other than the time interval, we do not vary other parameters for these individual query types.

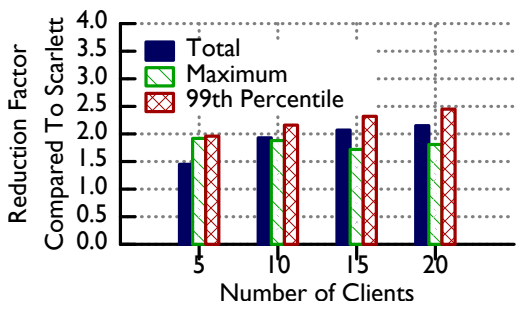
In our experiments, a workload generator client has its own broker to which it sends all its queries. Each client randomly picks a query mix ratio, and query injection rate between 50 and 150 queries/s. Instead of increasing per-client query rate (which would cause congestion due to throttling at both client and server), we scale query rates by linearly increasing numbers of clients and brokers. Each experiment (ingestion and workload generator) are run for 30 minutes.

Baselines. We compare Getafix against two baselines:

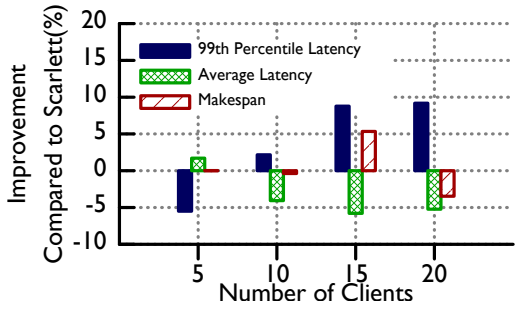
- **Scarlett:** Scarlett [31] is the closest existing system that handles skewed popularity of data. While the original implementation of Scarlett is intended for Hadoop, its ideas are general. Hence we re-implemented Scarlett's techniques into Druid (around 2000 lines of code).

  In particular, we implemented Scarlett's round-robin based algorithm [5]. The round-robin algorithm counts the number of *concurrent accesses to a segment*, as an indicator of popularity. Scarlett gives more replicas to segments with more concurrent accesses. We collect the concurrent segment access statistics from the historical nodes (HNs) and send it to the coordinator to calculate and modify the number of replicas for each segment. The algorithm uses a configurable network budget parameter. Since we did not cap network

[5]We avoid the priority-based algorithm since it is intended for variable file sizes, but segment sizes in interactive analytics engines are in the same ballpark.



(a) Scarlett memory divided by Getafix total memory. Higher is better.



(b) Reduction in Makespan, Average and 99th Percentile Latency of Getafix compared to Scarlett. Higher is better.

Figure 4.8: AWS Experiments: Getafix vs. Scarlett with increasing load (number of client varying from 5 to 20).

budget usage in Getafix, we do not do it for Scarlett (for fairness in comparison).

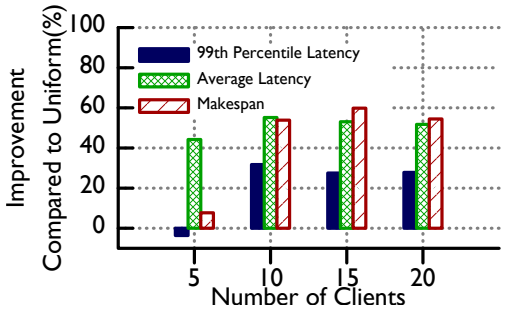
- **Uniform:** We compare our system to the simple (but popular in Druid deployments today) approach where all segments are uniformly replicated. We vary the replication factor (RF) across experiments.

Metrics. Across the entire run, we measure: 1) total memory used across all HNs, 2) maximum memory used across all HNs, and 3) effective replication factor. Effective replication factor is calculated as the total number of replicas created by a system, divided by the total number of segments ingested by the system. This metric is useful to estimate the memory requirements of an individual machine while provisioning a cluster (§4.6). We also measure: 1) average and 99th percentile (tail) query latency and 2) makespan.

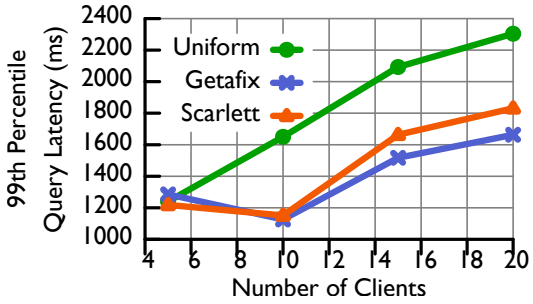
To calculate memory dollar cost savings in a public cloud, we multiply the memory savings with cost per GB of memory. We calculate the cost of 1 GB memory in a public cloud by solving a set of linear equations (elided for brevity) derived from the published instance type prices. For AWS, memory cost is $0.005 per GB per hour.

4.5.2 Comparison against Baselines

Comparison vs. Scarlett: We increase the query load (number of workload generator clients varied from 5 to 20) while keeping the compute capacity (HNs) fixed (20). Figure 4.8a plots the savings in Getafix's memory usage compared to Scarlett's. Getafix uses 1.45 - 2.15× less total memory (across HNs), and 1.72 - 1.92× less maximum memory in a single HN. Scarlett alleviates query hotspots by creating more replicas of popular segments, while Getafix carefully balances replicas of popular and unpopular segments to keep overall replication (and memory usage) low. Getafix's memory savings also increases as more clients



(a) **Improvement in makespan, 99th percentile and average latency in Getafix compared to Uniform.**



(b) **Input Load vs Tail Latency Tradeoff Curve. Comparing Getafix with Scarlett and Uniform. Lower is better.**

Figure 4.9: AWS Experiments: Getafix vs. Baselines with increasing load (number of client varying from 5 to 20). Uniform uses an order of magnitude more memory compared to Getafix.

are added.

Memory Dollar Cost Savings in Public Cloud: We perform a back of the envelope calculation, based on our experimental numbers. For the 20 HN + 20 client experiment, Getafix has an effective replication factor of 1.9 compared to Scarlett's 4.2. (The heavy-tailed nature of segment popularity from Figure 4.1 implies the very popular segments influence effective replication factor.) In a public cloud deployment, where popular data size is 100 TB [6], Getafix thus can reduce memory usage by approximately 230 TB (100 TB $\times$ (4.2 - 1.9)). This amounts to cost savings of 230 $\times$ 10^3 GB $\times$ \$0.005/GB/hour = \$1150 per hour. Annually, this would amount to \$10 million worth of savings.

To quantify the impact of this memory savings on performance, Figure 4.8b plots the reduction in makespan, average and 99th percentile latency for Getafix compared to Scarlett. Getafix completes all the queries within $\pm$5% of Scarlett for all the experiments. Query latency is also comparable.

We conclude that compared to Scarlett, Getafix significantly reduces memory usage in a private cloud, dollar cost in a public cloud, with small impact on query performance.

Comparison vs. Uniform: We compare Getafix with the Uniform strategy configured to use a replication factor of 4. We increase load (number of clients varied from 5 to 20).

Getafix uses 4 - 10$\times$ less memory than Uniform. For latency and makespan, see Figure 4.9a. Getafix improves average query latency by 44-55%. The reason is that popular segments in the Uniform approach are replicated infrequently compared to Getafix, causing

[6]Most present day production clusters in Google, Yahoo handle petabytes of data [88] per day. Of this only a fraction of the data is most popular and hosted in memory. We conservatively estimated 100 TB as the ballpark of popular data size.

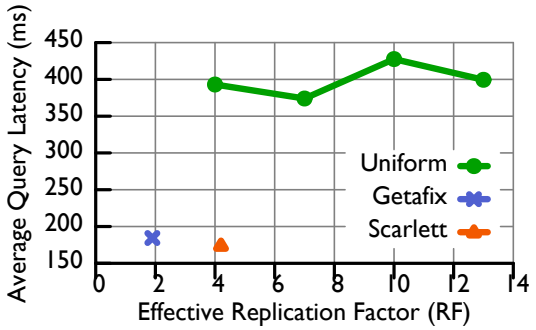


Figure 4.10: AWS Experiments: Memory-Latency Tradeoff Curve. 20 HNs and 15 clients using: 1) Getafix, 2) Scarlett and 3) Uniform (RF: 4, 7, 10, 13). Lower is better on both the axes.

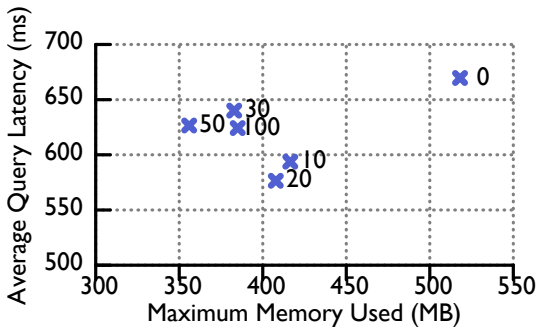


Figure 4.11: Emulab Experiments: Getafix – Maximum memory vs. Query Latency Tradeoff for different γ values. Lower is better on both axes.

hotspots at HNs hosting popular segments, increasing average query latency. We observe improvements in makespan (53 - 59%) and 99th percentile query latency (27 - 31%) for high query load (10 or more clients). The 5 client setting is marginally worse in Getafix because unpopular segments have more replicas in Uniform than in Getafix. Queries accessing these segments form the tail and they run faster in Uniform.

To evaluate Getafix's impact on tail latency, we compare it with Uniform and Scarlett as query load increases. Figure 4.9b plots the 99th percentile tail latency for all three approaches as the number of clients increases. Getafix outperforms the baselines at tail, even as the query load increases.

Memory-Latency Tradeoff: Figure 4.10 plots the memory-latency tradeoff (replication factor vs. average query latency). Points closer to the origin are more preferable. Uniform's tradeoff curve plateaus at a query latency that is 2.15× higher than Getafix and Scarlett. Getafix memory is 3.5× smaller than Uniform and 2.2× smaller than Scarlett.

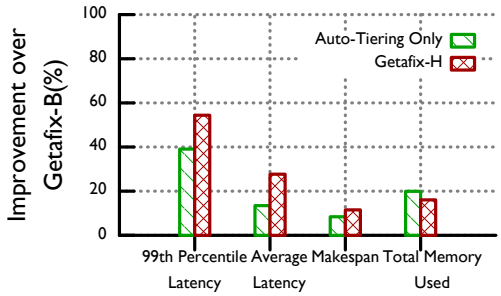


Figure 4.12: Emulab Experiments: Improvement in 99th Percentile, Average Query latency, Makespan and Total Memory Used with Getafix-H compared to Getafix-B. Experiment performed with 2 HNs straggling among 20.

4.5.3 Segment Balancer Tradeoff

In §4.4.3, we introduced a threshold parameter γ that determines the tradeoff space between maximum memory used and query performance. $\gamma = 0\%$ implies no balancing while $\gamma = 100\%$ implies aggressive balancing.

Figure 4.11 quantifies γ's impact in a cluster of 20 HNs and 15 clients (labels are γ values). As we increase γ, maximum memory used decreases (at $\gamma = 50\%$ memory is reduced by 31.3%.). However, latency decreases until $\gamma = 20\%$ and then starts to rise. We observed a similar trend in makespan and 99th percentile latency (elided for brevity). This occurs because of higher CPU utilization at HNs hosting popular segments. At smaller γ, moving a few unpopular segments to such HNs allows the CPU to remain busy while the popular segment is falling in popularity. Too high γ values move popular segments too, hurting performance.

While the above plot shows maximum memory, we also saw savings in total memory. The largest reduction observed was 19.26% when $\gamma = 20\%$. This occurs because better query balance results in faster completion of the queries, which in turn keeps segments in memory for lesser time.

4.5.4 Cluster Heterogeneity

We evaluate the performance of Capacity-Aware ModifiedBestFit (§4.4.4) (labeled Getafix-H). We consider two types of heterogeneous environments: a) Homogeneous cluster with stragglers and b) Heterogeneous cluster with mixed node types. We compare these techniques against baseline Getafix (labeled Getafix-B).

Stragglers: We inject stragglers in a homogeneous Emulab cluster with 20 HNs and 15 clients. Two HNs are manually slowed down by running CPU intensive background tasks,

and creating memory intensive workloads on 32GB memory using the `stress` command.

Capacity-Aware ModifiedBestFit does two things - i) It makes replication decisions based on individual node capacities, and ii) As a consequence of (i), it implicitly does Auto-tiering. To understand the impact of (i) and (ii) separately, we implement a version of Auto-tiering on top of Getafix-B. In that, the replication decisions are made assuming uniform capacity, but the segments are mapped to HNs based on sorted HN capacity. Segments with high CPU time get mapped to HNs with high capacity. We call this the "Auto-tiering Only" scheme.

Figure 4.12 shows Auto-Tiering by itself improves 99th percentile query latency by 40% and reduces average latency by 14% when compared with Getafix-B. With Getafix-H, the overall gains increase to 55% and 28% respectively. Both Auto-Tiering Only and Getafix-H show memory savings (16-20%). Memory improvement with Getafix-H is slightly less than Auto-Tiering Only. We believe this is because Capacity-aware ModifiedBestFit detects straggling HNs as low capacity nodes and allocates lesser segment CPU time on them. As a result, it needs to assign the remaining query load of that segment on other HNs, which results in creating extra replicas. This shows that given a trade-off between reducing memory vs query latency, Capacity-aware ModifiedBestFit chooses the latter.

Tiered Clusters: Experiments are run in AWS on two cluster configurations consisting of mixed EC2 instances as shown in Table 4.3. Cluster-1 has 15 HNs/5 clients and Cluster-2 has 25 HNs/10 clients.

Node type	**Node config (core / memory)**	**Cluster-1**	**Cluster-2**
m4.4xlarge	16 / 64GB	3 nodes	4 nodes
m4.2xlarge	8 / 32GB	6 nodes	6 nodes
m4.xlarge	4 / 16GB	6 nodes	10 nodes

Table 4.3: AWS HN heterogeneous cluster configurations.

Figure 4.13 shows that for Cluster-1, with a core mix of 48:48:24 (hot:warm:cold), Getafix-H improves the 99th percentile latency by 23% and reduces the total memory used by 18%, compared to Getafix-B. Cluster-2 (64:48:40) has higher heterogeneity than Cluster-1. We see that the 99th percentile latency improves by 18% and Total Memory Used reduces by 27%. This shows that even as the heterogeneity gets worse, Getafix-H continues to give improvements in latency, makespan, and memory.

To evaluate how well Getafix-H can help reduce sysadmin load by performing automatic tiering, we draw a heat map in Figure 4.14. HNs are sorted on the x axis with more powerful HNs to the left. The three colors (hot, warm, cold) indicate the effective load capacity of

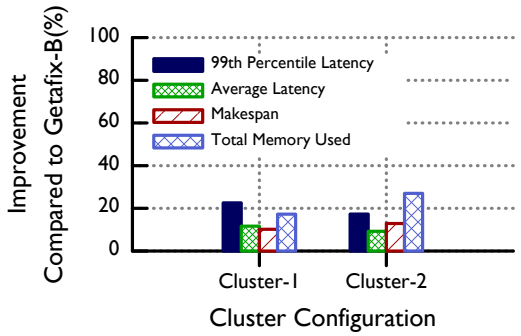


Figure 4.13: AWS Experiments: Improvement in 99th Percentile, Average Query latency, Makespan and Total Memory Used with Getafix-H compared to Getafix-B. Experiments performed with 2 different node mixtures and clients (refer Table 4.3).

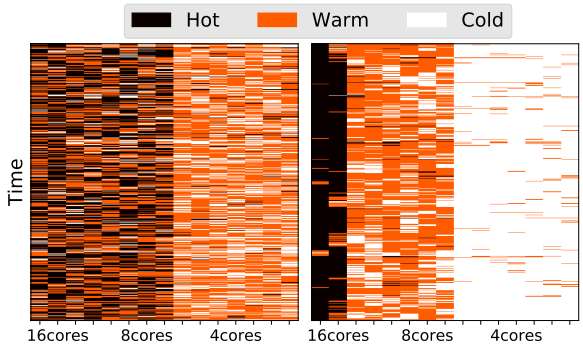


Figure 4.14: AWS Experiments: Getafix-B on left, Getafix-H on right. Effectiveness of Auto-Tiering shown using heat map. X-axis represents HNs sorted by the number of cores they have. Y-axis plots a period of time in the duration of the experiments. For each time, we classify HNs as hot, warm and cold (represented with 3 different colors) based on the reported CPU time for processed queries.

HNs based on our run with Cluster 1. We expect to see three tiers based on Cluster-1 config with 3 HNs assigned to Hot tier and 6 each to Warm and Cold tiers (Table 4.3). Getafix-B (plot on left) fails to tier the cluster in a good way. Visually, Getafix-H achieves better tiering with 3 distinct tiers. Quantitatively, Getafix-B has a tiering accuracy of 42% and Getafix-H has 75% (net improvement of 80%). Accuracy is calculated as number of correct tier assignments divided by overall tier assignments. These numbers can be boosted further with sophisticated HN capacity estimation techniques (beyond our scope).

4.5.5 Comparing Query Routing Schemes

We evaluate three routing schemes, of which two are new: 1) ABR: Allocation Based Query Routing from §4.4.2. 2) LBR-CC (LBR with Connection Count): In this scheme (Druid's default), broker routes queries to that HN with which it has the lowest number of open HTTP connections (indicating low query count). 3) LBR-CC+ML (LBR with Connection

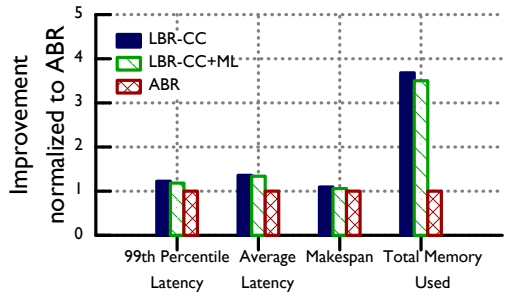


Figure 4.15: Emulab Experiments: Comparing 3 different query routing strategies on Getafix-B – 1) LBR-CC, 2) LBR-CC+ML, 3) ABR. Higher is better.

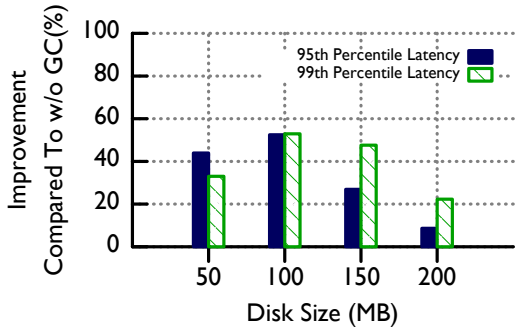


Figure 4.16: Emulab Experiment: Improvement in 99th and 95th percentile query latency for Getafix with GC compared to without GC. Disk sizes are varied from 50 - 200 MB.

Count + Minimum Load): Augments LBR-CC by considering both open HTTP connections and the number of waiting queries at the HN, using their sum as the metric to pick the least loaded HN for the query.

Figure 4.15 compares these schemes on 15 HNs/10 clients homogeneous Emulab cluster. The two LBR schemes are comparable, and are better than ABR, especially on total memory. This difference is because of the following reason. While ABR knows the exact segment allocation proportions, that information is only updated periodically (every round), making ABR slow to react to dynamic cluster conditions and changing segment popularity trends. Overall, Getafix works well with Druid's existing LBR-CC scheme.

4.5.6 Benefit From Garbage Collection

When the data size of the working set (queried data) exceeds the total memory available across the HNs, queries are processed out-of-core (e.g., disk). In such scenarios the Garbage Collector (GC) is crucial to performance–it allows freshly minted segments gaining in popularity to be loaded and queried.

We emulated smaller disks and Figure 4.16 plots the improvement in tail latency (95th and 99th percentile) when the GC is used compared to when GC is disabled. The GC improves tail latency by 25% to 55%. As disk sizes increase, the frontend tier can accommodate more segments and the marginal gain from GC falls. We recommend the GC always be enabled, but especially in circumstances such as a large differential between backend and frontend storage sizes, or low query locality, or wimpy frontend tiers.

4.6 DISCUSSION

Saving Memory costs in practice: System administrators use various techniques to estimate how much memory to provision in an interactive analytics cluster. Some of these are based on workload profiling, which is beyond our scope. However, a rule of thumb to calculate per-HN memory is to multiply the expected working set data size with the effective replication factor and divide by the number of HNs. Since Getafix significantly reduces replication factor ($\sim 2\times$ compared to Scarlett), it can reduce capital expenses (Capex) in a private cloud and dollar expenses in a public cloud. In disaggregated datacenters, Getafix's dollar cost savings would be higher as memory cost is decoupled from CPU costs.

Getafix savings extend to Disk Storage: If one were to increase the working set of (popular) segments without increasing cluster size or per-HN memory, a cutoff point will be reached when cluster memory no longer suffices and HNs will need to use out-of-core memory (e.g., disk). Compared to Druid (uniform) and Scarlett, Getafix reaches this cutoff point much later (at higher dataset sizes). Beyond the cutoff point, Getafix is still preferable to competing systems because it is able to fit more segments in memory, and thus it minimizes disk usage. Extremely large datasets where disk dominates memory are not typical of production scenarios today as high latencies will necessitate scaling out the cluster anyway.

Getafix vs. On-demand Replication: Consider the (alternative) pure on-demand approach which keeps one replica per segment in the cluster, but creates an extra replica on-demand per query. In comparison, Getafix is "sticky" and retains a recently-created replica, expecting that this potentially-popular segment will be used by an impending query. Thus, Getafix will have significantly less network usage and lower query latency than the pure on-demand approach. This is also borne out by the observations from production that query popularity persists for a while, and that segment transfer times are significant.

Overhead of Getafix's Planning Algorithm: In a system with 20 HNs, 15 brokers, 30 segments, Getafix's planning algorithm took a median time of 211.5 ms, much smaller than the reconfiguration period of 5000 ms. This planning overhead is completely hidden from

the end user because queries are scheduled in parallel with this planning.

4.7 RELATED WORK

Allocation Problem: Our problem has similarities to the *data allocation problem* [106] in databases which tries to optimize for performance [107, 108] and/or network bandwidth [109]. A generalized version of the problem has been shown to be NP-hard [106]. Typical heuristics used are best fit and first fit [110, 111] or evolutionary algorithms [108]. This problem is different from the one Getafix solves. In databases, each storage node also acts as a client site generating its own characteristic access pattern. Thus, performance optimization often involves intelligent data localization through placement and replication. On the contrary, brokers in Druid receive client queries and are decoupled from the compute nodes in the system. Getafix aggregates the access statistics from different brokers to make smart segment placement decisions. Some of Getafix's ideas may be applicable in traditional databases.

Workload-Aware Data Management: We are not the first to use popularity for data management. Nectar [112] trades off storage for CPU by not storing unpopular data, instead, recomputing it on the fly. In our setting neither queries generate intermediate data, nor can our input data be regenerated, so Nectar's techniques do not apply. Workload-aware data partitioning and replication has been explored in Schism [113], whose techniques minimize cross-partition transactions in graph databases. There are other works which look at adaptive partitioning for OLTP systems [114] and NoSQL databases [115] respectively, however they do not explore Druid-like interactive analytics engines. E-Store [116] proposes an elastic partition solution for OLTP databases by partitioning data into two tiers. The idea is to assign data with different levels of popularity into different sizes of data chunks so that the system can smoothly handle load peaks and popularity skew. This approach is ad-hoc and an adaptive strategy like Getafix is easier to manage.

Saving Memory and Storage: Facebook's f4 [117] uses erasure codes for "warm" BLOB data like photos, videos, etc., to reduce storage overhead while still ensuring fault tolerance. These are optimizations at the deep storage tier and orthogonal to our work. Parallel work like BlowFish [118], have looked at reducing storage by compressing data while still providing guarantees on performance. It is complementary to our approach and can be combined with Getafix.

Interactive data analytics engines: Current work in interactive data analytics engines [119, 120, 121, 18] focus on query optimization and programming abstractions. They are transparent to the underlying memory challenges of replication and thus, to performance.

In such scenarios, Getafix can be implemented inside the storage substrate [8]. Since Getafix uses data access times and not query semantics, it can reduce memory usage generally.

Amazon Athena [122] and Presto [18] attempt to co-locate queries with the data in HDFS, but these systems do not focus on data management. Details about these systems are sketchy (Athena is closed-source, Presto has no paper), but we believe Getafix's ideas can be amended to work with these systems. Athena's cost model is per TB processed and, we believe, is largely driven by memory usage. Getafix's cost model is finer-grained, and focuses on memory, arguably the most constrained resource today. Nevertheless, these cost models are not mutually exclusive and could be merged.

Systems like Druid [17], Pinot [30], Redshift [87], Mesa [88], couple data management with rich query abstractions. Our implementation inside Druid shows that Getafix is effective in reducing memory for this class of systems, with the exception that Mesa allows updates to data blocks (Getafix, built in Druid, assumes segments are immutable).

Cluster Heterogeneity: Optimizing query performance in heterogeneous environments is well-studied in batch processing systems like Hadoop [123, 124, 125, 126]. Typical approaches involve estimating per job progress and then speculatively re-scheduling execution. Real time system query latencies tend to be sub-second which makes the batch solutions inapplicable.

4.8 SUMMARY

We have presented replication techniques intended for interactive data analytics engines applicable to systems like Druid, Pinot, etc. Our techniques use latest (running) popularity of data segments to determine their placement and replication level at compute nodes. Our solution to the static query/segment placement problem is provably optimal in both makespan and total memory used. Our system, called Getafix, generalizes the solution to the dynamic version of the problem, and effectively integrates adaptive and continuous segment placement/replication with query routing. We implemented Getafix into Druid, the most popular open-source interactive analytics engine. Our experiments use workloads derived from production traces in Yahoo!'s production Druid cluster. Compared to the best existing technique (Scarlett), Getafix uses 1.45 - 2.15× less memory, while minimally affecting makespan. In a public cloud, for a 100 TB hot dataset size, Getafix can cut memory dollar costs by as much as 10 million dollars annually with negligible performance impact.

Chapter 5: Fast Compaction Algorithms for NoSQL Databases

In this chapter, we discuss the problem of compaction of SSTables in log structured databases like Cassandra. We motivate the problem in Section 5.1. We formally define the problem of compaction in Section 5.2. Our theoretical contributions in the form of heuristics and analysis is listed in Section 5.3. Our simulation results using real world workload traces have been presented in Section 5.4.

5.1 INTRODUCTION

Distributed NoSQL storage systems are being increasingly adopted for a wide variety of applications like online shopping, content management, education, finance etc. Fast read/write performance makes them an attractive option for building efficient back-end systems.

Supporting fast reads and writes simultaneously on a large database can be quite challenging in practice [127, 128]. Since today's workloads are write-heavy, many NoSQL databases [52, 24, 129, 42] choose to optimize writes over reads. Figure 5.1 shows a typical write path at a server. 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*.

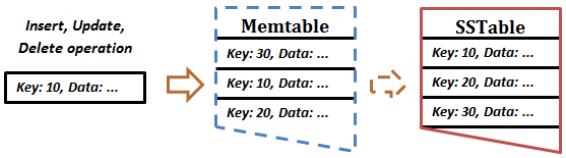


Figure 5.1: Schematic representation of a typical write operations. Dashed box represents a memtable. Solid box represents a sstable. Dashed arrow represents flushing of memory to disk.

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. Figure 5.2 illustrates an example.

In order to minimally affect normal database CRUD (create, read, update, delete) operations, 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

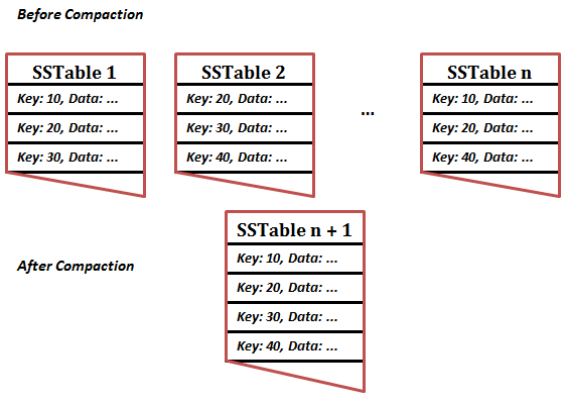


Figure 5.2: A compaction operation merge sorts multiple sstables into one sstable.

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. For the rest of the chapter, we will use the term "disk I/O" to refer to this amount of data. We consider the static version of the problem, i.e., the sstables do not change while compaction is in progress.

In this chapter, 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.

Our Contribution: In this chapter, we study the compaction problem from a theoretical perspective. We formalize the compaction problem as an optimization problem. We further show a generalization of the problem, which can model a wide class of compaction cost functions. Our contributions are as follows:

- Prove that the optimization problem is NP-hard [130].
- Propose a set of greedy algorithms with provable approximation guarantees (Section 5.3).
- Quantitatively evaluate the greedy algorithms with real-life workloads using our implementation (Section 5.4).

Related Work: A practical implementation of compaction was first proposed in Bigtable [6]. 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 [131, 132] merges every insert, update and delete operations instead. They optimize for read performance by sacrificing writes. NoSQL databases like Cassandra [10] and Riak [29] implement both these strategies [133, 134]. Cassandra's Size-Tiered compaction strategy [134], inspired from Google's Bigtable, merges sstables of equal size. This approach bears resemblance to our SmallestInput heuristic defined in Section 5.3. For data which becomes immutable over time, such as logs, recent data is prioritized for compaction [135, 136]. Again, the goal here is to improve read throughput.

Our work looks at a *major compaction* operation. Mathieu et. al. [137] 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.

5.2 PROBLEM DEFINITION

Consider the compaction problem on n sstables for the case where $k = 2$, i.e., in each iteration, 2 sstables are merged into one. As we discussed in Section 4.1, an sstable consists of multiple entries, where each entry has a key and associated values. When 2 sstables are merged, the new sstable is created which contains only one entry per key present in either of the two base sstables. To give a theoretical formulation for the problem, we assume that: 1) all key-value pairs are of the same size, and 2) the value is comprehensive, i.e., contains all columns associated with a key. This makes the size of an sstable proportional to the number of keys it contains. Thus an sstable can be considered as a set of keys and a merge operation on sstables performs simple union of sets (where each sstable is a set). With this intuition, we can model the compaction problem for $k = 2$ as the following optimization problem.

Given a ground set $U = \{e_1, \ldots e_m\}$ of m elements, and a collection of n sets (sstables), $A_1, \ldots, A_n$ where each $A_i \subseteq U$, the goal is to come up with an optimal merge schedule. A merge schedule is an ordered sequence of set union operations that reduces the initial collection of sets to a single set. Consider the collection of sets, initially $A_1, \ldots, A_n$. At

each step we merge two sets (*input sets*) in the collection, where a merge operation consists of removing the two sets from the collection, and adding their union (*output set*) to the collection. The cost of a single merge operation is equal to the sum of the sizes of the two input sets plus the size of the output set in that step. With n initial sets there need to be $(n-1)$ merge operations in a merge schedule, and the total cost of the merge schedule is the sum of the costs of its constituent merge operations.

Observe that any merge schedule with $k = 2$ creates a full[1] binary tree T with n leaves. Each leaf node in the tree corresponds to some initial set A_i, each internal node corresponds to the union of the sets at the two children, and the root node corresponds to the final set. We assume that the leaves of T are numbered $1, \ldots, n$ in some canonical fashion, for example using an in-order traversal. Thus a merge schedule can be viewed as a full binary tree T with n leaves, and a permutation $\pi : [n] \to [n]$ that assigns set A_i (for $1 \leq i \leq n$), to the leaf numbered $\pi(i)$. We call this the *merge tree*. Once the merge tree is fixed, the sets corresponding to the internal nodes are also well defined. We label each node by the set corresponding to that node. By doing a bottom-up traversal one can label each internal node. Let ν be an internal node of such a tree and A_ν be its label. For simplicity, we will use the term *size of node* ν, to denote the cardinality of A_ν.

In our cost function the size of a leaf node or the root node is counted only once. However, for an internal node (non-leaf, non-root node) it is counted twice, once as input, and once as output. Let V' be the set of internal nodes. Formally, we define the cost of the merge schedule as:

$$\mathsf{cost}_{actual}(T, \pi, A_1, \ldots, A_n) = \sum_{\nu \in V'} 2\,|A_\nu| + \sum_{i=1}^{n} |A_i| + |A_{root}| \tag{5.1}$$

Then, the problem of computing the optimal merge schedule is to create a full binary tree T with n leaves, and an assignment π of sets to the leaf nodes such that $\mathsf{cost}_{actual}(T, \pi, A_1, \ldots, A_n)$ is minimized. This cost function can be further simplified as follows:

$$\mathsf{cost}(T, \pi, A_1, \ldots, A_n) = \sum_{\nu \in T} |A_\nu| \tag{5.2}$$

The optimization problems over the two cost functions are equivalent because the size of the leaf nodes, and the root node is constant for a given instance. Further, an α-approximation for $\mathsf{cost}(T, \pi, A_1, \ldots, A_n)$ immediately gives a $2{\cdot}\alpha$-approximation for $\mathsf{cost}_{actual}(T, \pi, A_1, \ldots, A_n)$. For ease of exposition, we use the simplified cost function in equation (5.2) for all the theoretical analysis presented in this chapter. We call this optimization problem as the

[1] A binary tree is *full* if every non-leaf node has two children

BINARYMERGING problem. We denote the optimal cost by $\mathsf{opt}_s(A_1, \ldots, A_n)$.

A Reformulation of the Cost A useful way to reformulate the cost function $\mathsf{cost}(T, \pi, A_1, \ldots, A_n)$ is to count the cost per element of U. Since the cost of each internal node is just the size of the set that labels the node, we can say that the cost receives a contribution of 1 from an element at a node if it appears in the set labeling that node. The cost can now be reformulated in the following manner. For a given element $x \in U$, let $T(x)$ denote the *minimal subtree* of T that spans all the nodes ν in T whose label sets $\pi(\nu)$ contain x and the root node. Let $|T(x)|$ denote the number of edges in $T(x)$. Then we have that:

$$\mathsf{cost}(T, \pi, A_1, \ldots, A_n) = \sum_{x \in U} (|T(x)| + 1). \tag{5.3}$$

Relation to the problem of Huffman Coding We can view the problem of Huffman coding as a special case of the BINARYMERGING problem. Suppose we have n disjoint sets $A_1, \ldots, A_n$ with sizes $|A_i| = p_i$. We can see that, using the full binary tree view and the reformulated cost in equation (5.3), the cost function is the same as the problem of an optimal prefix free code on n characters with frequencies $p_1, \ldots, p_n$.

Generalization of BinaryMerging As we saw, BINARYMERGING models a special case of the compaction problem where in each iteration 2 sstables are merged. However in the more general case, one may merge atmost k sstables in each iteration. To model this, we introduce a natural generalization of the BINARYMERGING problem called the K-WAYMERGING problem. Formally, given a collection of n sets, $A_1, \ldots, A_n$, covering a groundset U of m elements, and a parameter k, the goal is to merge the sets into a single set, such that at each step: 1) atmost k sets are merged and 2) the merge cost is minimized. The cost function is defined similar to BINARYMERGING.

5.3 GREEDY HEURISTICS FOR BINARYMERGING

In this section, we present and analyze four greedy heuristics that approximate an optimal merge schedule. As pointed out in Section 5.2, the BINARYMERGING problem can be viewed as a generalization of the Huffman coding problem. The optimal greedy algorithm for Huffman coding thus motivates the design of three out of our four heuristics, namely: SMALLESTOUTPUT, SMALLESTINPUT and LARGESTMATCH. The BALANCETREE heuristic was conceived because it is easy to parallelize and is thus a natural candidate for a fast implementation unlike others. We prove $O(\log n)$ approximation guarantees for all the heuristics.

This section is organized as follows. We start by giving a lower bound on the cost of the optimal merge schedule. Later, we will use this lower bound to prove the approximation ratio for our greedy heuristics.

5.3.1 A Lower bound on Optimal Cost

We know that $\mathsf{OPT} = \mathsf{opt}_s(A_1, \ldots, A_n)$ is the cost of the optimal merge schedule, see Section 5.2. Let, Cost denote the cost of the merge schedule returned by our algorithm. To give an α-approximate algorithm, we need to show that $\mathsf{Cost} \leq \alpha \cdot \mathsf{OPT}$. Since OPT is not known, we instead show that $\mathsf{Cost} \leq \beta \cdot \mathsf{L_{OPT}}$, where $\mathsf{L_{OPT}}$ is a lower bound on OPT. This gives an approximation bound with respect to OPT itself. Observe that $\mathsf{OPT} \geq \sum_{i=1}^{n} |A_i|$. This follows immediately from the cost function (equation (5.3)), since the cost function size of each node in the merge tree is considered once and sum of the sizes of leaf nodes is $\sum_{i=1}^{n} |A_i|$. Henceforth, we use $\sum_{i=1}^{n} |A_i|$ as $\mathsf{L_{OPT}}$.

5.3.2 Generic Framework for Greedy Algorithm

```
1 Algorithm GreedyBinaryMerging(A_1, ... A_n)
2     C ← {A_1, ..., A_n};
3     for i = 1, ..., n − 1 do
4         S_1, S_2 ← ChooseTwoSets(C);
5         C ← C \ {S_1, S_2};
6         C ← C ∪ {S_1 ∪ S_2};
7     end
```

Algorithm 5.1: Generic greedy algorithm.

The four greedy algorithms we present in this section are special cases of a general approach, which we call the GreedyBinaryMerging algorithm. The algorithm proceeds as follows: at any time it maintains a collection of sets C, initialized to the n input sets $A_1, \ldots A_n$. The algorithm runs iteratively. In each iteration, it calls the subroutine ChooseTwoSets, to choose greedily two sets from the collection C to merge. This subroutine implements the specific greedy heuristic. The two chosen sets are removed from the collection and replaced by their union i.e., the merged set. After $(n-1)$ iterations only 1 set remains in the collection and the algorithm terminates. Details are formally presented in Algorithm 5.1.

5.3.3 Heuristics

We present 4 heuristics for the ChooseTwoSets subroutine in the GreedyBinaryMerging algorithm. We show that three of these heuristics are $O(\log n)$-approximations. To explain the algorithms we will use the following working example:

Working Example 5.1 *We are given as input 5 sets:* $A_1 = \{1, 2, 3, 5\}$, $A_2 = \{1, 2, 3, 4\}$, $A_3 = \{3, 4, 5\}$, $A_4 = \{6, 7, 8\}$, $A_5 = \{7, 8, 9\}$. *The goal is to merge them into a single set such that the merge cost as defined in Section 5.2 is minimized.*

BalanceTree (BT) Heuristic: Assume for simplicity that n is a power of 2. One natural heuristic for the problem is to merge in a way such that the underlying merge tree is a complete binary tree. This can be easily done as follows: the input sets form the leaf nodes or level 1 nodes. The n leaf nodes are arbitrarily divided into $n/2$ pairs. The paired sets are merged to get the level 2 nodes. In general, the level i nodes are arbitrarily divided into $n/2^i$ pairs. Each pair is merged i.e., the corresponding sets are merged to get $n/2^i$ nodes in the $i + 1^{th}$ level. This builds a complete binary tree of height $\log n$.

However, when n is not a power of 2, to create a merge tree of height $\lceil \log n \rceil$ involves a little more technicality. To do this, annotate each set with its level number l, and let $minL$ be the minimum level number across all sets at any point of time. Initially, all the sets are marked with $l = 1$. In each iteration, we choose two sets whose level number is $minL$, merge these sets, and assign the new merged set the level $(minL + 1)$. If only 1 set exists with level number equal to $minL$, we increment its l by 1 and retry the process. Figure 5.3 shows the merge schedule obtained using this heuristic on our working example.

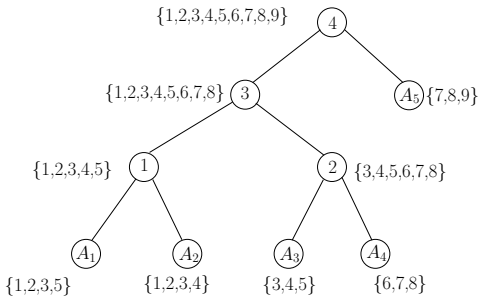


Figure 5.3: Merge schedule using BalanceTree heuristic. The label inside the leaf nodes denotes the corresponding set. The label inside internal nodes denote the iteration in which the merge happened. The sets corresponding to each node is shown outside the node. Cost of the merge = 45.

Lemma 5.2 *Consider an instance* $A_1, \ldots, A_n$ *of the* BinaryMerging *problem.* BalanceTree *heuristic, gives a* $(\lceil \log n \rceil + 1)$*-approximation.*

Proof: Let T be the merge tree constructed. By our level-based construction, $height(T) = \lceil \log n \rceil$. Let C^l denote the collection of sets at level l. Now observe that each set in C^l is either the union of some initial sets, or is an initial set by itself. Also, each initial set participates in the construction of atmost 1 set in C^l. This implies that:

$$\sum_{S \in C^l} |S| \leq \sum_{i=1}^{n} |A_i| = \mathsf{L}_{\mathsf{OPT}} \leq \mathsf{OPT} \tag{5.4}$$

Therefore,

$$\mathsf{Cost} = \sum_{l=1}^{\lceil \log n \rceil + 1} \sum_{S \in C^l} |S| \leq (\lceil \log n \rceil + 1) \cdot \mathsf{OPT} \tag{5.5}$$

Lemma 5.3 *The approximation bound proved for* BalanceTree *in Lemma 5.2 heuristic is tight.*

Proof: We show an example where the merge cost obtained by using BalanceTree heuristic is $\Omega(\log n) \cdot \mathsf{OPT}$. Consider n initial sets where n is a power of 2. The sets are $A_1 = \{1\},\ A_2 = \{1\}, \ldots, A_{n-1} = \{1\},\ A_n = \{1, 2, 3, \ldots n\}$, i.e., we have $(n-1)$ identical sets which contain just the element 1, and one set which has n elements. An optimal merge schedule is the *left-to-right merge*, i.e., it starts by merging A_1 and A_2 to get the set $A_1 \cup A_2$, then merges $A_1 \cup A_2$ with A_3 to get $A_1 \cup A_2 \cup A_3$ and so on. The cost of this merge is $(4n-3)$. However the BalanceTree heuristic creates a complete binary tree of height $\log n$, and the large n size set $\{1, 2, \ldots, n\}$ appears in every level. Thus the cost will be atleast $n \cdot (\log n + 1)$. This lower bounds the approximation ratio of BalanceTree heuristic to $\Omega(\log n)$.

SmallestInput (SI) Heuristic: This heuristic selects in each iteration, those two sets in the collection that have the smallest cardinality. The intuitive reason behind this approach is to defer till later the larger sets and thus, reduce the recurring effect on cost. Figure 5.4 shows the merge tree we obtain when we run the greedy algorithm with SmallestInput heuristic on our working example.

SmallestOutput (SO) Heuristic: In each iteration, this heuristic chooses those two sets in the collection whose union has the least cardinality. The intuition behind this approach is similar to SI. In particular, when the sets $A_1, \ldots, A_n$ are all disjoint, these two heuristics lead to the same algorithm. Figure 5.5 depicts the merge tree we obtain when executed on our working example.

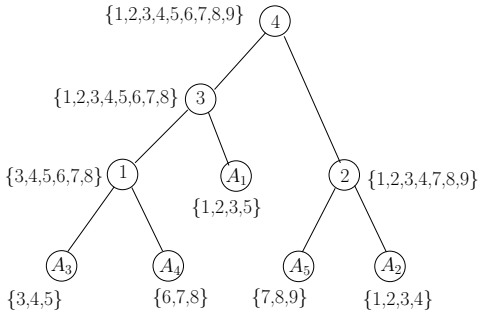


Figure 5.4: Merge schedule using SmallestInput heuristic. Initially the smallest sets are $A_3,\ A_4,\ A_5$. The algorithm arbitrarily choses A_3 and A_4 to merge, creating node 1 with corresponding set $\{3,4,5,6,7,8\}$. Next the algorithm proceeds with merging A_1 and A_2 as they are the current smallest sets in collection, and so on. Cost of the merge = 47.

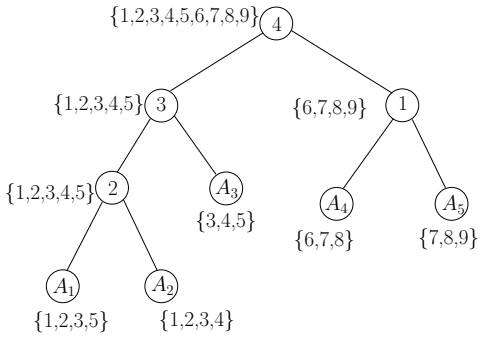


Figure 5.5: Merge schedule using SmallestOutput heuristic. Initially the smallest output set is obtained by merging sets $A_4,\ A_5$. In first iteration $A_4,\ A_5$ is merged to get the new set $\{6,7,8,9\}$. Next the algorithm chooses $A_1,\ A_2$ to merge as they create the smallest output of size 4, and so on. Cost of the merge = 40.

Lemma 5.4 *Given n disjoint sets $A_1, \ldots, A_n$, the* BinaryMerging *problem can be solved optimally using* SmallestInput *(or* SmallestOutput*) heuristics.*

Proof: As we remarked in Section 5.2 that for this special case, the BinaryMerging problem reduces to the Huffman coding problem, and as is well known, the above greedy heuristic is indeed the optimal greedy algorithm for prefix free coding [138].

Lemma 5.5 *Consider an instance $A_1, \ldots, A_n$ of the* BinaryMerging *problem. Both the* SmallestInput *and* SmallestOutput *heuristics, give $O(\log n)$ approximate solutions.*

Proof:

Let $A_1^j, , \ldots, A_{n-j}^j$, be the sets left after the j^{th} iteration of the algorithm. Now observe that each A_i^j is either the union of some initial sets, or is an initial set itself. Further each

initial set contributes to exactly 1 of the A_i^j's. This implies that:

$$\sum_{i=1}^{n-j} \left|A_i^j\right| \leq \sum_{i=1}^{n} |A_i| = \mathsf{L_{OPT}} \leq \mathsf{OPT} \tag{5.6}$$

Without loss of generality, let us assume that after j iterations, A_1^j and A_2^j are the two smallest cardinality sets left.

$$\left|A_1^j \cup A_2^j\right| \leq \left|A_1^j\right| + \left|A_2^j\right| \leq \frac{2}{n-j} \sum_{i=1}^{n-j} \left|A_i^j\right| \tag{5.7}$$

If the greedy algorithm uses the SmallestInput heuristic, then in the $(j+1)^{th}$ iteration, sets A_1^j, A_2^j will be chosen to be merged. In case of the SmallestOutput heuristic, we choose the two sets that give the smallest output set. Let C_{j+1} be the output set created in the $(j+1)^{th}$ iteration. Combining the above we can say that:

$$C_{j+1} \leq \left|A_1^j \cup A_2^j\right| \leq \frac{2}{n-j} \cdot \mathsf{OPT} \tag{5.8}$$

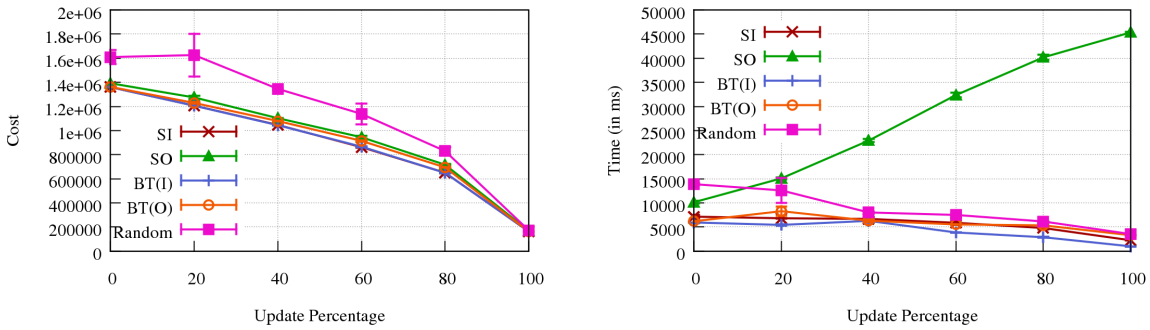
Thus, for either of the greedy strategies, SmallestInput and SmallestOutput, the total cost is:

$$\begin{aligned} \mathsf{Cost} &\leq \sum_{i=1}^{n} |A_i| + \sum_{j=1}^{n-1} |C_j| \leq \mathsf{OPT} + \sum_{j=1}^{n-1} \frac{2}{n-j+1} \cdot \mathsf{OPT} \\ &\leq (2H_n + 1) \cdot \mathsf{OPT} \quad [H_n \text{ is the } n^{th} \text{ harmonic number}] \end{aligned} \tag{5.9}$$

Lemma 5.6 *The greedy analysis is tight with respect to the lower bound for optimal ($\mathsf{L_{OPT}}$).*

Proof: We show an example where the ratio of the cost of merge obtained by using SmallestInput or SmallestOutput heuristic and $\mathsf{L_{OPT}}$ is $\log n$. Consider n initial sets where n is a power of 2. The sets are $A_1 = \{1\}, \ldots, A_i = \{i\}, \ldots, A_n = \{n\}$, i.e., each set is of size 1 and they are disjoint. The lower bound we used for the greedy analysis is $\mathsf{L_{OPT}} = \sum_{i=1}^{n} |A_i| = n$. Both the heuristics, SmallestInput and SmallestOutput, creates a complete binary tree of height $\log n$. Since the initial sets are disjoint, the collection of sets in each level is also disjoint and the total size of the sets in each level is n. Thus the total merge cost is $n \cdot \log n = \log n \cdot \mathsf{L_{OPT}}$.

Remark: Lemma 5.6 gives a lower bound with respect to $\mathsf{L_{OPT}}$, and not OPT. It suggests that the approximation ratio cannot be improved unless the lower bound ($\mathsf{L_{OPT}}$) is refined.



(a) (b)

Figure 5.6: Cost and time for compacting sstables generated by varying update percentage with latest distribution.

Finding a bad example with respect to OPT is an open problem.

LargestMatch Heuristic: In each iteration, this approach chooses those two sets that have largest intersection [139]. However, the worst case performance bound for this heuristic can be arbitrarily bad. It can be shown that the approximation bound for this algorithm is $\Omega(n)$. Consider a collection of n sets, where set $A_i = \{1, 2, \ldots, 2^{i-1}\}$, for all $i \in [n]$. The optimal way of merging is left-to-right merge. The cost of this merge is $1 + 2 \cdot (2 + 4 + \ldots 2^{n-1}) = 2^{n+1} - 3$. However, the LargestMatch heuristic will always choose $\{1, 2, \ldots, 2^{n-1}\}$ as one of the sets in each iteration as it has largest intersection with any other set. Thus the cost will be $2^{n-1} \cdot (n-1)$. This shows a gap of $\Omega(n)$ between the optimal cost and LargestMatch heuristic.

5.4 SIMULATION RESULTS

In this section, we evaluate the greedy strategies from Section 5.3. Our experiments answer the following questions:

- Which compaction strategy should be used in practice, given real-life workloads?
- How close is a given compaction strategy to optimal?
- How effective is the cost function in modeling running time for compaction?

5.4.1 Setup

Dataset: We generated the dataset from an industry benchmark called YCSB (Yahoo Cloud Servicing Benchmark) [61]. YCSB generates CRUD (create, read, update, delete) operations for benchmarking a key-value store emulating a real-life workload. YCSB parameters are

explained next. YCSB works in two distinct phases: 1) load: inserts keys to an empty database. The *recordcount* parameter controls the number of inserted keys. 2) run: generates CRUD operations on the loaded database. The *operationcount* parameter controls the number of operations.

We consider insert and update operations only to load memtables (and thus, sstables). In practice, deletes are handled as updates. A *tombstone* flag is appended in the memtable which signifies the key should be removed from sstables during compaction. Reads do not modify sstables. Thus, we ignore both of them in our simulation.

In YCSB, update operations access keys using one of the three realistic distributions: 1) Uniform: All the inserted keys are uniformly accessed, 2) Zipfian: Some keys are more popular than others (power-law), and 3) Latest: Recently inserted keys are more popular (power-law).

Cluster: We ran our experiments in the Illinois Cloud Computing Testbed [140] which is part of the Open Cirrus project [141]. We used a single machine with 2 quad core CPUs, 16 GB of physical memory and 2 TB of disk capacity. The operating system running is CentOS 5.9.

Simulator: Our simulator works in two distinct phases. In the first phase, we create sstables. YCSB's load and run phases generate operations which are first inserted into a fixed size (number of keys) memtable. When the memtable is full, it is flushed as an sstable and a new empty memtable is created for subsequent writes. As a memtable may contain duplicate keys, sstables may be smaller and vary in size.

In the second phase, we merge the generated sstables using some of the compaction strategies proposed in Section 5.3. By default, the number of sstables we merge in an iteration, k is set to 2. We measure the cost and time at the end of compaction for comparison. The cost represents $cost_{actual}$ defined in Section 5.2. The running time measures both the strategy overhead and the actual merge time.

We evaluate the following 5 compaction strategies:

1. SmallestInput (SI): We choose k smallest cardinality sstables in each iteration using a priority queue. This implementation works in $O(\log n)$ time per iteration.

2. SmallestOutput (SO): We choose k sstables whose union has the smallest cardinality. Calculating the cardinality of an output sstable without actually merging the input sstables is non-trivial. We estimate cardinality of the output sstable using Hyperloglog [142] (HLL). We compute the HLL estimate for all $\binom{n}{k}$ combinations of sstables in the first iteration. At the end of the iteration, k sstables are removed and a new sstable is added. In the next iteration, we have to compute the estimates for $\binom{n-k+1}{k}$ combinations. We

can reduce this number by making the following two observations: 1) some of the estimates from the last iteration (involving sstables not removed) can be reused and 2) new estimate is required for only those combinations which involve the new sstable. Thus, the total number of combinations for which we need to estimate cardinality is $\binom{n-k}{k-1}$. The per-iteration overhead for this strategy is high.

3. BalanceTree with SmallestInput at each level ($BT(I)$): This strategy merges sstables in a single level together. Since all sstables at a single level can be simultaneously merged, we use threads to parallelly initiate multiple merge operations. BalanceTree does not specify a order for choosing sstables to merge in a single level. We use SmallestInput strategy and pick sstables in the increasing order of their cardinality.

4. BalanceTree with SmallestOutput at each level ($BT(O)$): This is similar to $BT(I)$ strategy except we use SmallestOutput for finding sstables to merge together at each level. Even though, the SO strategy has a large per-iteration strategy overhead, the overhead for this strategy is amortized over multiple iterations that happen in a single level.

5. Random: As a strawman to compare against, we implemented a random strategy that picks random k sstables to merge (at each iteration). This represents the case when there is no compaction strategy. It will thus provide a baseline to compare with.

5.4.2 Strategy Comparison

We compare the compaction heuristics from Section 5.3 using real-life (YCSB) workloads. We fixed the operationcount at 100K, recordcount at 1000 and memtable size at 1000. We varied the workload along a spectrum from insert heavy (insert proportion 100% and update proportion 0%) to update heavy (update proportion 100% and insert proportion 0%). We ran experiments with all 3 key access distributions in YCSB.

With 0% updates, the workload only comprises of new keys. With 100% updates, all the keys inserted in the load phase will be repeatedly updated implying a larger intersection among sstables. When keys are generated with a power-law distribution (zipfian or latest) the intersections increase as there will be a few popular keys updated frequently. We present result for latest distribution only. The observations are similar for zipfian and uniform and thus, excluded.

Figures 5.6 plots the average and the standard deviation for cost and time for latest distribution from 3 independent runs of the experiment. We observe that SI and $BT(I)$ have a compaction cost that is marginally lower than $BT(O)$ (for latest distribution) and SO.

Compaction using $BT(I)$ finishes faster compared to SI because of its parallel implementation. RANDOM is the worst strategy. Thus, $BT(I)$ is the best choice to implement in practice. As updates increase, the cost of compaction decreases for all strategies. With a fixed operationcount, larger intersection among sstables implies fewer unique keys, which in turn implies fewer disk writes.

RANDOM is much worse than our heuristics at small update percentage. This can be attributed to the balanced nature of the merge trees. Since sstables are flushed to disk when the memtable reaches a size threshold, the sizes of the actual sstable have a small deviation. Merging two sstables (S_1 and S_2) of similar size with small intersection (small update percentage) creates another sstable (S_3) of roughly double the size at the next level. Both SI and SO choose S_3 for merge only after all the sstables in the previous level have been merged. Thus, their merged trees are balanced and their costs are similar. On the contrary, RANDOM might select S_3 earlier and thus, have a higher cost.

As the intersections among sstables increase (with increasing update percentage), the size of sstables in the next level (after a merge) is close to the previous level. At this point, it is immaterial which sstables are chosen at each iteration. Irrespective of the merge tree, the cost of a merge is constant [2]. Thus, RANDOM performs as well as the other strategies when the update percentage is high.

The cost of SO and $BT(O)$ is sensitive to the error in cardinality estimation. The generated merge schedule differs from the one generated by the naive sstable merging scheme which accurately identifies the smallest union. This results in slightly higher overall cost. The running time of SO increases linearly as updates increase because of cardinality estimation overhead.

5.4.3 Comparison with Optimal

In this experiment, we wish to evaluate how close $BT(I)$, our best strategy, is to optimal. Extensively searching all permutations of merge schedules for finding the optimal cost for large number and size of sstable is prohibitive and exponentially expensive. Instead, we calculate the sum of sstable sizes, our known lower bound for optimal cost from Section 5.3.1. We vary the memtable size from 10 to 10K and fix the number of sstables to 100. The recordcount for load stage is 1000 and update insert ratio is set to 60:40. The number of operations (operationcount) for YCSB is calculated as: memtable size(10 to 10K) $\times$ number of sstables (100) $-$ recordcount (1000). We ran experiments for all three key access

[2] If sstable size is s, number of sstables to merge in an iteration is 2 and the number of sstables is n, then $cost_{actual}$ would be $3 \cdot (n-1) \cdot s$.

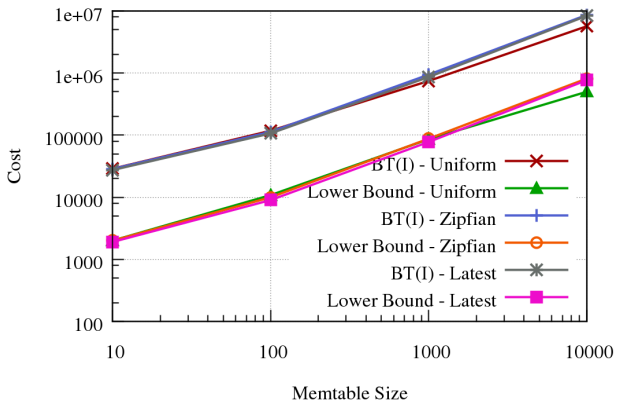
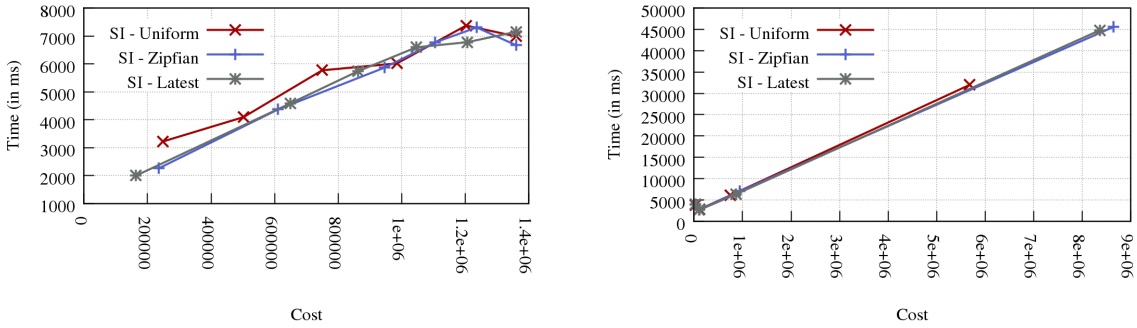


Figure 5.7: Comparing cost of $BT(I)$ to optimal which is lower bounded by sum of sizes of all sstables. Both x and y-axis are in log scale.



(a) Increasing update percentage **(b)** Increasing operationcount

Figure 5.8: Effect of cost function on completion time for compaction. *SI* strategy used. Update Percentage and datasize varied for the plots respectively.

distributions.

Figure 5.7 compares the cost of merge using $BT(I)$ with the lower-bounded optimal cost, averaged over 3 independent runs of the experiment. Both x and y-axis use log scale. As the maximal memtable size (before flush) increases exponentially, both the curves show a linear increase in log scale with similar slope. Thus, in real life workloads, the cost of our strategy is within a constant factor of the lower bound of the optimal cost. This is a better performance than the analyzed worst case $O(\log n)$ bound (Lemma 5.6).

5.4.4 Cost Function Effectiveness

In Section 5.2 we defined $cost_{actual}$ to model amount of data to be read from and written to disk. This cost also determines the running time for compaction. The goal of this experiment is to validate how the defined cost function affects the compaction time. In this experiment, we compare the cost and time for SI. We chose this strategy because of its low overhead and single-threaded implementation. We ran our experiments with the same

settings as described in Section 5.4.2 and Section 5.4.3. Cost and time values are calculated by averaging the observed values of 3 independent runs of the experiment.

Figure 5.8 plots the cost on x-axis and time on y-axis. As we increase update proportion ((Figure 5.8b) and operationcount (Figure 5.8a), we see an almost linear increase for time as cost increases for all 3 distributions. This validates the cost function in our problem formulation, as by minimizing it, we will reduce the running time as well.

5.5 CONCLUSION

In this work, we formulated compaction in key value stores as an optimization problem. We proved it to be NP-hard. We proposed 3 heuristics and showed them to be $O(\log n)$ approximations. We implemented and evaluated the proposed heuristics using real-life workloads. We found that a balanced tree based approach $BT(I)$ provides the best tradeoff in terms of cost and time.

Many interesting theoretical questions still remain. The $O(\log n)$ approximation bound shown for the SmallestInput and SmallestOutput 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.

Chapter 6: Joint Network and Cache Management for Tiered Architectures

In this chapter, we introduce a tiered version of a popular distributed file system, HDFS. It maintains symbolic links to actual data residing in a data lake like Amazon S3. On demand, it loads data from the backend. In this work, we discuss techniques for prefetching and caching data in this setting such that jobs accessing the data can meet their SLOs. We motivate the need for such a scheme in Section 6.1. Using production traces, we draw insights which helps us design NetCachier. We summarize our observations in Section 6.2. We present NetCachier system architecture in Section 6.3. We list all the implementation details in Section 6.4. Finally, we present our evaluation results in Section 6.5.

6.1 INTRODUCTION

With the explosive growth of data over the past decade [143], we are seeing an increasing adoption of *tiered architectures* [144, 145, 146, 147]. Such architectures typically consist of two tiers – a storage tier built to store exabytes of data, and a compute tier with terabytes of fast, local storage. This architecture, popularized by public cloud infrastructures (e.g., Amazon EC2 [148] and Amazon S3 [25]), is also increasingly being adopted in enterprise settings (e.g., InfoScale [144]).

In such architectures, data is regularly ingested to the storage tier and data analytics jobs are run on the compute tier, resulting in data being transferred over the network connecting the two tiers. To minimize cost, the inter-tier network is provisioned with limited capacity and generally provides best-effort service. As a result, it often ends up being a contended resource with significant variability in the bandwidth achieved by individual transfers. Recent studies have measured "almost chaotic" I/O performance between compute instances and cloud storage, observing relative standard deviation up to 94% on I/O bound benchmarks [27] [1]. In contrast, providers define storage SLOs for VMs by guaranteeing IOPS and throughput [149, 150, 151].

This network induced variation might result in delayed data transfers, and consequently, job execution times may become unpredictable. Missing job deadlines has a direct impact on business productivity. It has been recently reported that up to 25% of users' escalations in Microsoft's internal big-data clusters are due to unpredictable execution [152].

Existing systems [152, 153, 154] address intra-cluster variability by improving isolation

[1] Also called the coefficient of variation, relative standard deviation (RSD) is the standard deviation divided by the mean. Small RSD values (roughly less than 5%) correlate with a predictable I/O rate in this benchmark.

among tenants. However, applying these techniques in tiered settings would assume either that jobs will not become contrained on bandwidth to the external store, or that application deadlines can be met by prioritizing flows. By translating job specifications into network requirements, we can explictly provision this critical resource.

Motivated by these observations, we improve upon the state of the art by building a substrate for providing predictable data access to analytics workloads in tiered architectures. In building this substrate, some practical considerations are (1) minimize over-provisioning of resources, (2) minimize peak network utilization[2], and (3) maximize the number of jobs that can be run in the cluster. Collectively, these three factors allow cluster operators to maximize ROI (return on investment) without compromising on user experience.

To realize this goal, we tackle the problem of jointly optimizing the allocation of network bandwidth and local storage resources across multiple jobs. The system we build, NetCachier, allows future reservations of these two resources, which enables meeting performance expectations, or Service Level Objectives (SLOs) of job owners. As a concrete scenario for an SLO, we focus here on meeting job deadlines. Deadline SLOs are particularly important, as a large portion of cluster workload in enterprises comprises of production jobs with strict completion time requirements [152, 155].

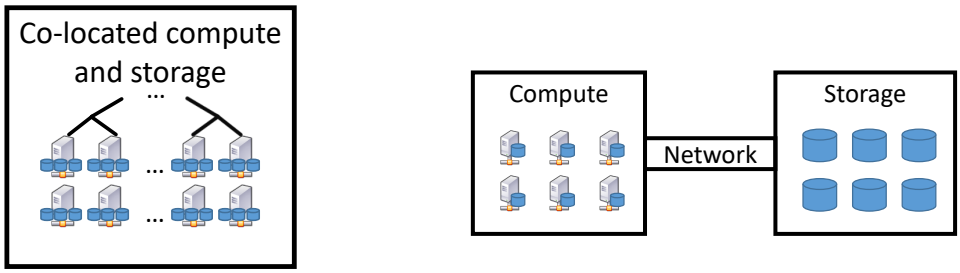
We motivate our approach by making two key observations about analytics workloads. First, jobs with completion SLOs have sufficient slack between the time the data is available to the time the job is submitted. We utilize this slack to efficiently schedule and complete the data transfer before the job starts. By caching data in the compute tier and providing it to jobs, the inter-tier network induced variability can be eliminated. Second, by suitably allocating network bandwidth, the data can be transferred across tiers at the rate at which it will be consumed. Notably, in either case, we need to plan and schedule the network transfers.

The resource allocation problem tackled in this chapter can be cast as a combined network and cache planning problem. As such, the algorithmic problem is a cross of a couple of fundamental problems. The first one is scheduling transfers over a shared communication channel. A special case of this problem is interval scheduling on multiple machines [156, 157], in which "jobs" (files in our case) need to be scheduled in a pre-specified window. The second problem is caching *shared* data in the local store. Data sharing is fairly common with data analytics workloads since the output of one job is input to multiple other jobs (see Figure 6.2a). Efficiently managing a shared cache is related to the densest-k-subgraph

[2]While other metrics related to network utilization may be considered, our main concern is peak utilization. Peak utilization is a good indicator for network congestion in the short term, and guides provisioning decisions in the longer term.

problem [158, 159], where k is the cache size, vertices correspond to files, and edges to jobs. Each of the latter special cases has been studied in depth in the scheduling literature, and is relatively hard to approximate. Nevertheless, we draw important insights from existing solutions that help us design efficient heuristics for the problem as a whole.

To ground these ideas in a practical setting, we implement NetCachier by extending Apache Hadoop/HDFS [8]. Conceptually, our implementation comprises of two key components (§4.4.1). First, as noted above, the cluster workload consists of jobs whose submission time as well as completion time requirements are known a-priori. This information is fed to a *planner* which allocates network bandwidth and schedules network transfers appropriately. Second, in a tiered architecture, there is storage available on the nodes of the compute tier. We treat this storage as a *shared cache*, and use it to temporarily hold the data until jobs consume them.



(a) Big data clusters today. **(b)** Tiered architecture.

Figure 6.1: Different architectures for data analytics clusters.

Building such a tiered storage substrate required substantial efforts. Though tiered architectures are becoming popular and are gradually being deployed, the underlying storage systems are all single-tier systems. Consequently, at job submission time, jobs need to specify the tier from which data should be accessed. This makes it extremely cumbersome for users to run jobs predictably in a tiered setting. To address this limitation, NetCachier implementation includes modifying HDFS to be tier-aware. A high-level system architecture of NetCachier is shown in Figure 6.3. As shown in the figure, an HDFS instance runs in the compute tier (and caches data using local storage) and extensions built for NetCachier allow data to be seamlessly migrated between tiers. Additionally, for files stored in a remote storage tier, our extensions allow HDFS to retrieve data on-demand in an application transparent manner (see §6.4). For brevity, we omit many engineering details here, yet we intend to contribute our modifications to the Apache Hadoop code base.

We deploy our implementation of NetCachier on a 280 node cluster and show that NetCachier reduces the peak network utilization by up to 8X compared to techniques that perform caching alone, or read all data directly from the storage cluster while meeting the deadlines

(a) Number of jobs reading a file shown as a CDF.

(b) Average File processing rate bucketed by number of jobs that read a file.

(c) CDF (over files) of *prefetch slackness*: Time from creation to first access of a file, divided by the time to transfer the file

Figure 6.2: Workload characteristics.

of all jobs. This shows that using NetCachier allows us to run analytics workloads with significantly lower network bandwidth between the storage and compute tiers. Further, using trace-driven simulations, we show that we can improve the median runtime of ad hoc jobs by up to 60% with NetCachier.

In summary, our main contributions are:

- We formulate the problem of jointly scheduling network and local storage resources to meet job SLOs in tiered architectures. We design heuristics to solve the problem; the heuristics are designed based on principles drawn from related algorithmic problems.
- We propose a new architecture to enforce a long-term plan for network and storage resources. We fully implement this architecture on top of HDFS (§6.4).
- Using deployments on large-scale clusters and simulations based on production workloads, we show that NetCachier makes judicious use of network and storage resources and improves the number of jobs meeting their SLOs with significant reductions in the bandwidth required to do so (§6.5).

While the context of this chapter is enterprise tiered store architectures, both the algorithmic solutions and their infrastructure support can be extended to the public cloud setting, where bandwidth between the compute and storage tiers can be significantly limited. Such an extension would have to address various issues that arise in the cloud — for example, per-VM and per storage account throttling.

6.2 MOTIVATION

Here we provide a brief background on tiered architectures, and describe results of our analysis of production traces which motivate the design of NetCachier.

6.2.1 Tiered architectures

In the past decade, datacenter scale clusters built around commodity servers with colocated storage and compute resources have become commonplace (see Figure 6.1a). One of the key lessons learned with data analytics workloads is that there is a growing gap between the volume of data collected (and stored/archived for regulatory purposes) versus the size of the working set. For instance, it has been reported that the size of the working set as compared to the total data stored in the cluster can be as low as 6% [160]. As the data volumes continue to grow, the classic big data architectures [3] are a poor match for this workload, as 100s or 1000s of commodity servers will be an expensive store for mostly cold data.

To address this problem, most enterprises and cloud providers are increasingly relying on *tiered* architectures, to lower costs (Figure 6.1b). Data is ingested into clusters optimized for storage (e.g., InfoScale [144]), and applications are run on a separate compute tier. Running big data analytics on such architectures to meet SLOs is hard as the network bandwidth between the two tiers is often limited and becomes a contended resource. However, our analysis in §6.2.2 shows that typical big data workloads exhibit various characteristics which make them amenable to tiered architectures.

6.2.2 Insights from production workloads

We analyze the characteristics of typical big data workloads, and provide insights into how such workloads can be run in tiered architectures. This analysis motivates the design of NetCachier. For this purpose, we use job execution and filesystem logs from a large production cluster consisting of several thousands of machines running data analytics workloads. These logs were collected over a week and comprise details from millions of jobs executed, and tens of millions of files read. In addition, we report on relevant recent findings from enterprise big-data clusters.

Job characteristics are becoming predictable: Recent studies of big-data analytics clusters indicate that an operator can infer information about important job characteristics (see [155, 152, 161] and references therein). In particular, by processing existing job execution telemetry, an operator can deduce the submission time and deadlines of periodic production jobs (even if those are not explicitly specified by the job owner), job durations, the identity and size of files read by the job, etc. This facilitates resource planning for a large portion of enterprise jobs [152].

Caching files in the compute tier can lead to substantial network bandwidth savings: Figure 6.2a shows the number of jobs accessing a single file as a cumulative distribution function (CDF). We see that nearly 15% of files are accessed by more than 10 jobs,

and nearly 5% files are accessed by more than 100 jobs. These shared files account for more than 90% of the jobs' input. Similar observations have been made earlier [28]. Since caching input data in local storage requires transferring the data only once, the above statistics indicate that caching shared files would lead to significant reduction in the amount of data transferred across the network.

Standard caching algorithms might not suffice: As shown in Figure 6.2b, we see a low correlation between the number of jobs that read a file and the rate at which it is read – files that are accessed fewer times can still be read by jobs at a high rate. Caching or prefetching files that are read at higher rates reduces the network bandwidth consumed when jobs reading these files are run. On the other hand, files with low read rate can be read remotely with low network overhead. Unfortunately, standard caching algorithms such as LRU-k [162] account primarily for access count (or frequency) considerations, and are oblivious to network bandwidth consequences.

Data can be prefetched before job execution: The time from when data is created or uploaded to the cluster, to when it is processed can vary between a few minutes to several hours. In Figure 6.2c, we show the "prefetch slackness" of files, which is defined as the ratio between the time available for transferring the file (i.e., the time from its creation to the first access to the file), and the time to transfer the file (for a given transfer rate). For the time to transfer a file, we show lines corresponding to allocating 1Gbps and 5Gbps to the file. Prefetch-slackness greater than one means that the file can be safely prefetched before job execution. The figures show that the bulk of the files can indeed be prefetched on time. Further, a sizeable portion of the files has substantial slack, which can be exploited by intelligent resource management (e.g., schedule a prefetch of a large file at off-peak times).

6.2.3 Summary

The primary takeaways from our analysis are as follow:

- Job and file characteristics are accessible to operators, and can by used for network and storage resource planning.
- File prefetching can be used for substantial reduction in bandwidth consumption.
- Resource management should reason jointly about network and cache. This requires new algorithms which schedule prefetches, remote reads and cache operations, while accounting for bandwidth and storage capacity limits.

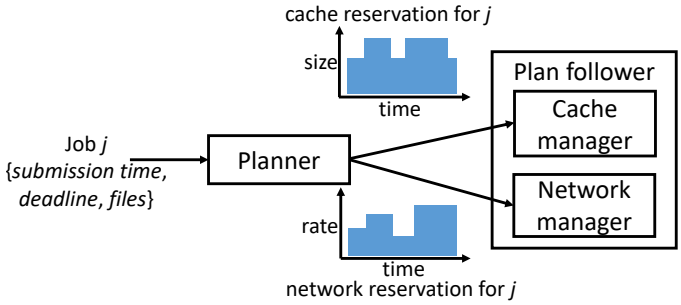


Figure 6.3: NetCachier **architecture**

6.3 SYSTEM ARCHITECTURE

In this section, we provide an overview of the architecture of NetCachier. NetCachier consists of two main components — a *planner*, and a *plan follower* (Figure 6.3). The planner considers jobs that arrive in the next *planning window*, and computes an *execution plan* for the jobs that need to meet their completion time deadlines. The plan follower is responsible for the enforcement of the plan during runtime. The planner runs at the start of every planning window, which can be a configurable parameter (e.g., every few hours) — this ensures that if any new jobs with deadlines are submitted, they can be planned for. It can also be invoked as required (e.g., changes in cluster conditions etc.). We next describe at a high level the setting under which the planner operates, and how its plan is used to satisfy job SLOs. A detailed description of the our solution thereof can be found in §6.3.1. Implementation details are deferred to §6.4.

Planner: The planner considers various job characteristics to determine the execution plan — these include job submission time, deadline, the set of files each job has to read, and the maximum rate at which the job is expected to consume the input data. These characteristics can be determined from previous executions of the job (see, e.g., [152] and references therein). Given a set of jobs to be scheduled, the planner takes the current cache and network state into account and schedules the job to complete by its deadline, while attempting to minimize the network bandwidth used. If the deadline of a job cannot be satisfied given the current cluster state, the job is rejected. Rejected job can be executed as best effort jobs, or submitted at a later time.

Execution plan: The planner outputs an execution plan. The execution plan maintains information how input files will be consumed by each accepted job. Specifically, each input file of a job is classified as either (i) prefetched before the job starts execution, (ii) read from the cache (i.e., resides in the cache before the job execution; for example, the input file was prefetched for a different job that executes earlier), or (iii) read directly from the remote store, in which case it is specified whether the file should be cached for future use. For the

files that are either prefetched or read from remote store, the plan also specifies the rate of the transfer.

The execution plan is enforced using appropriate reservations on the cache and the network. For example, consider a file of size s bytes that is prefetched at rate r bps from time t_1 to t_2, and read by jobs with a latest deadline d. That file requires a cache reservation of s bytes during the interval $[t_1, d]$, and a network bandwidth reservation of r bps during the interval $[t_1, t_2]$. As part of the cache reservation, NetCachier ensures that a file that needs to be read from the cache is indeed *pinned* in the cache, i.e., it is not evicted until jobs finish reading it.

Plan follower: The plan follower executes the plan that is determined by the planner. This involves (a) scheduling prefetches for files at the appropriate time and rate, which also requires reserving the required cache space, (b) enforce network reservations to ensure that files are transferred at the required rate, (c) ensure that files are pinned in cache while they are consumed. Accordingly, the plan follower uses a *cache manager* and a *network manager* to enforce the reservations of the respective resources.

In building NetCachier, we make two assumptions. First, we assume that compute resources can be scheduled using existing techniques and it is not a contended resource. We discuss implications of this assumption in §6.6. Second, we only focus on scheduling network transfers for the input data of jobs. The reverse direction, namely, writing back the output produced by jobs to the external store for archival purposes is assumed to be accommodated on a best-effort basis and hence, is not discussed further.

6.3.1 Planning algorithm

The planning algorithm (Algorithm 6.1) maintains the state of the local cache ($\mathcal{C}$) and the state of the network ($\mathcal{N}$). The cache state consists of files that are expected to be present in the cache at any point in time. The network state keeps track of files that are being transferred over the network over time and the rate of the transfers. When the algorithm terminates, these states are used to determine the execution plan to be enforced by the plan follower (§4.4.1).

The algorithm works in two steps. First, it determines which files must be prefetched (Line 2 in Algorithm 6.1). The goal of this step is to prefetch a subset of files before the jobs requiring the files start execution.

Scheduling prefetches: Algorithm 6.2 shows the pseudocode for determining which files can be prefetched. It determines the start time for each prefetch as well as the corresponding rate of the transfer.

```
   input: Cache state C, Network state N, jobs 1,...,N, files 1,...,L
1  (C, N, P) ← find_files_to_prefetch(C, N, L)
2          → P: the prefetch plan includes files to be prefetched, prefetch start times and
   the rates; updates cache and network state.
3  for j ∈ {1,...,N} do
4      if fit(j, C, N) then
5          Accept j, and update cache state by pinning required data in cache, and
            network state by scheduling required transfers.
6      end
7  end
```

Algorithm 6.1: Algorithm used by the planner.

We handle files in a certain order which takes into account both their size and the extent to which they are shared by multiple jobs. As a concrete metric for the latter, we use the maximum aggregate bandwidth that will be needed by jobs if the file is read remotely (this accounts for simultaneous reads of the same file). Specifically, we sort the files based on the ratio between their maximum aggregate bandwidth and their size.

We handle files based on this sorted order. For each file we first determine whether it can be fetched, and if so what is the best timing for fetching. In detail, scheduling a prefetch for a file involves determining its (a) start time, (b) completion time, and (c) bandwidth allocation. For simplicity, we set the completion time of the prefetch to be the start time of the first job that requires the file, and ensure that the file is fetched at a constant rate. Thus, determining the start time completely defines a prefetch. For each file, we consider a set of candidate prefetch start times which are determined based on two key observations. First, for a prefetch operation to succeed, sufficient cache space must be available to accommodate the file. Second, when the cache is at high utilization, other files can be evicted from the cache when the job that accesses them finishes its execution. Thus, we only need to check the cache state at the times where jobs that read cached files complete, since these are the only times where evictions can happen.

Finally, the algorithm explores whether a prefetch can be scheduled at each of the candidate times and computes the corresponding peak bandwidth. If the peak bandwidth achieved with a prefetch operation is less than the one produced by a remote read (Line 14), the prefetch operation is scheduled.

Scheduling jobs: Once the set of files to prefetch are determined, the planning algorithm determines if each job j can be admitted into the cluster in order to meet its deadline (Lines 4–9 in Algorithm 6.1). For files that are read by j and are not prefetched before j starts, the algorithm tries to schedule reads from the remote store (in $fit()$) during the job's execution.

```
   input: Cache state C, Network state N, files 1, ..., L
 1 Calculate r_ℓ ← the aggregate rate at which file ℓ is accessed
 2 Sort files in descending order of r_ℓ/size(ℓ)
 3 Calculate s_ℓ ← time when file ℓ is required
 4 prefetches P = ∅
 5 for ℓ ∈ {1, ..., L} do
 6     remote ← peak bandwidth when reading ℓ remotely
 7     for t ∈ {candidate prefetch start times} do
 8         if there is enough space from t to s_ℓ then
 9             prefetch_t ← peak bandwidth when
10             prefetching ℓ at [t, s_ℓ]
11         end
12     end
13     if min_t(prefetch_t) < remote then
14         P ← P ∪ f
15         update N and C with ℓ being prefetched at t.
16     end
17 end
```

Algorithm 6.2: Algorithm used to prefetch files.

If these reads do not violate the network constraints, the job is deemed admissible. If the job is not admissible we free up network and cache capacity corresponding to prefetched files that are not shared by other jobs. We repeat the algorithm a few times, to see if additional jobs can be admitted as a consequence of freeing up capacity.

6.4 IMPLEMENTATION

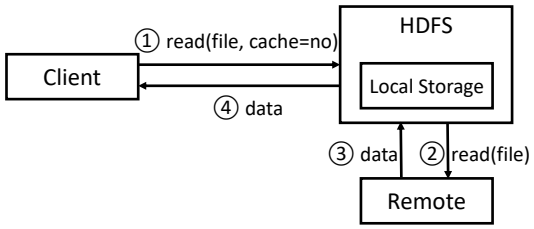


Figure 6.4: Remote read workflow with HDFS

We implemented NetCachier in Apache Hadoop/HDFS [3], a widely used data analytics platform. When HDFS is used to manage the local store in tiered architectures, users must manually schedule data copies between tiers [163, 164]. This is fragile for multiple reasons such as, input data is unavailable until the copy completes, users need to explicitly coordinate sharing of cached data, and concurrent transfers are uncoordinated.

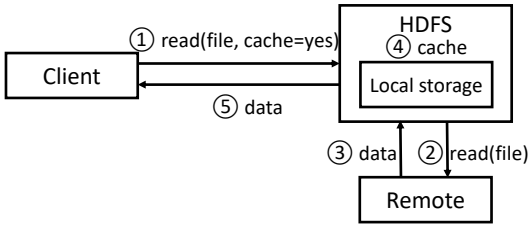


Figure 6.5: `DemandPaging` **workflow with HDFS**

To solve this issue and enable the use of local storage as a cache for the remote store in NetCachier, we modified HDFS (§ 6.4.2). to ensure that the plan generated by NetCachier is followed during job execution. Our implementation also involves a standalone planner component (§ 6.4.3).

6.4.1 HDFS overview

HDFS exports a hierarchical namespace through a central *NameNode*. Files are managed as a sequence of blocks. Each block can have multiple replicas stored on a cluster of *DataNode* servers. Clients read HDFS files by requesting block locations from the NameNode and reading block data from DataNodes.

Each DataNode generates periodic reports to the NameNode, including a set of attached *storage* devices with an associated type. The type identifies each storage as a disk, SSD, or RAMdisk mount. For all files in the namespace, the NameNode records a *replication factor* specifying the target number of replicas of that block, and a *storage policy* that defines the media (i.e., storage type) in which each replica should be stored.

6.4.2 Modifications to HDFS

Implementing NetCachier in HDFS required two major extensions. First, we introduce a new `EXTERNAL` storage type corresponding to the external store. Second, we add a throttling parameter to replication requests, to set the rate at which the NameNode creates each new block replica. Reconciling this abstraction with other invariants HDFS maintains for physical storage required substantial code modifications. The following elides engineering details to focus on conceptual implementation changes.

The contents of the external filesystem appear as a subtree mounted in the NameNode, as shown in Figure 6.6. We create this mount by synthetically mirroring external files in the HDFS namespace, with at least one replica in `EXTERNAL` storage. The NameNode considers all `EXTERNAL` replicas reachable when at least one DataNode reports attached `EXTERNAL`

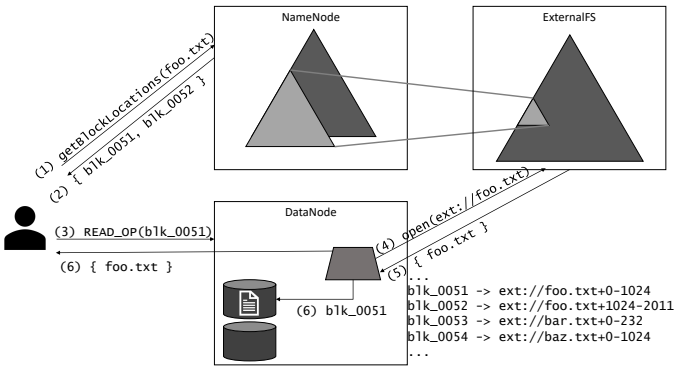


Figure 6.6: NetCachier **Implementation**

storage.

All requests to the external store pass through HDFS. An example read sequence proceeds as follows, and as illustrated in Figure 6.6.

1. Request block metadata from HDFS.
2. Receive list of blocks, including location and storage type metadata (not shown). When a client requests locations for a replica stored in `EXTERNAL` storage, the NameNode selects the closest DataNode[3].
3. Request block data from target DataNode.
4. Request file data from external store. The mapping to the external namespace is flexible, and need have no relationship to its location in the HDFS namespace.
5. Return file data to the Datanode.
6. Stream data back to the client, storing a cached copy in local storage.

When replication is increased in the NameNode, the request may include an optional throttling parameter. Our implementation ensures that when blocks of that file are replicated, concurrent transfers will not exceed this target rate.

In combination, the `EXTERNAL` storage type and metered transfers add throttled paging between the external store and HDFS. By adjusting the storage policy and replication factor of a file, the NetCachier plan follower can schedule metered transfers from the external store into local media. Mediated by HDFS, external data read through `EXTERNAL` storage is transferred at predictable rates and transparently shared between jobs.

[3]For most jobs, the client will be running on the same machine, so NetCachier adds no unnecessary, intra-cluster transfers.

6.4.3 Planner

The planner admits jobs submitted to NetCachier. For jobs that are periodic with predictable I/O requirements, NetCachier determines an execution plan using the techniques described in §6.3.1. It then runs these jobs based on the generated plan— this includes prefetching the required files at the rate determined by the plan at the appropriate time, and submitting the jobs to the resource manager (YARN) at the specified start time. Jobs that are ad hoc are directly submitted to YARN, and run as they would run today.

The plan follower manipulates the contents of the cache by adjusting storage policies and by scheduling metered replication of replicas, as discussed in §4.4.1 and illustrated in Figure 6.3. We implement the `NetworkManager` by modifying the HDFS Mover [165][4] to schedule metered replications, following the execution plan. The planner evicts replicas from local media by lowering the replication of a file.

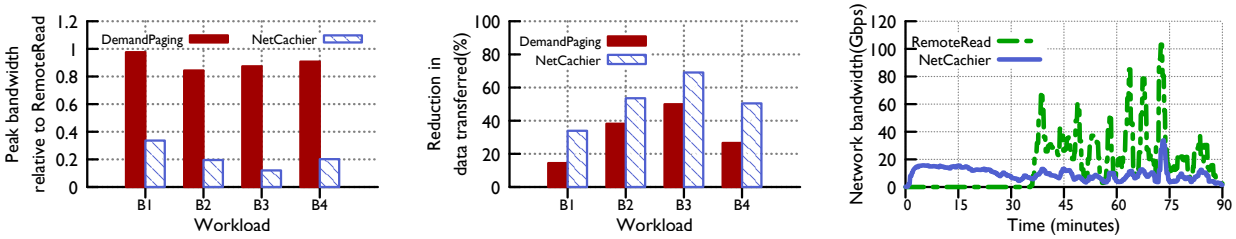
Ad hoc, unplanned workloads also benefit from caching in HDFS. The `CacheManager` implements a *readthrough* policy governing when unscheduled reads from the external store should be persisted in local media. If the replication target is scheduled but unmet, NetCachier may embed instructions in the block access token[5] for the DataNode to retain a copy of the replica in a storage as it streams data to the client (Figure 6.5). If the ad hoc workload reads the full block, then the prefetch can be cancelled, avoiding the transfer cost. If not selected by the caching policy, NetCachier simply proxies data to the client (Figure 6.4).

6.5 EVALUATION

We evaluate NetCachier using deployments on a 280 node Hadoop cluster and using large-scale simulations. Our experiments are based on workload traces from a data analytics production cluster from a large enterprise[6], running several 1000s of machines. We show that compared to various baselines, that represent state-of-the-art in tiered architectures, using NetCachier results in the following improvements.

- Reduces the peak utilization of the network between the storage and compute tier by up to 8X.
- Increases the number of jobs that meet their deadlines by 10-30%, under resource constrained scenarios.
- While current techniques can be more than 10X worse compared to a LP-based lower bound, NetCachier is at most 2.5X worse (about 4X improvement).

[4]HDFS utility for re-balancing cluster storage to reflect updated policies.
[5]Access control mechanism in HDFS verifying that the NameNode authorized the client to read the block.
[6]Company name withheld for anonymity.



(a) Peak network bandwidth used by NetCachier and DemandPaging.

(b) Reduction in the total inter-tier data transferred with NetCachier and DemandPaging.

(c) Variation in network bandwidth used over time Results for DemandPaging are similar to that of RemoteRead and hence, not shown.

Figure 6.7: Benefits of using NetCachier on SLO workloads. Improvements are shown relative to RemoteRead.

- Reduces the runtime of ad hoc jobs by 20%-68%, when ad hoc jobs are admitted along with deadline-constrained workloads.

6.5.1 Methodology

Experimental setup: We deployed our implementation of NetCachier on a 280 node cluster where machines were organized into 7 racks with 40 machines each. To emulate a tiered architecture, 6 racks were used for the compute tier and the remaining rack was used as the storage tier. We throttle the bandwidth between the two tiers as required. The storage tier runs stock HDFS and stores the input data for the workloads. The compute tier runs HDFS with our modifications (as described in §6.4), along with YARN as the resource management framework. The planner runs in the compute tier and accepts jobs submitted to NetCachier.

Workloads: Our workloads are based on traces from production clusters. We identify jobs with SLOs based on techniques similar to what were used in Morpheus [152]. We consider 4 different production workloads (B1, B2, B3 and B4) and scale them appropriately, to fit our 280 node cluster.

Metrics: We use two metrics to measure the benefits of using NetCachier under medium to high load scenarios: (i) peak network bandwidth, and (ii) number of jobs with SLOs that we can satisfy. For workloads with ad hoc jobs, we also use the runtime of the ad hoc jobs as a proxy for the effectiveness of NetCachier at allocating network and storage resources.

Baselines: We compare NetCachier against two baselines which represent how typical data analytics workloads process data in tiered architectures today [163, 164, 166].
(1) `RemoteRead`, in which all workloads process data by reading it remotely from the storage tier (see Figure 6.4).

(2) DemandPaging, in which the local storage is used as a cache for data read from the storage tier. Data is paged in on demand, and we use PACMan LIFE [28] as the policy to manage the cache space (see Figure 6.5).

Further, we evaluate the efficacy of our heuristics in NetCachier by comparing it with a lower bound obtained from a Linear Problem (LP) relaxation of the problem. The LP minimizes the peak network bandwidth used while maximizing the number of jobs that meet their deadlines. The formulation is a relaxation of the original problem, e.g., because it allows files to be partially prefetched (which is not supported in practice). We skip a detailed description for brevity.

6.5.2 Benefits from NetCachier

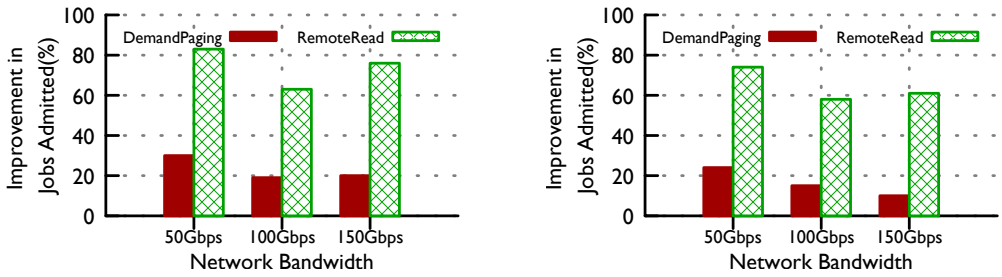
For each of the workloads described above, we generate a job trace lasting for one hour, and run them using the Gridmix workload generator [167] on our 280 node cluster setup. NetCachier generates a plan and runs them using our implementation on top of HDFS (§6.4). DemandPaging uses the same implementation to perform caching. However, RemoteRead is run by directly reading files from the storage rack.

We evaluate NetCachier in two different load regimes.

Medium load regime: Figure 6.7a shows the reduction in the peak bandwidth with DemandPaging and NetCachier, relative to RemoteRead under medium load scenarios (all jobs meet their deadlines).

Our observations are two-fold. First, while caching alone reduces the peak bandwidth, the reduction is not significant – this is because the first access to a file causes the file to be read remotely at the rate it is processed. This can result in high peak utilization. Second, we see that as NetCachier prefetches the data at a low rate before jobs start, it can reduce the peak bandwidth by upto 8X compared to RemoteRead (Figure 6.7a). This reduction is also illustrated by Figure 6.7c, which shows a timeline of the network usage in Gbps for workload B2 (solid blue line shows NetCachier, and dashed green line shows RemoteRead). NetCachier uses the first 30 minutes to prefetch data, and the jobs start running after that.

NetCachier also reduces the amount of data transferred across the network to storage. As shown in Figure 6.7b, compared to RemoteRead, NetCachier results in 70% less data transferred. While caching helps DemandPaging (upto 50% less data transferred compared to RemoteRead), it still reads more data than NetCachier. This is because when a file is read in parallel by multiple jobs, and it is not in the local cache, DemandPaging fetches the data independently for each of job. On the other hand, as NetCachier can potentially prefetch such a file, a single transfer from the remote store brings the data into cache, and jobs read it



(a) Offline scenario. **(b) Online scenario.**

Figure 6.8: Increase in percentage of jobs that meet their SLOs with NetCachier. All jobs are planned for (a) offline, and (b) online, as the jobs arrive. Workloads are based on B3.

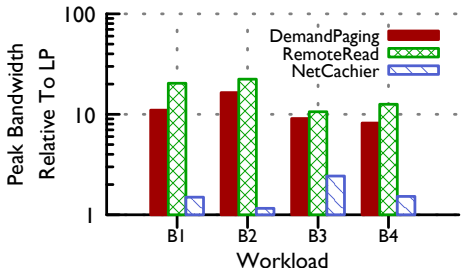


Figure 6.9: Peak network bandwidth relative to lower bound.

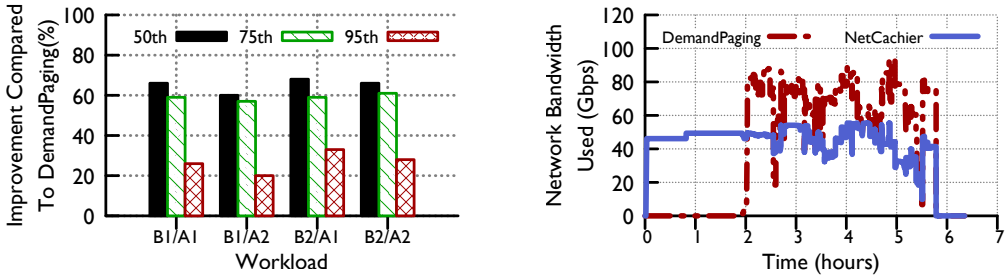
from the cache.

High load regime: Under high load when it is not possible to meet deadlines of all SLO-constrained jobs, NetCachier aims to maximize the number of jobs that meet their deadlines. Figure 6.8 shows the increase in number of jobs that meet their completion time SLOs with NetCachier compared to the different baselines. We consider two scenarios — (a) an offline scenario, where we assume all jobs are known ahead of time, and (b) an online scenario, where jobs are planned for as they are submitted. The same workload is run for different network capacities between the storage and compute racks.

While NetCachier completes lesser jobs in the online scenario compared to the offline, the differences are small. Overall, NetCachier accepts 10-30% more jobs compared to `DemandPaging`, and upto 80% more jobs compared to `RemoteRead`. This increase is a consequence of (a) the reduction in the peak network bandwidth with the use of prefetching, and (b) efficient use of the cache and network resources by planning ahead of job submissions.

6.5.3 Performance of NetCachier compared to a LP-based lower bound

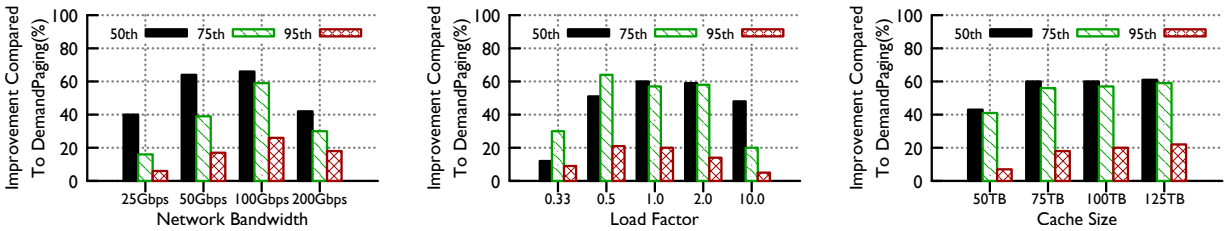
The planning algorithm in NetCachier (§6.3.1) makes design choices amenable for a practical implementation. For example, when a file is prefetched, it is read at a fixed rate without



(a) Reduction in ad hoc job runtimes for different workload combinations.

(b) Variation in network bandwidth (over time) for SLO jobs in workload B1/A1.

Figure 6.10: Results for workloads with both SLO and ad hoc jobs.



(a) Varying network bandwidth.

(b) Increasing load of ad hoc jobs.

(c) Varying available cache size.

Figure 6.11: Reduction in ad hoc job runtime percentiles for workload B1/A1 using NetCachier**.**

any variation over time. Only whole files are cached. While these choices make for a simpler and more robust implementation, it can suffer from potential sub-optimality. For example, when cache space is limited, fetching partial files might prove to be useful even though the whole file may not fit in cache.

As described in §6.5.1, we formulated an LP that aims to minimize the peak bandwidth used. It provides a lower bound for the planning algorithm in NetCachier. Here, we compare NetCachier with this lower bound using trace-driven simulations.

Figure 6.9 shows the peak bandwidth achieved by NetCachier and the different baselines relative to the LP. We consider workloads from the four different business groups. We see that in all cases NetCachier performs less than 2.5X worse than the LP. On the other hand, the `RemoteRead` and `DemandPaging` can be more than 10X worse. This shows that even though the planning algorithm in NetCachier can lead to potential sub-optimality, it is significantly better than existing baselines.

6.5.4 Benefits for ad hoc jobs

In practice, data analytics clusters run SLO jobs along with ad hoc jobs for data exploration or research purposes. While no guarantees are provided to the ad hoc jobs, users expect them to finish at the earliest. To this end, one of the goals of NetCachier is to efficiently provision resources for SLO-constrained jobs in order to free up network and cache resources for ad hoc jobs. In this section, we show that the techniques used in NetCachier can lead to significant improvements in the runtimes of ad hoc jobs.

We derive the workloads for ad hoc jobs from (a) our internal production cluster traces, and (b) traces from Facebook's data analytics clusters [168]. We will label these workloads A1 and A2, respectively. For the traces from Facebook, we randomly sample 40% of the jobs to be ad hoc jobs (this has been shown to be typical percentage of ad hoc jobs in clusters [169, 152]).

We use trace-driven simulations to run the ad hoc jobs alongside SLO jobs, which are derived from the workloads of business groups B1 and B2. The simulations reserve resources for the SLO jobs to avoid interference from the ad hoc jobs. Ad hoc jobs share the remaining resources based on max-min fairness.

Varying workloads: Figure 6.10a shows the improvement in various runtime percentiles for ad hoc jobs with NetCachier compared to using `DemandPaging`. We consider different workload combinations (Bi/Aj denotes a workload with SLO jobs drawn from workload Bi and ad hoc jobs from workload Aj). We observe upto 68% improvements in the 50^{th} percentile and upto 33% improvements in the 95^{th} percentile. The benefits reduce at higher percentiles because these jobs run longer. Thus, they are less affected by spikes in the network usage. Jobs with lower runtimes can be significantly delayed due to network usage spikes – as shown in Figure 6.10b, such network utilization peaks are significantly lower when using NetCachier compared to `DemandPaging`.

Varying cluster configurations: The benefits of NetCachier hold over a variety of cache sizes, network bandwidths and load conditions. Figures 6.11a– 6.11c show how NetCachier compares to using `DemandPaging` for workload B1/A1 under various cluster configurations.

Our observations are three-fold. First, as network bandwidth increases, the benefits from using NetCachier increase, and then decrease. At lower bandwidths, there is significant contention between the ad hoc jobs which leads to an increase in their runtimes. At higher bandwidth, there are ample resources available, and ad hoc jobs are not constrained. This leaves little room for improvement when using NetCachier's plan.

Second, based on the same intuition, we find that NetCachier results is significantly lower runtimes at moderate load conditions. Finally, at larger cache sizes, NetCachier improves the

runtimes of ad hoc jobs more as there are more opportunities to prefetch and cache the input data of SLO jobs. This in turn results in lower network utilization.

6.6 RELATED WORK

NetCachier uses various techniques that are related to the following areas of research.

Caching and prefetching: Caching and prefetching have been extensively studied in various areas of computer science, such as operating systems [170], computer architecture [171], database systems [162] and networking [172, 173, 174]. The different works span theoretical analysis of algorithms [175, 176, 177, 178], and practical implementations thereof [179, 180, 181, 162, 182, 183, 184]. Recently, caching has been considered in the context of big data analytics systems (e.g., PACMan [28], FairRide [185], EC-Cache [186]). In contrast to these works which mostly employ online reactive techniques, NetCachier plans ahead which files should be cached in the local store; additionally, NetCachier also schedules the associated data transfers while taking into account network bandwidth considerations.

Scheduling network flows: Scheduling network transfers has been widely explored in the context of both shared multi-tenanted datacenter settings [154, 187, 188, 189, 190, 191], and wide-area networks [192, 193, 194, 195, 196]. Some of these works target increasing network utilization and finish flows faster [189, 190, 191, 192, 193, 194], while others aim at providing guaranteed network bandwidth or meet deadlines for network transfers (e.g., [195, 196, 154, 187, 188]). Recent works, such as Corral [169] and a series of projects centered around the coflow abstraction [197, 198, 199, 190], aim at reducing end-to-end application runtimes, for example, by speeding up the entire shuffle phase in data analytics applications. However, these projects do not explicitly consider utilizing local storage in tiered architectures for caching input data.

Storage systems: Tiering is common in storage systems, particularly for serving web requests [117, 200] and analytics [201, 8, 202]. Further, systems like Tachyon [203] and Nectar [112] target analytic workloads and use job lineage to control tiering decisions. In the context of public clouds, CAST [204] determines on which storage tiers should input data be stored, to achieve the best job performance. These techniques are complementary to NetCachier, which solves the problem of how to transfer data from remote stores to the compute tier in order to meet job SLOs. Systems like IOFlow [153] provide guaranteed IO bandwidth across datacenter storage. However, such systems mainly focus on the mechanics of performing rate allocation once the desired transfer rates have been determined.

6.7 CONCLUSION

In this chapter, we design and implement NetCachier, a substrate for providing predictable data access to analytics workloads in tiered architectures. NetCachier jointly plans the allocation of network and storage resources in order to meet job SLOs, while minimizing network bandwidth consumption. The planning algorithms relies on the ability to (i) extract job characteristics before their execution, (ii) utilize local storage in the compute tier to cache data accessed by jobs, and (iii) control the timing of data transfers for both remote access and prefetching.

We implement NetCachier in Apache Hadoop. We intend to release the code as open source by contributing back to Apache Hadoop/HDFS code base. Using deployments on a 280 node cluster, we show that NetCachier can reduce the network capacity requirement of production workloads by up to 86%, and meet the deadlines of 20-30% more jobs compared to using network-oblivious caching techniques.

While our focus in this chapter has been on enterprise clusters, we envision using the mechanisms and techniques of NetCachier in public cloud deployments, where tiered store architectures can lead to substantial cost reductions. Adapting NetCachier to the public cloud setting introduces new challenges, such as accounting for per-VM bandwidth limits and per storage account limits.

Chapter 7: Conclusion and Future Work

In this thesis, we have discussed projects to support the central thesis that *data reconfiguration mechanisms can be done in the background by using new optimal or near-optimal algorithms coupled with performant system designs*. In all the 5 projects presented in this thesis, we built practical systems to solve real-world data reconfiguration problems. In Chapter 2 and Chapter 3, we solved table level configuration change in NoSQL databases with Morphus and Parqua respectively. In Chapter 4, we showed how Getafix can reduce memory usage in distributed interactive analytics engine. We formally analyzed the problem of compaction in Chapter 5. Finally, in NetCachier, we presented novel prefetching and caching techniques for batch systems working on top of a tiered architecture.

In all our solutions, we have used well-known theoretical abstraction to design algorithms which have been shown to be optimal or near-optimal under simplifying assumptions. In all the solutions, we have built techniques to handle reconfiguration in the background without affecting the fast path. In Getafix and NetCachier, we automatically trigger reconfiguration as workload changes, thereby significantly reducing the load on sys-admins and often reducing capital expense. Reconfiguration operations also involve significant data movement. We showed in Morphus, Getafix and NetCachier that through sophisticated network planning and bandwidth allocation, one can significantly improve the reconfiguration runtime.

Our evaluation with real-life production traces further show how effective our systems are in handling the respective problems. Even though we used real-life datasets to evaluate our systems, our systems generalize to other datasets as well. Morphus and Parqua dealt with the problem of changing table-level configuration parameters in a distributed NoSQL database. Using bipartite matching, we were able to minimize the data transfer required during reconfiguration. Both the systems perform reconfiguration with several 9s of availability. Getafix reduces total replication in frontend compute nodes for distributed interactive analytics engines. Using best fit based algorithm, Getafix can reduce memory by upto 2.15× compared to state-of-the-art Scarlett and save deployment costs to the tune 10s of millions of dollars annually. Our compaction work proposed a theoretical framework to analyze the efficacy of different compaction algorithms. Our evaluation showed that the algorithms perform constant factor of optimal in practice. In NetCachier work, we devised novel heuristics to allocate network and cache such that the number of job meeting their SLOs can be maximized. We were able to improve by job acceptance by 80% while reducing peak network utilization by 8×.

Our thesis opens up several directions for further research:

- **Changing shard keys in multi-index data stores:** Future work would involve handling reconfiguration in a database which allows shard keys to have multiple columns. In Morphus and Parqua, we assumed that only a single column from the schema is selected as a shard key. There can be use-cases like geo-spatial data where one might use more than one dimension (latitude and longitude) for partitioning data. Changing shard keys could involve: – adding a new index column, removing an existing column, changing the order of columns. An example system in this space is Replex [205]. Replex creates as many replicas as columns in the shard set. Each replica is partitioned by one of the keys in the set. Each Replex shard serves queries corresponding to the partitioning key as well as helps provide fault tolerance in presence of machine failures. Additionally they use chain replication for providing strong consistency. Since the system model is significantly different, matching based algorithm or the system designs for Morphus or Parqua cannot be applied here. We need better algorithms which can reduce network transfer without hurting some of the failure recovery guarantees of the system.

- **Selective Replication in Multi-Tenant Interactive Analytics Engine:** Getafix can be extended to handle multi-tenancy in distributed interactive analytics engines. In this problem, we assume multiple tenants are sharing a single deployment of the analytics engine. Each have their own SLOs for average or tail query latencies. Each tenant uses multiple namespaces to store their data at the backend. A single front end cluster is shared by all the tenants with much smaller total memory compared to the total disk space at the backend. We assume that each tenant's workload is known upfront and we can use that to find out the smallest number of cores required. In this multi-tenant extension, one would want to minimize the replication required while minimizing the makespan across all the tenants.

- **Dynamic Replication in Batch Processing Systems:** Getafix can also be extended to handle replication in batch processing systems. Scarlett [31] is the known system in this space. The solution is an intuitive heuristic but it tries to optimize solely for performance and not care about memory costs incurred by high replication. A key difference from interactive analytics engine is large variance in job running times. This can make it harder to differentiate between popular data vs unpopular data that are part of long running jobs. One would require more sophisticated data popularity prediction techniques.

- **Other Extensions to Getafix:** Currently, Druid does not support join operations.

Join represent a class of query semantics which require network transfer during query execution. In such scenarios, working memory can potentially dominate front end cluster memory consumption. Getafix does not reduce working set memory and can be an interesting problem to consider in future. Lastly, in this work, we have only dealt with DRAM. With the advent and rising popularity of newer memory technologies like NVRAM, PCM, etc, it will be interesting to explore how Getafix works with them.

- **Using NetCachier to improve job scheduling:** Future work would involve creating a unified resource scheduler for compute, network and storage. There are several solutions [206, 152] for assigning CPU resources to jobs with deadlines. They try to optimize for job running times, cluster utilization, etc. NetCachier can in principle be combined with these works for settings in which network, compute and storage are all scarce resources. Consider the case when two jobs J_1 and J_2 arrive at time 2 and 4 time units. Their expected runtime is 3 units. They have a deadline to finish the jobs by 7 time units. Lets assume they require two files which cannot be fetched at the same time because of network bandwidth limits. If the scheduler unknowing of this fact schedules them to start at the time unit 4, NetCachier will not be able to meet the deadline for one of the jobs. On the contrary, if the scheduler can start job J_1 at 2 time units and assuming the input file has already been created before the job arrived, then NetCachier can prefetch the file and complete job J_1 and subsequently J_2. One of the key challenges is the complexity of the problem grows as we try to solve this multi-resource scheduling problem.

References

[1] L. Columbus, "53% of companies are adopting big data analytics," 2017. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/12/24/53-of-companies-are-adopting-big-data-analytics/#466ab02139a1

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

[3] T. A. S. Foundation, "Hadoop," 2014. [Online]. Available: https://hadoop.apache.org

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113 pp. 10–10.

[5] iDatalabs, "Apache hadoop market share and competitors in big data," 2017. [Online]. Available: https://idatalabs.com/tech/products/apache-hadoop

[6] 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*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323 pp. 15–15.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/945445.945450 pp. 29–43.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972 pp. 1–10.

[9] 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*, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281 pp. 205–220.

[10] "Cassandra," http://cassandra.apache.org/, visited on 2014-04-29.

[11] "MongoDB," http://www.mongodb.org, visited on 2015-1-5.

[12] L. Columbus, "10 ways real-time business analytics are driving revenue," 2017. [Online]. Available: https://selecthub.com/business-analytics/10-ways-real-time-business-analytics-driving-revenue/

[13] T. A. S. Foundation, "Samza," 2015. [Online]. Available: https://samza.apache.org

[14] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *Proceedings of the 2015 ACM International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788 pp. 239–250.

[15] A. S. Foundation, "Apache kylin," 2015. [Online]. Available: http://kylin.apache.org/

[16] Microsoft, "Sql server," 2018. [Online]. Available: https://www.microsoft.com/en-us/sql-server/default.aspx

[17] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595631 pp. 157–168.

[18] Facebook, "PrestoDB," 2013. [Online]. Available: https://prestodb.io/

[19] Markets and Markets, "Streaming analytics market - worldwide market forecast and analysis (2015 - 2020)," 2016. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/streaming-analytics-market-64196229.html

[20] Seagate, "Data age 2025," 2017. [Online]. Available: https://www.seagate.com/our-story/data-age-2025/

[21] Cisco, "The zettabyte era: Trends and analysis," 2017. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026899 pp. 265–283.

[23] L. Hester, "Maximizing data value with a data lake," 2016. [Online]. Available: https://www.datasciencecentral.com/profiles/blogs/maximizing-data-value-with-a-data-lake

[24] "HBase," https://hbase.apache.org, visited on 2015-1-5.

[25] A. W. S. (AWS), "S3," 2018. [Online]. Available: https://aws.amazon.com/s3/

[26] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633 pp. 5:1–5:16.

[27] P. Leitner and J. Cito, "Patterns in the chaos–a study of performance variation and predictability in public iaas clouds," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, p. 15, 2016.

[28] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228326

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

[30] LinkedIn, "Pinot," 2015. [Online]. Available: https://github.com/linkedin/pinot/wiki

[31] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in Mapreduce clusters," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966472 pp. 287–300.

[32] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: Supporting online reconfigurations in sharded nosql systems," in *Proceedings of the 2015 IEEE International Conference on Autonomic Computing*, ser. ICAC '15. Washington, DC, USA: IEEE Computer Society, 2015. [Online]. Available: http://dx.doi.org/10.1109/ICAC.2015.42 pp. 1–10.

[33] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: Supporting online reconfigurations in sharded nosql systems," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 466–479, Oct 2017.

[34] Y. Shin, M. Ghosh, and I. Gupta, "Parqua: Online reconfigurations in virtual ring-based nosql systems," in *2015 International Conference on Cloud and Autonomic Computing*, Sept 2015, pp. 220–223.

[35] M. Ghosh, A. Raina, L. Xu, X. Qian, I. Gupta, and H. Gupta, "Popular is cheaper: Curtailing memory costs in interactive analytics engines," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190542 pp. 40:1–40:14.

[36] M. Ghosh, I. Gupta, S. Gupta, and N. Kumar, "Fast compaction algorithms for nosql databases," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, June 2015, pp. 452–461.

[37] "NoSQL market forecast 2015-2020, Market Research Media," http://www.marketresearchmedia.com/?p=568, 2012, visited on 2015-1-5.

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

[39] "How to change the Shard Key," http://stackoverflow.com/questions/6622635/how-to-change-the-shard-key, visited on 2015-1-5.

[40] "SMG Research Reveals DBA Tools Not Effective for Managing Database Change," http://www.datical.com/news/research-reveals-dba-tools-not-effective-for-database-change/, visited on 2015-04-11.

[41] D. Sjøberg, "Quantifying schema evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35–44, 1993.

[42] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[43] "Change shard key mongodb faq," http://docs.mongodb.org/manual/faq/sharding/#can-i-change-the-shard-key-after-sharding-a-collection, visited on 2015-1-5.

[44] 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.*, 2014. [Online]. Available: https://www.usenix.org/conference/icac14/technical-sessions/presentation/barker pp. 33–43.

[45] 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*, vol. 4, no. 8. VLDB Endowment, May 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002974.2002977 pp. 494–505.

[46] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *Proceedings of the 2011 ACM International Conference on Special Interest Group on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989356 pp. 301–312.

[47] "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.

[48] "Altering Cassandra column family primary key," http://stackoverflow.com/questions/18421668/alter-cassandra-column-family-primary-key-using-cassandra-cli-or-cql, visited on 2015-1-5.

[49] "The great primary key debate," http://www.techrepublic.com/article/the-great-primary-key-debate/, visited on 2015-1-5.

[50] 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 Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2491245

[51] "RethinkDB," http://rethinkdb.com/, visited on 2015-1-5.

[52] "CouchDB," http://couchdb.apache.org, visited on 2015-1-5.

[53] "Hungarian algorithm," http://en.wikipedia.org/wiki/Hungarian_algorithm, visited on 2015-1-5.

[54] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of 2008 the ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967 pp. 63–74.

[55] J. Kim, W. J. Dally, and D. Abts, "Efficient topologies for large-scale cluster networks," in *Proceedings of the 2010 Conference on Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference(OFC/NFOEC)*. IEEE, 2010, pp. 1–3.

[56] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250679 pp. 126–137.

[57] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *Proceedings of the ACM Special Interest Group on Data Communication 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018448 pp. 98–109.

[58] "Mongodb manual for geo-distributed deployment (2.4.9)," http://docs.mongodb.org/manual/tutorial/deploy-geographically-distributed-replica-set/, visited on 2015-1-5.

[59] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *Proceedings of the 7th ACM Conference on Recommender Systems*, ser. RecSys '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2507157.2507163 pp. 165–172.

[60] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, ser. OSDI '02. Boston, MA: USENIX Association, Dec. 2002, pp. 255–270.

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

[62] G. Pang, "Latencies gone wild!, AMPLab - UC Berkeley," https://amplab.cs.berkeley.edu/2011/10/20/latencies-gone-wild/, visited on 2015-1-5.

[63] "Google Cloud," https://cloud.google.com/, visited on 2015-1-5.

[64] M. J. Carey and H. Lu, "Load balancing in a locally distributed DB system," in *Proceedings of the 1986 ACM International Conference on Special Interest Group on Management of Data*, ser. SIGMOD '86. New York, NY, USA: ACM, 1986. [Online]. Available: http://doi.acm.org/10.1145/16894.16865 pp. 108–119.

[65] B. Kemme, A. Bartoli, and O. Babaoglu, "Online reconfiguration in replicated databases based on group communication," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, ser. DSN '01. Washington, DC, USA: IEEE Computer Society, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=647882.738226 pp. 117–130.

[66] 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*, vol. 6, no. 11. VLDB Endowment, Aug. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2536222.2536230 pp. 1045–1056.

[67] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi, "Squall: Fine-grained live reconfiguration for partitioned main memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2723372.2723726 pp. 299–313.

[68] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *Proceedings of the 1988 ACM International Conference on Special Interest Group on Management*, ser. SIGMOD '88. New York, NY, USA: ACM, 1988. [Online]. Available: http://doi.acm.org/10.1145/50202.50213 pp. 99–108.

[69] M. Mehta and D. J. DeWitt, "Data placement in shared-nothing parallel database systems," *The Very Large Database Journal*, vol. 6, no. 1, pp. 53–72, Feb. 1997. [Online]. Available: http://dx.doi.org/10.1007/s007780050033

[70] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: a database service for the cloud." in *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/cidr/cidr2011.html#CurinoJPMWMBZ11 pp. 235–240.

[71] M. S. Ardekani and D. B. Terry, "A self-configurable geo-replicated cloud storage system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685077 pp. 367–381.

[72] 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*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251203.1251223 pp. 273–286.

[73] 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*, ser. VEE '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254834 pp. 169–179.

[74] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy, ""Cut me some slack": Latency-aware live migration for databases," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2247596.2247647 pp. 432–443.

[75] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI '10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855730 pp. 19–19.

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

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

[78] W. G. Yee, "Orbitz: Technical challenges and opportunities in a leading online travel business," https://sites.google.com/site/gcasrworkshop/2015/program/wai-gen-yee, 2015.

[79] 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: IEEE, 2015.

[80] 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*, vol. 11, no. 1, pp. 17–32, Feb. 2003. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.808407

[81] "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.

[82] "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.

[83] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 230–243.

[84] Research and Markets, "Streaming analytics market by verticals - worldwide market forecast & analysis (2015 - 2020)," Report, June 2015. [Online]. Available: https://www.researchandmarkets.com/research/mpltnp/streaming

[85] T. A. S. Foundation, "Storm," 2015. [Online]. Available: https://storm.apache.org

[86] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik, "Distributed operation in the Borealis stream processing engine," in *Proceedings of the 2005 ACM International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066274 pp. 882–884.

[87] Amazon, "Redshift," 2012. [Online]. Available: https://aws.amazon.com/redshift/

[88] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal, "Mesa: A geo-replicated online data warehouse for google's advertising system," *Communications of the ACM*, vol. 59, no. 7, pp. 117–125, June 2016. [Online]. Available: http://doi.acm.org/10.1145/2936722

[89] A. Redshift, "Customer success," 2018. [Online]. Available: https://aws.amazon.com/redshift/customer-success/

[90] Metamarkets, "Powered by druid," 2018. [Online]. Available: http://druid.io/druid-powered.html

[91] H. Gupta, "Beyond hadoop at yahoo!: Interactive analytics with druid," Talk, September 2016. [Online]. Available: https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51640

[92] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273005 pp. 59–72.

[93] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851 pp. 11–11.

[94] Oracle, "Mysql," 2018. [Online]. Available: https://www.mysql.com

[95] Wikipedia, "Jaccard index," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[96] Wikipedia, "Bin packing problem," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Bin_packing_problem

[97] W. Stallings, *Operating Systems: Internals and Design Principles Edition: 5*. Pearson, 2005.

[98] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901354 pp. 36:1–36:15.

[99] A. W. S. (AWS), "Instance types," 2018. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

[100] Microsoft, "Blob storage," 2018. [Online]. Available: https://azure.microsoft.com/en-us/services/storage/blobs/

[101] Wikipedia, "Hungarian algorithm," 2018. [Online]. Available: http://en.wikipedia.org/wiki/Hungarian_algorithm

[102] Amazon, "Aws," 2018. [Online]. Available: https://aws.amazon.com/

[103] D. Inc., "Docker," 2018. [Online]. Available: https://www.docker.com/

[104] Emulab, "d430," 2018. [Online]. Available: https://wiki.emulab.net/wiki/d430

[105] Amazon, "Ebs," 2018. [Online]. Available: https://aws.amazon.com/ebs/pricing/

[106] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[107] O. Wolfson, S. Jajodia, and Y. Huang, "An adaptive data replication algorithm," *ACM Transactions on Database Systems*, vol. 22, no. 2, pp. 255–314, June 1997. [Online]. Available: http://doi.acm.org/10.1145/249978.249982

[108] T. Rabl and H.-A. Jacobsen, "Query centric partitioning and allocation for partially replicated database systems," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064052 pp. 315–330.

[109] P. M. G. Apers, "Data allocation in distributed database systems," *ACM Transactions on Database Systems*, vol. 13, no. 3, pp. 263–304, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/44498.45063

[110] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2523616.2525970 pp. 13:1–13:17.

[111] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: ACM, 1998. [Online]. Available: http://doi.acm.org/10.1145/291069.291036 pp. 139–149.

[112] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924949 pp. 75–88.

[113] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920853

[114] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proceedings of the 2012 ACM International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213844 pp. 61–72.

[115] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça, "Met: Workload aware elasticity for NoSQL," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465370 pp. 183–196.

[116] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 245–256, Nov. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735508.2735514

[117] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm blob storage system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '14. Berkeley, CA, USA: USENIX Association, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685078 pp. 383–398.

[118] A. Khandelwal, R. Agarwal, and I. Stoica, "Blowfish: Dynamic storage-performance tradeoff in data stores," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930643 pp. 485–500.

[119] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824076

[120] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A high-performance incremental query processor for diverse analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 401–412, Dec. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735496.2735503

[121] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920886

[122] Amazon, "Athena," 2018. [Online]. Available: https://aws.amazon.com/athena/

[123] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924962 pp. 265–278.

[124] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing mapreduce on heterogeneous clusters," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150984 pp. 61–74.

[125] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '08. Berkeley, CA, USA: USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744 pp. 29–42.

[126] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju, "Marla: Mapreduce for heterogeneous clusters," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: https://doi.org/10.1109/CCGrid.2012.135 pp. 49–56.

[127] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '03. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=644108.644201 pp. 546–554.

[128] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 8, no. 1, pp. 4:1–4:22, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2071379.2071383

[129] "RocksDB," http://rocksdb.org/, visited on 2014-11-20.

[130] M. Ghosh, I. Gupta, S. Gupta, and N. Kumar, "Fast compaction algorithms for nosql databases," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, June 2015, pp. 452–461.

[131] "LevelDB," http://google-opensource.blogspot.com/2011/07/leveldb-fast-persistent-key-value-store.html, visited on 2014-11-24.

[132] "Leveled Compaction in Cassandra," http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra, visited on 2014-11-24.

[133] "LevelDB in Riak," http://docs.basho.com/riak/latest/ops/advanced/backends/leveldb/, visited on 2014-11-24.

[134] "Size Tiered Compaction," http://shrikantbang.wordpress.com/2014/04/22/size-tiered-compaction-strategy-in-apache-cassandra/, visited on 2014-11-20.

[135] "Date Tiered Compaction," https://issues.apache.org/jira/browse/CASSANDRA-6602, visited on 2014-11-20.

[136] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "Logbase: A scalable log-structured database system in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1004–1015, June 2012. [Online]. Available: http://dx.doi.org/10.14778/2336664.2336673

[137] C. Mathieu, C. Staelin, and N. E. Young, "K-slot sstable stack compaction," *CoRR*, vol. abs/1407.3008, 2014. [Online]. Available: http://arxiv.org/abs/1407.3008

[138] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sept 1952.

[139] "Improving compaction in Cassandra with cardinality estimation," http://www.datastax.com/dev/blog/improving-compaction-in-cassandra-with-cardinality-estimation, visited on 2014-12-09.

[140] "Illinois Cloud Computing Testbed," http://cloud.cs.illinois.edu/, visited on 2014-11-20.

[141] R. Campbell, I. Gupta, M. Heath, S. Y. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Y. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y. C. Soh, "Open Cirrus Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855533.1855534

[142] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *IN AOFA 07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[143] "Dell EMC Digital Universe Survey: The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things, 2014," https://www.emc.com/leadership/digital-universe/index.htm.

[144] "InfoScale Enterprise," https://www.veritas.com/product/software-defined-storage/infoscale-enterprise.

[145] "Managing Hot and Cold Data Using a Unified Storage System," https://content.pivotal.io/blog/managing-hot-and-cold-data-using-a-unified-storage-system.

[146] "Shared Infrastructure for Big Data: Separating Hadoop Compute and Storage," http://www.bluedata.com/blog/2015/12/separating-hadoop-compute-and-storage/.

[147] "The Big Data Challenge: Intelligent Tiered Storage at Scale," http://www.cray.com/Assets/PDF/Integrated_Tiered_Storage_Whitepaper.pdf.

[148] "Amazon EC2," https://aws.amazon.com/ec2/.

[149] "D-Series Performance Expectations," https://azure.microsoft.com/en-us/blog/d-series-performance-expectations.

[150] "Amazon EBS Product Details," https://aws.amazon.com/ebs/details.

[151] "[GCE] Optimizing Persistent Disk and Local Storage Performance," https://cloud.google.com/compute/docs/disks/performance.

[152] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated slos for enterprise clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026887

[153] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: A software-defined storage architecture," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522723

[154] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018465

[155] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168847

[156] J. Chuzhoy, S. Guha, S. Khanna, and J. Naor, "Machine minimization for scheduling jobs with interval constraints," in *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, 2004, pp. 81–90.

[157] J. Chuzhoy and J. Naor, "New hardness results for congestion minimization and machine scheduling," *J. ACM*, vol. 53, no. 5, pp. 707–721, 2006.

[158] U. Feige, G. Kortsarz, and D. Peleg, "The dense k-subgraph problem," *Algorithmica*, vol. 29, no. 3, pp. 410–421, 2001.

[159] A. Bhaskara, M. Charikar, E. Chlamtac, U. Feige, and A. Vijayaraghavan, "Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k-subgraph," in *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, 2010, pp. 201–210.

[160] P. Bailis, E. Gan, K. Rong, and S. Suri, "Prioritizing attention in fast data: Principles and promise," *CIDR'17*.

[161] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudre-Mauroux, "Dependency-driven analytics: A compass for uncharted data oceans," ser. CIDR, 2017.

[162] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.

[163] "Cloudera Enterprise Reference Architecture for Azure Deployments," http://www.cloudera.com/documentation/other/reference-architecture/PDF/cloudera_ref_arch_azure.pdf.

[164] "Moving Data into HDFS from Amazon S3," http://documentation.altiscale.com/moving-data-from-s3-to-hdfs.

[165] "Hadoop HDFS Mover," https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html#Mover_-_A_New_Data_Migration_Tool.

[166] "Use HDFS-compatible Azure Blob storage with Hadoop in HDInsight," https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-hadoop-use-blob-storage.

[167] "Apache Gridmix," https://hadoop.apache.org/docs/r1.2.1/gridmix.html.

[168] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367519

[169] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787488

[170] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[171] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[172] A. Bestavros, "Using speculation to reduce server load and service time on the www," Boston, MA, USA, Tech. Rep., 1995.

[173] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 3, pp. 22–36, July 1996. [Online]. Available: http://doi.acm.org/10.1145/235160.235164

[174] J. Wang, "A survey of web caching schemes for the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/505696.505701

[175] S. Albers, S. Arora, and S. Khanna, "Page replacement for general caching problems," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '99. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=314500.314528

[176] M. Brehob, S. Wagner, E. Torng, and R. Enbody, "Optimal replacement is np-hardfor nonstandard caches," *IEEE Trans. Comput.*, vol. 53, no. 1, pp. 73–76, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1109/TC.2004.1255792

[177] S. Irani, "Page replacement with multi-size pages and applications to web caching," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997. [Online]. Available: http://doi.acm.org/10.1145/258533.258666

[178] N. Bansal, N. Buchbinder, and J. S. Naor, "A primal-dual randomized algorithm for weighted paging," *Journal of the ACM (JACM)*, vol. 59, no. 4, p. 19, 2012.

[179] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, ser. VLDB '85. VLDB Endowment, 1985. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286760.1286772

[180] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=645922.673462 pp. 330–341.

[181] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672996 pp. 439–450.

[182] S. Jiang and X. Zhang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '02. New York, NY, USA: ACM, 2002. [Online]. Available: http://doi.acm.org/10.1145/511334.511340

[183] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, ser. FAST '03. Berkeley, CA, USA: USENIX Association, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1090694.1090708

[184] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri, "Sharing buffer pool memory in multi-tenant relational database-as-a-service," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 726–737, 2015.

[185] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "Fairride: Near-optimal, fair cache sharing," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930637

[186] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016.

[187] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM 2011*.

[188] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea, "Chatty tenants and the cloud network sharing problem," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482644 pp. 171–184.

[189] C. Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM*, 2012.

[190] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM 2014*.

[191] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM 2014*.

[192] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012

[193] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486019

[194] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang, "Guaranteeing deadlines for inter-datacenter transfers," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741957

[195] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula, "Calendaring for wide area networks," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626336

[196] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic pricing and traffic engineering for timely inter-datacenter transfers," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934893

[197] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2390231.2390237

[198] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787480

[199] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934880

[200] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell, "Ambry: Linkedin's scalable geo-distributed object store," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 253–265.

[201] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[202] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endow.*

[203] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–15.

[204] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt, "Cast: Tiering storage for data analytics in the cloud," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749252

[205] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi, "Replex: A scalable, highly available multi-index data store," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3026991 pp. 337–350.

[206] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670981