OPTIMIZING INTERACTIVE ANALYTICS ENGINES FOR HETEROGENEOUS CLUSTERS

BY

ASHWINI RAINA

THESIS

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

Urbana, Illinois

Adviser:

Professor Indranil Gupta

ABSTRACT

This thesis targets the growing area of interactive data analytics engines. It builds upon a system called Getafix, an intelligent data replication and placement algorithm, and optimizes Getafix for running mixed queries over a heterogeneous cluster. The new algorithm is called Getafix-H, a cluster aware version of Getafix replication algorithm, with built-in optimizations for segment balancing and cluster auto-tiering. We integrated Getafix-H as an extension to Getafix inside Druid, a modern open-source interactive data analytics engine. We present experimental results using workloads from Yahoo!'s production Druid cluster. Compared to Getafix, Getafix-H improves the tail latency by 18% and reduces memory usage by upto 27% (2 - 3X improvement over Scarlett). In presence of stragglers, Getafix-H improves tail latency by 55% and reduces memory usage by upto 20% compared to Getafix. Getafix-H enables sysadmins to auto-tier a heterogeneous cluster with the tiering accuracy of upto 80%.

To my mother.

ACKNOWLEDGMENTS

I want to thank my advisor Prof. Indranil Gupta for providing me a research opportunity in his group. This thesis came about a very tough phase of my life. I thank my advisor for being immensely supportive throughout and providing valuable academic advice.

I would like to thank Mainak Ghosh without whom this work wouldn't have been possible. I also want to thank Vipul Harsh for stimulating conversations and brutally honest feedback. At UIUC, my interactions with Atul Sandur, Bhargav Mangipudi, Nitin Bhat, Vishaal Mohan, Omkar Thakur, Umang Mathur, Faria Kalim, Mayank Bhatt and Jayasi Mehar have greatly shaped my research interests.

I want to thank my family for supporting me throughout this process. I deeply thank my wife Shruti who is the biggest source of inspiration in my life.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Real-time analytics is projected to grow annually at a rate of 31% [1]. Apart from stream processing engines, which have received much attention [2, 3, 4], real time analytics now also includes the burgeoning area of interactive data analytics engines such as Druid [5], Redshift [6], Mesa [7], Presto [8] and Pinot [9]. These systems have seen widespread adoption [10, 11] in companies which require applications to support sub-second query response time. Applications span usage analytics, revenue reporting, spam analytics, ad feedback, and others [12]. 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 [12].

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 [13] or Amazon S3 [14]. 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[*]. 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.

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.

Getafix [15] proposes an intelligent scheme for placement of data segments in interactive analytics engines. The key idea is to exploit the strong evidence [16] that at any given point of time, some data segments are more popular than others. Getafix analyzed traces from

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

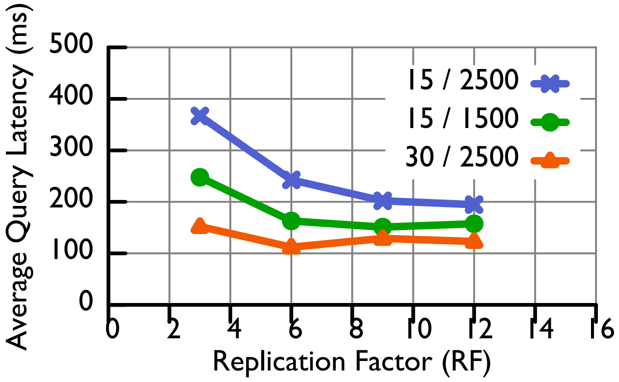


Figure 1.1: Average Query Latency observed with varying replication factors for different (cluster size / query injection rate) combinations.

Yahoo!'s Druid cluster, and found that the top 1% of data is an order of magnitude more popular than the bottom 40%. Figure 1.1 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. Getafix achieves the knee of the curve for individual segments in an adaptive way.

The Getafix replication algorithm makes certain homogeneity assumptions about the cluster environment. It does segment allocation assuming equal capacity compute nodes. This assumption does not hold for heterogeneous clusters or clusters with straggling nodes. Getafix's segment placement decisions are also based on this assumption. Ideally, a segment placement algorithm should assign more popular segments to powerful compute nodes and unpopular segments to lesser capable compute nodes. Since Getafix treats all compute nodes the same, it's segment placement decisions aren't fully aligned with the cluster environment. In Chapter 4, we show how a heterogeneity aware segment placement scheme can auto-tier a cluster, eschewing manual sysadmin work. Getafix is also prone to producing imbalanced assignment of segments to HNs. Such skewed assignment can saturate the memory of some HNs, while other HN's memories stay underutilized. All these shortcomings result in increased memory usage and higher query latency in heterogeneous clusters.

1.1 CONTRIBUTIONS OF THIS THESIS

In this thesis we present a system called Getafix-H, a variant of Getafix, that exploits the heterogeneity of the underlying cluster and optimizes for memory reduction and query latency performance. Getafix-H estimates the capacity of the cluster nodes by continually measuring query injection rate, segment popularity and query computation time on each node; and makes replication decisions based on that.

We integrated Getafix-H as an extension to Getafix inside Druid [5], 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-H to two baselines: 1) Getafix, and 2) Scarlett [16], which solves replication in batch systems like Hadoop [17], Dryad [18], etc. Compared to Getafix, Getafix-H improves the tail latency by 18% and reduces memory usage by upto 27%. Getafix had previously demonstrated memory reductions of 1.45 - 2.15× compared to Scarlett. Getafix-H improves the memory reduction to 2 - 3×. In presence of stragglers, Getafix-H improves tail latency by 55% and reduces memory usage by upto 20% compared to Getafix. Getafix-H also enables auto-tiering a heterogeneous cluster with a tiering accuracy of 80%.

The main contributions of this thesis are:

- We present a cluster-aware variant of Getafix replication algorithm, called Getafix-H, that supports segment balancing and auto-tiering.
- We present two classes of query routing schemes for Getafix-H system.
- We implement Getafix-H as an extension to Getafix inside Druid [5].
- We evaluate Getafix-H using workload derived from Yahoo! production clusters.

1.2 THESIS ORGANIZATION

This thesis is organized as follows:

- Chapter 2 introduces the segment replication problem and revisits the high level design of Getafix replication scheme.
- In Chapter 3, we present the design of Getafix-H and discuss the segment balancer and auto-tiering optimizations.
- In Chapter 4, we evaluate Getafix-H using workload derived from Yahoo! production clusters.
- Chapter 5 briefly describes related work.
- Chapter 6 concludes this thesis and discusses future work.

CHAPTER 2: BACKGROUND

This chapter presents an overview of the interactive analytics engine system model. It discusses the segment replication problem in such system and explains the Getafix segment replication algorithm.

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 [5].

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 results are then indexed and pushed into deep storage. This chunk of results 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). 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.

In Druid, all internal coordination like segment loading between coordinator and HN is handled by a Zookeeper [19] cluster. Druid also uses MySQL [20] for storing metadata from segments and failure recovery. As a result, the broker, coordinator, and historical nodes are all stateless. This enables fast recovery by spinning up a new machine.

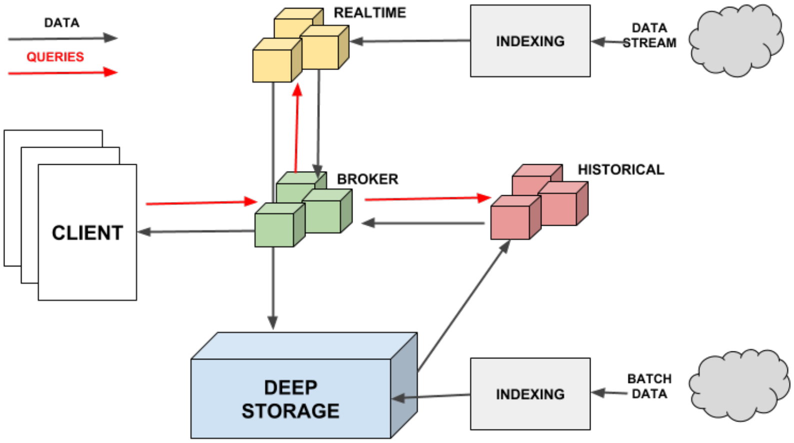
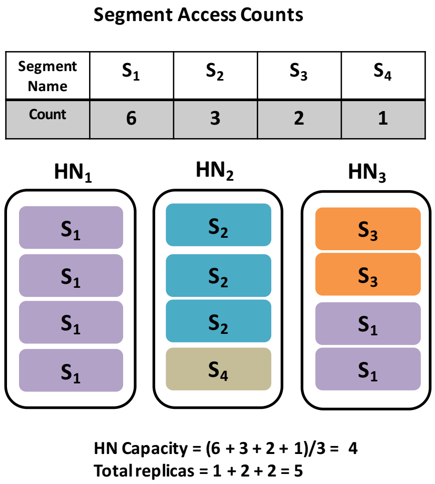


Figure 2.1: Main components of druid architecture and associated dataflow [5]

2.2 SEGMENT REPLICATION PROBLEM AND GETAFIX

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.


Segment Access Counts

Segment Name	S_1	S_2	S_3	S_4
Count	6	3	2	1

HN Capacity = (6 + 3 + 2 + 1)/3 = 4
Total replicas = 1 + 2 + 2 = 5

Figure 2.2: 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.

Consider the segment-query pairs in the given static workload, i.e., all pairs (s_j, q_i) where query q_i needs to access segment s_j. Spreading these segment-query 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 segment-query 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 fetched by the above layout is 7. The minimum number of replicas 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 2.2).

Formally, the input to the segment replication problem is: 1) segment access request counts $C = \{c_1, \ldots c_m\}$ for k queries accessing m segments, and 2) n HNs each with capacity $\lceil \frac{\sum_i c_i}{n} \rceil$ ("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 $\sum_i \sum_j x_{ij}$.

Algorithm 2.1: Generalized Allocation Algorithm.

```
input: $C$: Access counts for each segment
       $nodelist$: List of HNs
Algorithm ModifiedFit($C, nodelist$)
    $n \leftarrow$ Length($nodelist$)
    $capacity \leftarrow \lceil \frac{\sum_{C_i \in C} |C_i|}{n} \rceil$
    $binCap \leftarrow$ InitArray($n, capacity$)
    $priorityQueue \leftarrow$ BuildMaxHeap($C$)
    while !Empty($priorityQueue$) do
        $(segment, count) \leftarrow$ Extract($priorityQueue$)
        $(left, bin) \leftarrow$ ChooseHistoricalNode($count, binCap$)
        LoadSegment($nodelist, bin, segment$)
        if $left > 0$ then
            Insert($priorityQueue, (segment, left)$)
```

Getafix solves this problem as a *colored* variant of the traditional bin packing problem [21]. Algorithm 2.1 depicts the 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_i from the head of the queue, and allocates the segment-query 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 segment-query pairs, and finally, the segment is re-inserted back into the priority queue at the appropriate position (via binary search). The ChooseHistoricalNode problem bears similarities with segmentation in traditional operating systems [22]. Getafix uses a modified variant of best fit strategy called ModifiedBestFit.

To handle dynamically arriving segments and queries, Getafix runs Algorithm 2.1 in periodic rounds. In each round, it collects query load statistics from all HNs and runs Algorithm 2.1 which 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.

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:

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

Next, the coordinator runs ModifiedBestFit using Popularity(.) values. 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). Getafix sets the round duration to 5 seconds, which allows us to catch popularity changes early but not react too aggressively. This duration can be chosen based on the segment creation frequency.

CHAPTER 3: DATA REPLICATION IN HETEROGENEOUS CLUSTERS

Compute clusters exhibit heterogeneous behavior due to various reasons. The primary source of heterogeneity arises from cluster hardware. Clusters may be intentionally provisioned using heterogeneous hardware to support the application requirements. In many cases heterogeneity arises due to multiple generations of hardware. In virtualized cloud environments (like AWS EC2), co-location of multiple VMs on a single physical host may also cause heterogeneity. Straggling nodes (called stragglers) are another subtle but detrimental form of heterogeneity.

A data replication algorithm that is oblivious to cluster hardware configuration may underutilize the cluster. On the other hand, a data replication algorithm oblivious to dynamic cluster conditions (caused by stragglers) will suffer from poor performance. Getafix replication algorithm assumes a homogeneous cluster consisting of machines with equal compute capacity. Such an assumption isn't well suited for heterogeneous environments. In this chapter we introduce Getafix-H, a data replication design that relaxes those assumptions and present modifications to Getafix for heterogeneous settings.

3.1 CAPACITY AWARE ALLOCATION

Tailoring Getafix for heterogeneous clusters requires knowledge of the compute capacity of individual nodes. HN capacities can be estimated either statically, using offline benchmarking of nodes [23], or dynamically, by monitoring task progress of nodes [17]. Design of Getafix-H allows HN capacities to be estimated independently using any preferred method and then input to the ModifiedBestFit algorithm. Getafix-H uses a dynamic run-time scheme for estimating HN capacities. In each replication round, the coordinator collects statistics on total *CPU time* (excluding disc or network IO) spent in query processing in the previous round, from each HN. Powerful HNs, equipped with bigger memory and cores, process more queries and report higher CPU time values compared to weaker HNs. The CPU time value reported by an HN is used as a proxy for its capacity. Instead of assuming equal capacity in Algorithm 2.1 (line 3), we distribute the total query load proportionally among HNs based on their estimated *compute capacities*. We call this new scheme Capacity-Aware ModifiedBestFit (depicted in Algorithm 3.1).

Although Capacity-Aware ModifiedBestFit was designed to address the heterogeneity

Algorithm 3.1: Capacity-Aware Allocation Algorithm.

```
input: $C$: Access counts for each segment
       $T$: Query CPU time for each HN
       $nodelist$: List of HNs
Algorithm Capacity-Aware ModifiedFit($C, nodelist$)
    $n \leftarrow \textsc{Length}(nodelist)$
    $totalCapacity \leftarrow \sum_{C_i \in C} |C_i|$
    $capacity_i \leftarrow \lceil \frac{T_i \times totalCapacity}{\sum_{T_i \in T} |T_i|} \rceil$
    $binCap \leftarrow \textsc{InitArray}(n, capacity)$
    $priorityQueue \leftarrow \textsc{BuildMaxHeap}(C)$
    while $!\textsc{Empty}(priorityQueue)$ do
        $(segment, count) \leftarrow \textsc{Extract}(priorityQueue)$
        $(left, bin) \leftarrow \textsc{ChooseHistoricalNode}$
        $(count, binCap)$
        $\textsc{LoadSegment}(nodelist, bin, segment)$
        if $left > 0$ then
            $\textsc{Insert}(priorityQueue, (segment, left))$
```

caused by the cluster hardware, it also works well against heterogeneity caused due to dynamic cluster conditions. Another useful side effect of Capacity-Aware ModifiedBestFit is it's natural ability to tier a cluster based on the node capacities. We discuss them next.

3.1.1 Straggler Mitigation

Heterogeneity in clusters can also arise due to unexpected machine slowdowns. Some nodes may become stragglers due to bad memory, slow disk, flaky NIC, background tasks, etc. A good data replication scheme should avoid placing popular segments on straggling nodes as it can severely degrade the query latency performance. Getafix divides the segment CPU time equally across all HNs, irrespective of their individual capacities. This makes it vulnerable to assigning a popular segment to a straggling node. Since Capacity-Aware ModifiedBestFit runs periodically, it can detect and mitigate the effect of stragglers with minimal overhead. 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 attribute lesser capacity to such nodes. Lesser capacity will ensure popular segments are not assigned to these HNs.

3.1.2 Avoiding Manual Tiering

A typical deployment of an interactive analytics engine happens over a heterogeneous cluster, wherein some nodes are more compute and memory capable than other nodes. Under such deployment, popular segments are assigned to powerful nodes and queries for such segments are served from memory. Today system administrators manually configure clusters into tiers by grouping machines with similar hardware characteristics into a single tier. They use hard-coded rules for placing segments within these tiers, with recent (popular) segments assigned to the hot tier. This approach doesn't react to changes in individual segment popularity very well, thus resulting in in-efficient memory use and higher query latency.

Eschewing this manual approach, Getafix-H continuously tracks changes in segment popularity and cluster configuration, to automatically move popular replicas to powerful HNs, thereby creating its own tiers. Thus, Getafix-H can help avoid laborious sysadmin activity and cut opex (operational expenses) of the cluster.

3.2 BALANCING SEGMENT LOAD

For skewed segment access distributions, the output of ModifiedBestFit (in Getafix) could produce imbalanced assignment of segments to HNs. Segment imbalance creates two issues. Firstly, less-loaded HNs, those with fewer segments, could be idle in some scenarios (e.g., if some segments became unpopular), thus creating query load imbalance. Secondly, in a private cloud setting, cluster hardware needs to be provisioned to meet the maximum memory usage requirement. The former issue affects the query latency performance while the latter issue impacts the cluster provisioning dollar cost. In traditional systems, such imbalances require continuous intervention by human operators. We wish to minimize the maximum memory used by any HN in the system in order to achieve segment balancing. 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. We define *segment load* of a HN as the number of segments assigned to that HN. Starting with the output of Capacity-Aware 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 an HN is the value of the corresponding routing table entry.

It picks one replica at a time from this list and assign 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:

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

In our evaluation (§4.4), we found that a default $\gamma = 20\%$ gives the best segment balance with minimal impact on query load balance. This is checked after every Capacity-Aware ModifiedBestFit round.

3.3 QUERY ROUTING

As depicted in Figure 2.1, clients issue queries to the brokers. A single query may access one or more segments. The query routing problem entails brokers deciding which HNs an incoming query should be run at. A good segment replication algorithm must be matched with a suitable query routing scheme in order to achieve best query latency performance. A query routing scheme that isn't load balanced or isn't in tune with the data replication scheme, can cause hot spots in the cluster and impact query latency. With Getafix-H, we explore two types of routing schemes (described below).

3.3.1 Allocation Based Query Routing (ABR)

Apart from segment placement, Capacity-Aware ModifiedBestFit also provides sufficient information to build a query routing table. Concretely, Capacity-Aware ModifiedBestFit proportionally allocates the total CPU time among each replica of a segment. In our running example (Figure 2.2), 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-H creates a routing table that captures exact *query proportions*. The full routing table for this example is depicted in Table 3.1.

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.

	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 3.1: Routing Table for Figure 2.2. Each entry represents percentage of queries accessing segment S_i to be routed to HN_j.

3.3.2 Load Based Query Routing (LBR)

In ABR, routing table updates happen periodically (in rounds). In our experiments we observed that ABR lags in adapting to fast changes in workload. This is because queries complete much faster than a round duration and the segment popularity trends may change within a single round. 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 [24], this scheme works well in our evaluations because of its small overhead.

CHAPTER 4: EVALUATION

In this chapter, we present the evaluation of Getafix-H and discuss key results. We evaluate Getafix-H on both a private cloud (Emulab [25]) and a public cloud (AWS [26]). We use workload traces from Yahoo!'s production Druid cluster. We summarize our results here:

- Compared to Getafix, Getafix-H 27% less memory, while reducing the tail latency by 18%. It improves the performance of Getafix, over the best existing strategy (Scarlett), by consuming 2 - 3× less memory, while improving the makespan (16%).
- Compared to Getafix, Getafix-H improves tail query latency by 55% when 10% of the nodes are slow and by 17 - 22% when there is a mix of nodes in the cluster. It 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%.

4.1 METHODOLOGY

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

- *Emulab*: We deploy Druid on dedicated machines as well as on Docker [27] containers (to constrain disk for GC experiment). We use d430 [28] machines each with two 2.4 GHz 64-bit 8-Core processor, 64 GB RAM, connected using a 10Gbps network.
- *AWS*: We use m4.4xlarge [29] instances (16 cores, 64 GB memory), S3 [14] as the deep storage, and Amazon EBS General Purpose SSD (gp2) volumes [30] 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 are longer than top-K and timeseries. There can be considerable deviation in runtime among groupby queries themselves based on how many dimensions were 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 [16] 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. We borrow and re-use Getafix's implementation of Scarlett.

  Getafix's implementation of Scarlett is based on a round-robin algorithm. 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. The algorithm collects 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 Getafix did not cap network budget usage, we do not do it for Scarlett (for fairness in comparison).

Metrics. For the private cloud, we measure, across the entire run: 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. We also measure: 1) average and 99th percentile (tail) query latency and 2) makespan.

4.2 EVALUATION GOALS

Through our evaluation, we want to answer the following questions.

1. How does Getafix-H compare to Getafix-B (in terms of memory, query latency and makespan) for heterogeneous clusters with mixed hardware type?

2. How does Getafix-H compare to Getafix-B (in terms of memory, query latency and makespan) in presence of stragglers?

3. How efficiently can Getafix-H tier a heterogeneous cluster compared to Getafix-B?

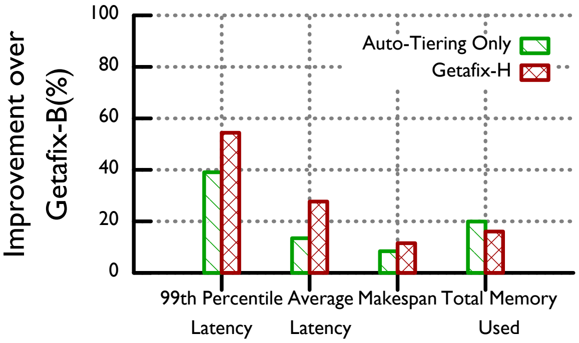


Figure 4.1: Emulab Experiments: Improvement in 99th Percentile, Average Query latency, Makespan and Total Memory Used with Getafix-H compared to Getafix-B. Experiment performed with 2 HNs straggling among 20.

4. What amount of segment balancing offers the best trade-off between maximum memory used and query performance?

5. How does LBR and ABR routing schemes compare to each other?

4.3 CLUSTER HETEROGENEITY

We evaluate the performance of Capacity-Aware ModifiedBestFit (§3.1) (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).

4.3.1 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.

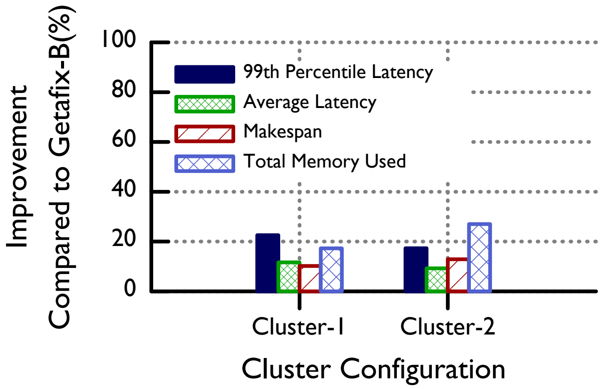


Figure 4.2: AWS Experiments: Improvement in 99th Percentile, Average Query latency, Makespan and Total Memory Used with Getafix-H compared to Getafix-B. Experiments performed with 2 different node mixtures and clients (refer Table 4.1)

Figure 4.1 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.

4.3.2 Tiered Clusters

Experiments are run in AWS on two cluster configurations consisting of mixed EC2 instances as shown in Table 4.1. 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 4.1: AWS HN heterogeneous cluster configurations.

Figure 4.2 shows that for Cluster-1, with a core mix of 48:48:24 (hot:warm:cold), Getafix-

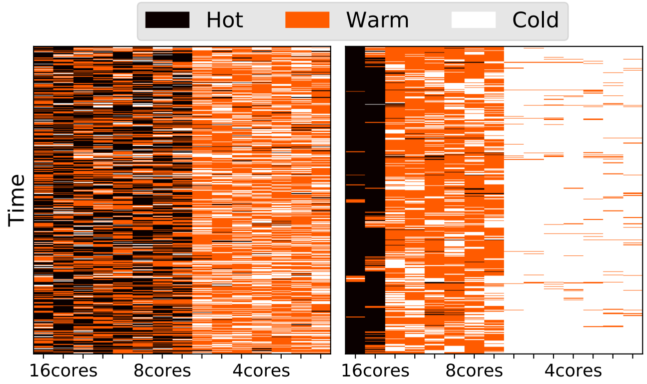


Figure 4.3: AWS Experiments: Getafix-B on left, Getafix-H on right. Effectiveness of Auto-Tiering shown using heat map. X-axis represents HNs sorted by the number of cores they have. Y-axis plots a period of time in the duration of the experiments. For each time, we classify HNs as hot, warm and cold (represented with 3 different colors) based on the reported CPU time for processed queries.

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 4.3. 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 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 4.1). 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).

4.4 SEGMENT BALANCER TRADEOFF

In §3.2, 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 4.4 quantifies γ's impact in a cluster of 20 HNs and 15 clients (labels are γ values).

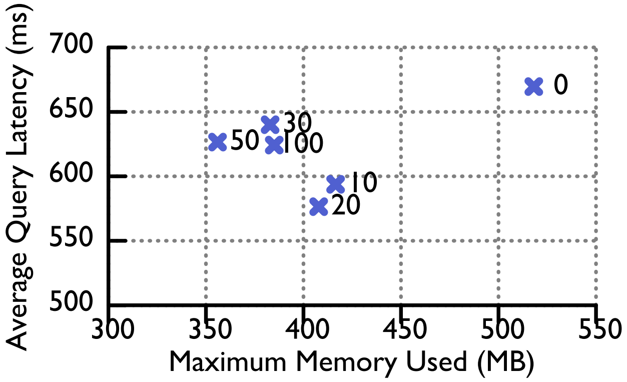


Figure 4.4: Emulab Experiments: Getafix – Maximum memory vs. Query Latency Tradeoff for different γ values. Lower is better on both axes.

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 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 COMPARING QUERY ROUTING SCHEMES

We evaluate three routing schemes, of which two are new: 1) ABR: Allocation Based Query Routing from §3.3. 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 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 4.5 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

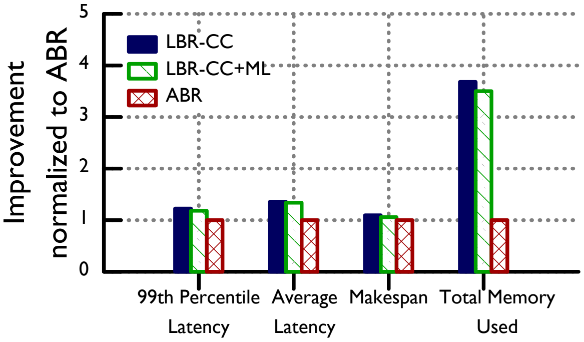


Figure 4.5: Emulab Experiments: Comparing 3 different query routing strategies on Getafix-B – 1) LBR-CC, 2) LBR-CC+ML, 3) ABR. Higher is better.

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.

CHAPTER 5: RELATED WORK

Allocation Problem: Our problem has similarities to the *data allocation problem* [31] in databases which tries to optimize for performance [32, 33] and/or network bandwidth [34]. A generalized version of the problem has been shown to be NP-hard [31]. Typical heuristics used are best fit and first fit [35, 36] or evolutionary algorithms [33]. 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 [37] 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 [38], whose techniques minimize cross-partition transactions in graph databases. There are other works which look at adaptive partitioning for OLTP systems [39] and NoSQL databases [40] respectively, however they do not explore Druid-like interactive analytics engines. E-Store [41] 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 [42] 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 [43], 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 [44, 45, 46, 8] 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 [13]. Since Getafix uses data access times and not query semantics, it can reduce memory usage generally.

Amazon Athena [47] and Presto [8] 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 [5], Pinot [9], Redshift [6], Mesa [7], 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 [48, 49, 50, 51]. 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.

CHAPTER 6: SUMMARY AND FUTURE WORK

We have presented data replication techniques intended for interactive data analytics engines like Druid, Pinot, etc. running in heterogeneous clusters. Our technique uses latest (running) popularity of data segments and the cluster node capacity to determine data placement and replication level at compute nodes. The new algorithm is called Getafix-H, a cluster aware version of Getafix replication algorithm, with built-in optimizations for segment balancing and cluster auto-tiering. We integrated Getafix-H as an extension to Getafix inside Druid, a modern open-source interactive data analytics engine. We present experimental results using workloads from Yahoo!'s production Druid cluster. Compared to Getafix, Getafix-H improves the tail latency by 18% and reduces memory usage by upto 27% (2 - 3X improvement over Scarlett). In presence of stragglers, Getafix-H improves tail latency by 55% and reduces memory usage by upto 20%. Getafix-H enables sysadmins to auto-tier a heterogeneous cluster, with the tiering accuracy of upto 80%.

Future Work: There are some important research directions that our work hasn't explored. Capacity estimation of the cluster nodes is at the core of Getafix-H. Investigation into better capacity estimation techniques should yield better system performance. In the current form, neither Getafix nor Getafix-H puts any cap on the amount of segments assigned to memory. Segment balancer alleviates this problem but doesn't fix it completely. An important extension of this work would be to design an algorithm that is constrained by the memory capacity of HNs. The core idea of Getafix could be applied to batch systems (like Hadoop) as well. It will be interesting to compare the performance of Getafix's ideas on Hadoop.

REFERENCES

[1] "Global Streaming Analytics Market Forecast & Analysis 2015-2020." https://tinyurl.com/hgbvajf. visited on 2017-2-12.

[2] "Apache Storm." http://storm.apache.org. visited on 2017-2-26.

[3] 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 International Conference on Management of Data*, SIGMOD '05, (New York, NY, USA), ACM, 2005.

[4] 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 International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), ACM, 2015.

[5] 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 International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), ACM, 2014.

[6] "Amazon Redshift." https://aws.amazon.com/redshift/. visited on 2016-3-2.

[7] 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, June 2016.

[8] "PrestoDB." https://prestodb.io/. visited on 2017-9-27.

[9] "LinkedIn Pinot." https://github.com/linkedin/pinot. visited on 2017-2-26.

[10] "Amazon Redshift Customer Success." https://aws.amazon.com/redshift/customer-success/. visited on 2017-2-12.

[11] "Powered by Druid." http://druid.io/druid-powered.html. visited on 2017-2-12.

[12] "Beyond Hadoop at Yahoo!: Interactive analytics with Druid." https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51640. visited on 2017-2-12.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 2010 IEEE Symposium on Mass Storage Systems and Technologies*, MSST '10, (Washington, DC, USA), IEEE Computer Society, 2010.

[14] "Amazon S3." https://aws.amazon.com/s3/. visited on 2017-2-26.

[15] 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 2018 European Conference on Computer Systems*, EuroSys '18, (New York, NY, USA), ACM, 2018.

[16] 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 2011 European Conference on Computer Systems*, EuroSys '11, (New York, NY, USA), ACM, 2011.

[17] "Hadoop." https://hadoop.apache.org. visited on 2017-2-26.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), ACM, 2007.

[19] 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 USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), USENIX Association, 2010.

[20] "MySQL." https://www.mysql.com. visited on 2017-3-1.

[21] "Bin Packing Problem." https://en.wikipedia.org/wiki/Bin$_{p}acking_{p}roblem.visitedon2016-3-2$.

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

[23] "AWS EC2 instance benchmarking." http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html. visited on 2017-10-2.

[24] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *Proceedings of the 2016 European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), ACM, 2016.

[25] 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 2002 Symposium on Operating Systems Design and Implementation, OSDI'02*, (Boston, MA), USENIX Association, Dec. 2002.

[26] "Amazon AWS." https://aws.amazon.com/. visited on 2017-10-18.

[27] "Docker." https://www.docker.com/. visited on 2017-3-1.

[28] "Emulab." https://wiki.emulab.net/wiki/d430. visited on 2016-3-2.

[29] "M4 Instance Type." https://aws.amazon.com/ec2/instance-types/. visited on 2017-10-18.

[30] "Amazon EBS." https://aws.amazon.com/ebs/pricing/. visited on 2017-10-18.

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

[32] O. Wolfson, S. Jajodia, and Y. Huang, "An adaptive data replication algorithm," *ACM Transactions on Database Systems*, vol. 22, June 1997.

[33] 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*, SIGMOD '17, (New York, NY, USA), ACM, 2017.

[34] P. M. G. Apers, "Data allocation in distributed database systems," *ACM Trans. Database Syst.*, vol. 13, Sept. 1988.

[35] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proceedings of the 2013 Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), ACM, 2013.

[36] 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*, ASPLOS VIII, (New York, NY, USA), ACM, 1998.

[37] 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 2010 Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), USENIX Association, 2010.

[38] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *Proceedings of the 2010 VLDB Endowment*, vol. 3, Sept. 2010.

[39] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proceedings of the 2012 International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), ACM, 2012.

[40] 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 2013 European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), ACM, 2013.

[41] 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 2014 VLDB Endowment*, vol. 8, Nov. 2014.

[42] 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 2014 Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), USENIX Association, 2014.

[43] A. Khandelwal, R. Agarwal, and I. Stoica, "Blowfish: Dynamic storage-performance tradeoff in data stores," in *Proceedings of the 2016 Conference on Networked Systems Design and Implementation*, NSDI'16, (Berkeley, CA, USA), USENIX Association, 2016.

[44] 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 2015 VLDB Endowment*, vol. 8, Aug. 2015.

[45] 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 2014 VLDB Endowment*, vol. 8, Dec. 2014.

[46] 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 2010 VLDB Endowment*, vol. 3, Sept. 2010.

[47] "Amazon Athena." https://aws.amazon.com/athena/. visited on 2018-2-11.

[48] 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 2010 Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), USENIX Association, 2010.

[49] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing mapreduce on heterogeneous clusters," in *Proceedings of the 2012 International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, (New York, NY, USA), ACM, 2012.

[50] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 2008 Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), USENIX Association, 2008.

[51] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju, "Marla: Mapreduce for heterogeneous clusters," in *Proceedings of the 2012 International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, (Washington, DC, USA), IEEE Computer Society, 2012.