# Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines

Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta
University of Illinois at Urbana-Champaign
{mghosh4,raina4,lexu1,qian13,indy}@illinois.edu

Himanshu Gupta
Oath Inc.
himanshg@oath.com

## ABSTRACT

This paper targets the growing area of interactive data analytics engines. We present a system called Getafix that intelligently decides replication levels and replica placement for data segments, in a way that is responsive to changing popularity of data access by incoming queries. We present an optimal solution to the static version of the problem, achieving minimality in both makespan and replication factor. Based on this intuition we build the Getafix system to handle queries and segments arriving in real time. We integrated Getafix into Druid, a modern open-source interactive data analytics engine. We present experimental results using workloads from Yahoo!'s production Druid cluster. Compared to existing work, Getafix achieves comparable query latency (both average and tail), while using 1.45-2.15× less memory in a private cloud. In a public cloud, for a 100 TB hot dataset size, Getafix can cut dollar costs by as much as 10 million annually with negligible performance impact.

## CCS CONCEPTS

• **Information systems → Online analytical processing engines**; **Cloud based storage**; *Distributed storage*; *Data warehouses*;

## KEYWORDS

Interactive data analytics engine, Adaptive replication, Memory reduction, Public clouds

**ACM Reference Format:**
Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta and Himanshu Gupta. 2018. Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190542

## 1 INTRODUCTION

Real-time analytics is projected to grow annually at a rate of 31% [42]. Apart from stream processing engines, which have received much attention [2, 21, 30], real time analytics now also includes the burgeoning area of interactive data analytics engines such as Druid [51], Redshift [4], Mesa [23], Presto [18] and Pinot [31]. These systems have seen widespread adoption [33, 41] in companies which require applications to support sub-second query response time. Applications span usage analytics, revenue reporting, spam analytics, ad feedback, and others [24]. 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 [24].

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 [43] or Amazon S3 [12]. 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.

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 paper proposes new intelligent schemes for placement of data segments in interactive analytics engines. The key idea is to exploit the strong evidence [8] 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 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

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


*EuroSys '18, April 23–26, 2018, Porto, Portugal*

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190542

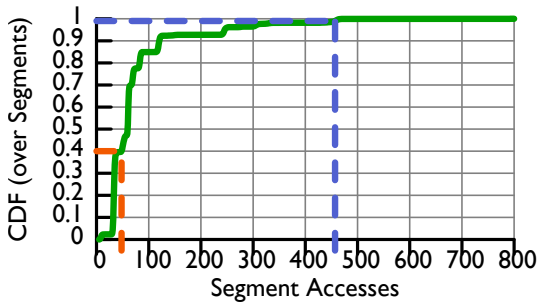



**Figure 1: CDF of segment popularity collected from Yahoo! production trace.**

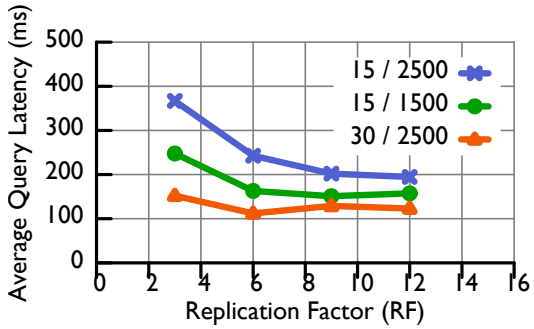



**Figure 2: Average Query Latency observed with varying replication factors for different (cluster size / query injection rate) combinations.**

it is manual, laborious, and cannot adapt in real time to changes in query patterns.

Figure 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 [51] approximate popularity by over-replicating data that was ingested recently (few hours to days). While there is correlation with recency, popularity needs to be treated as a first class citizen. Figure 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 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 (§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 [8]. 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, 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 [51], 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 [8], which solves replication in batch systems like Hadoop [20], Dryad [28], 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 savings of $10 M).

The main contributions of this paper are:

- We present workload characteristics of segment popularity in interactive data analytics engines (§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 (§3).
- We design the Getafix system to handle dynamic query and segment workloads (§4). We implement Getafix into Druid, a modern interactive data analytics engine.
- We evaluate Getafix using workload derived from Yahoo! production clusters (§5).

## 2 BACKGROUND

### 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 [51].

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.

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

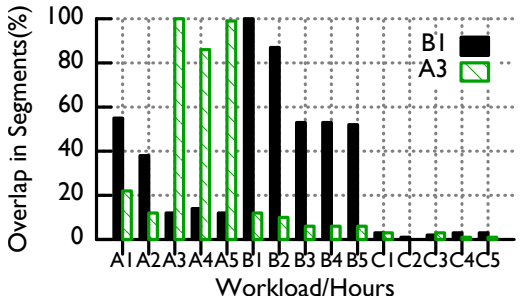



**Figure 3: Measures overlap in segment accesses across different hours of Yahoo! production trace. Each trace identified with a id (A/B/C: see Table 1) and the *i*th hour.**

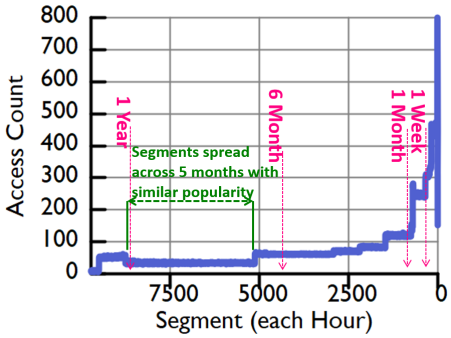



**Figure 4: Popularity of Segments collected from Yahoo! production trace. X axis represents segments ordered in increasing order of creation time. Y-axis plots the number of accesses each segment saw in a 5 hour trace from Yahoo!.**

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.

In Druid, all internal coordination like segment loading between coordinator and HN is handled by a Zookeeper [26] cluster. Druid also uses MySQL [36] 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.

## 2.2 Workload Insights

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 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:

| 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 1: Druid traces from Yahoo! production clusters.**

**Segment Access is skewed, and recent segments are generally more popular:** Figure 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 [8], 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 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) [3].

**Some (older) segments continue to stay popular:** Figure 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 [49] – 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.

# 3 STATIC VERSION OF SEGMENT REPLICATION PROBLEM

We formally define the static problem (§3.1), and our solution (§3.2).

## 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) relaxes these assumptions.

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

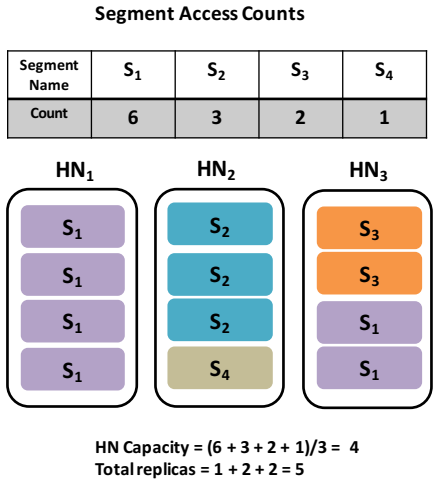



**Figure 5: Problem depicted with balls and bin. Query-segment pairs are balls and historical nodes represent bins. All balls of same color access the same segment. HN capacity refers to compute capacity. Optimal assignment shown.**

```
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 1: Generalized Allocation Algorithm.**

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 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 paper, "capacity" always means "compute capacity"). We wish to find: *Allocation* $X = \{x_{ij} = 1$, if segment $i$ 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 [47]. 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.

## 3.2 Solution

Algorithm 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 [44]. 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 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

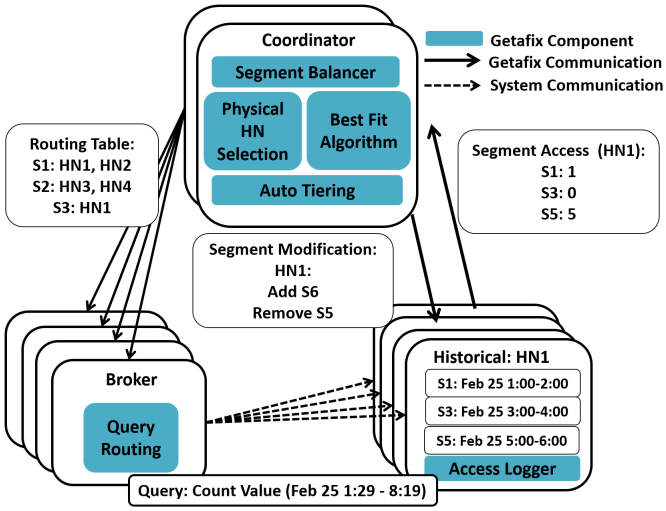



**Figure 6: Getafix Architecture.**

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 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.

We state our lemmas and optimality theorem here. We present the proofs in Appendix A.

Lemma 3.1. *Using ModifiedBestFit algorithm, no pair of HNs (bins) can have more than 1 segment (color) in common.*

A 2-way swap is an operation between two HNs (bins) that preserves load balance but swaps an equal number of query-segment pairs (i.e., balls), in an attempt to coalesce segment replicas (colors).

Lemma 3.2. *A k-way swap, involving k HNs, is equivalent to k 2-way swaps.*

Lemma 3.3. *No sequence of 2-way swaps, applied to the ModifiedBestFit algorithm's output, can further reduce the number of segment replicas (color splits).*

Theorem 3.4. *Given a set of queries, ModifiedBestFit minimizes both total number of segment replicas and makespan.*

# 4 GETAFIX: SYSTEM DESIGN

The Getafix system is intended to handle dynamically arriving segments as well as queries. Figure 6 shows the general architecture. Most of Getafix's logic resides in the Coordinator. The coordinator manages the segment replicas, runs the ModifiedBestFit algorithm (§4.1) 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 (§4.3) and handles heterogeneity in a deployed cluster (§4.4). We modified the broker code to implement different query routing strategies (§4.2).

## 4.1 Segment Replication Algorithm

For the dynamic scenario, Getafix leverages the static solution from §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.

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:

$$\text{Popularity}(s_j, K+1) = \frac{1}{2} \times \text{Popularity}(s_j, K) + \text{AccessTime}(s_j, K+1)$$

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 (§4.5 and §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.

## 4.2 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 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 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 2: Routing Table for Figure 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 [40], this scheme works well in our evaluations (§5.5) because of its small overhead.

## 4.3 Balancing Segment Load

For skewed segment access distributions (Figure 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 $\gamma$, 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 (§5.3), we found that a default $\gamma = 20\%$ gives the best segment balance with minimal impact on query load balance.

## 4.4 Handling Cluster Heterogeneity

ModifiedBestFit assumes a homogeneous cluster consisting of machines with equal compute capacity (§3.1). We now relax that assumption and present modifications for heterogeneous settings.

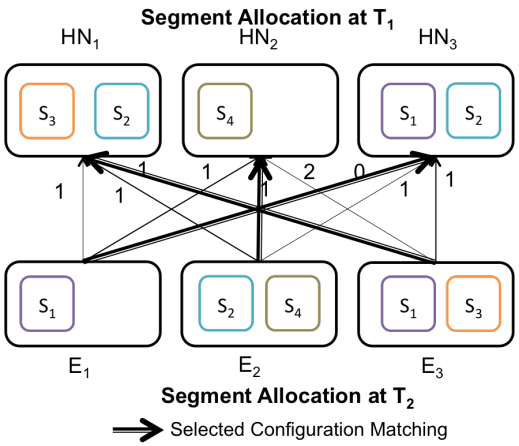



**Figure 7: Physical HN Mapping problem from Figure 5 represented as a bipartite graph.**

**Capacity-Aware ModifiedBestFit:** Instead of assuming equal capacity in Algorithm 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 [11].

**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.

## 4.5 Minimizing Network Transfers

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 [34]. We next discuss approaches that make Getafix network aware.

Consider the example shown in Figure 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 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 [48] 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.

## 4.6 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.

## 4.7 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 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 [44], 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.

## 4.8 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.

## 4.9 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.

# 5 EVALUATION

We evaluate Getafix on both a private cloud (Emulab [46]) and a public cloud (AWS [6]). 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%.

## 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 [27] containers (to constrain disk for GC experiment). We use d430 [17] 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 [11] instances (16 cores, 64 GB memory), S3 [12] as the deep storage, and Amazon EBS General Purpose SSD (gp2) volumes [7] 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 [8] 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).

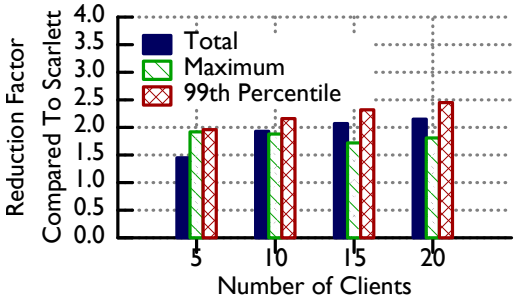
  In particular, we implemented Scarlett's round-robin based algorithm [4]. The round-robin algorithm counts the number of *concurrent accesses to a segment*, as an indicator of popularity. Scarlett gives more replicas to segments with more concurrent accesses. We collect the concurrent segment access 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 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 (§6). We also measure: 1) average and 99th percentile (tail) query latency and 2) makespan.

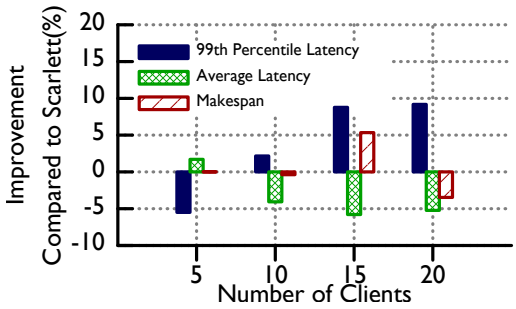
To calculate memory dollar cost savings in a public cloud, we multiply the memory savings with cost per GB of memory. We calculate the cost of 1 GB memory in a public cloud by solving a set of linear equations (elided for brevity) derived from the published instance type prices. For AWS, memory cost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

## 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 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 are added.




**(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.**

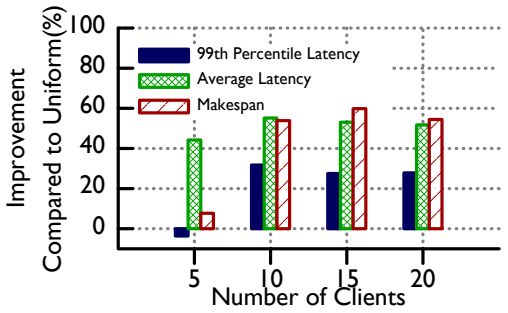
**Figure 8: AWS Experiments: Getafix vs. Scarlett with increasing load (number of client varying from 5 to 20).**

**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 1 implies the very popular segments influence effective replication factor.) In a public cloud deployment, where popular data size is 100 TB [5], Getafix thus can reduce memory usage by approximately 230 TB (100 TB × (4.2 - 1.9)). This amounts to cost savings of $230 \times 10^3$ GB × $0.005/GB/hour = $1150 per hour. Annually, this would amount to $10 million worth of savings.

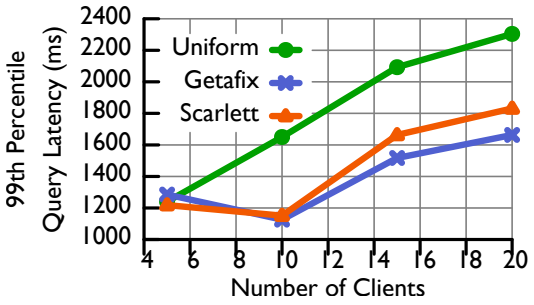
To quantify the impact of this memory savings on performance, Figure 8b plots the reduction in makespan, average and 99th percentile latency for Getafix compared to Scarlett. Getafix completes

[4] 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.

[5] Most present day production clusters in Google, Yahoo handle petabytes of data [23] 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.




**(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 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.**

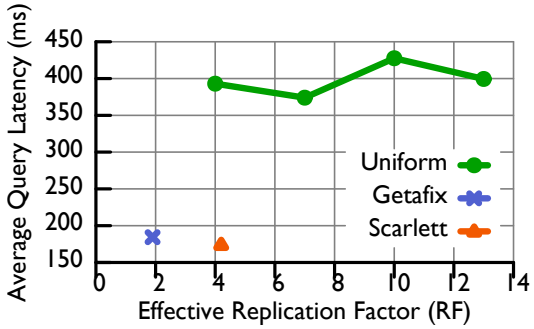



**Figure 10: AWS Experiments: Memory-Latency Tradeoff Curve. 20 HNs and 15 clients using: 1) Getafix, 2) Scarlett and 3) Uniform (RF: 4, 7, 10, 13). Lower is better on both the axes.**

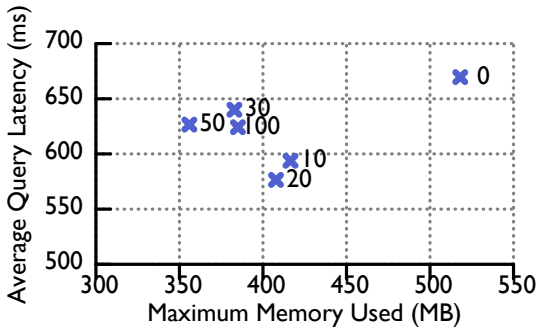



**Figure 11: Emulab Experiments: Getafix – Maximum memory vs. Query Latency Tradeoff for different $\gamma$ values. Lower is better on both axes.**

all the queries within ±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× less memory than Uniform. For latency and makespan, see Figure 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 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 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 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.

## 5.3 Segment Balancer Tradeoff

In §4.3, we introduced a threshold parameter $\gamma$ that determines the tradeoff space between maximum memory used and query performance. $\gamma$ = 0% implies no balancing while $\gamma$ = 100% implies aggressive balancing.

Figure 11 quantifies $\gamma$'s impact in a cluster of 20 HNs and 15 clients (labels are $\gamma$ values). As we increase $\gamma$, 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 $\gamma$, moving a few unpopular segments to such HNs allows the CPU to remain busy while the popular segment is falling in popularity. Too high $\gamma$ 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

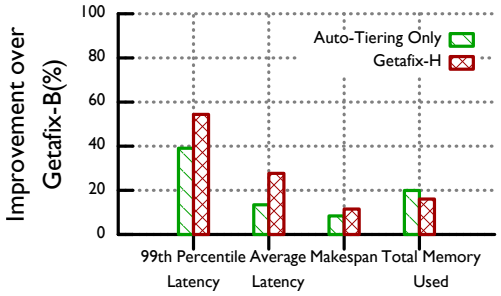



**Figure 12: Emulab Experiments: Improvement in 99th Percentile, Average Query latency, Makespan and Total Memory Used with Getafix-H compared to Getafix-B. Experiment performed with 2 HNs straggling among 20.**

faster completion of the queries, which in turn keeps segments in memory for lesser time.

## 5.4 Cluster Heterogeneity

We evaluate the performance of Capacity-Aware ModifiedBestFit (§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 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 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 3: AWS HN heterogeneous cluster configurations.**

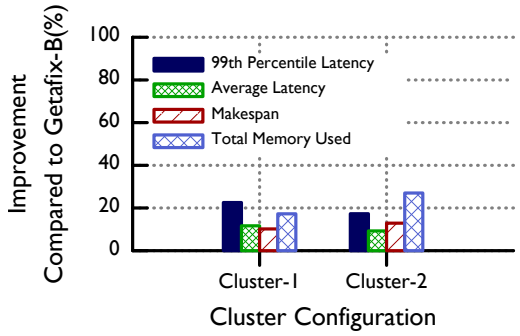



**Figure 13: AWS Experiments: Improvement in 99th Percentile, Average Query latency, Makespan and Total Memory Used with Getafix-H compared to Getafix-B. Experiments performed with 2 different node mixtures and clients (refer Table 3).**

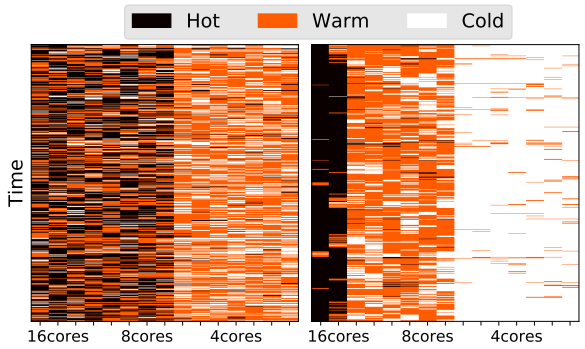



**Figure 14: AWS Experiments: Getafix-B on left, Getafix-H on right. Effectiveness of Auto-Tiering shown using heat map. X-axis represents HNs sorted by the number of cores they have. Y-axis plots a period of time in the duration of the experiments. For each time, we classify HNs as hot, warm and cold (represented with 3 different colors) based on the reported CPU time for processed queries.**

Figure 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 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 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

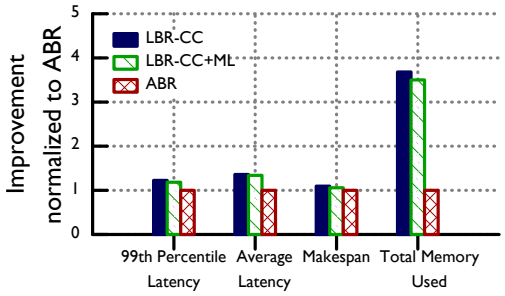



**Figure 15: Emulab Experiments: Comparing 3 different query routing strategies on Getafix-B – 1) LBR-CC, 2) LBR-CC+ML, 3) ABR. Higher is better.**

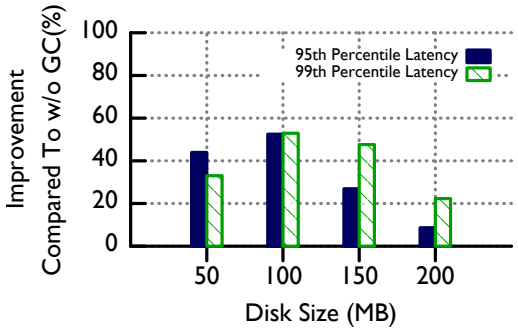



**Figure 16: Emulab Experiment: Improvement in 99th and 95th percentile query latency for Getafix with GC compared to without GC. Disk sizes are varied from 50 - 200 MB.**

each to Warm and Cold tiers (Table 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).

## 5.5 Comparing Query Routing Schemes

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

## 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 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.

# 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 (~ 2× 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.

## 7 RELATED WORK

**Allocation Problem:** Our problem has similarities to the *data allocation problem* [37] in databases which tries to optimize for performance [39, 50] and/or network bandwidth [10]. A generalized version of the problem has been shown to be NP-hard [37]. Typical heuristics used are best fit and first fit [13, 25] or evolutionary algorithms [39]. 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 [22] 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 [16], whose techniques minimize cross-partition transactions in graph databases. There are other works which look at adaptive partitioning for OLTP systems [38] and NoSQL databases [15] respectively, however they do not explore Druid-like interactive analytics engines. E-Store [45] 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 [35] 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 [29], 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 [3, 14, 18, 32] 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 [43]. Since Getafix uses data access times and not query semantics, it can reduce memory usage generally.

Amazon Athena [5] 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 [51], Pinot [31], Redshift [4], Mesa [23], 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 [1, 9, 19, 52]. 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.

## 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.

## ACKNOWLEDGMENTS

This work was supported in part by: NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, and a generous gift from Microsoft. We thank our shepherd Dushyanth Narayanan and the anonymous reviewers for their invaluable input. We thank U. Utah for Emulab support.

## A OPTIMALITY PROOF

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

### A.1 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 §3.1. We claim that ModifiedBestFit minimizes the cost for a load balanced assignment of balls in bins.

## A.2 Proofs

Lemma A.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 A.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 A.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 A.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:

**Case 1:** $n_1 > n_2$**:** 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.

**Case 2:** $n_1 < n_2$**:** 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.

**Case 3:** $n_1 = n_2$**:** 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 A.2), no swap strategy can further reduce the number of segment replicas, computed by ModifiedBestFit.

Theorem A.4. *Given a set of queries, ModifiedBestFit minimizes both total number of segment replicas and makespan.*

Proof. By Lemma A.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. □

## REFERENCES


[1] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. 2012. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 61–74. https://doi.org/10.1145/2150976.2150984

[2] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. 2005. Distributed Operation in the Borealis Stream Processing Engine. In *Proceedings of the 2005 ACM International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 882–884. https://doi.org/10.1145/1066157.1066274

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. 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* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[4] Amazon. 2012. Redshift. (2012). Retrieved February 28, 2018 from https://aws.amazon.com/redshift/

[5] Amazon. 2018. Athena. (2018). Retrieved February 28, 2018 from https://aws.amazon.com/athena/

[6] Amazon. 2018. AWS. (2018). Retrieved February 28, 2018 from https://aws.amazon.com/

[7] Amazon. 2018. EBS. (2018). Retrieved February 28, 2018 from https://aws.amazon.com/ebs/pricing/

[8] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. 2011. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/1966445.1966472

[9] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=1924943.1924962

[10] Peter M. G. Apers. 1988. Data Allocation in Distributed Database Systems. *ACM Transactions on Database Systems* 13, 3 (Sept. 1988), 263–304. https://doi.org/10.1145/44498.45063

[11] Amazon Web Services (AWS). 2018. Instance Types. (2018). Retrieved February 28, 2018 from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

[12] Amazon Web Services (AWS). 2018. S3. (2018). Retrieved February 28, 2018 from https://aws.amazon.com/s3/

[13] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious Data Placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, USA, 139–149. https://doi.org/10.1145/291069.291036

[14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment* 8, 4 (Dec. 2014), 401–412. https://doi.org/10.14778/2735496.2735503

[15] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. 2013. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/2465351.2465370

[16] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[17] Emulab. 2018. d430. (2018). Retrieved February 28, 2018 from https://wiki.emulab.net/wiki/d430

[18] Facebook. 2013. PrestoDB. (2013). Retrieved February 28, 2018 from https://prestodb.io/

[19] Zacharia Fadika, Elif Dede, Jessica Hartog, and Madhusudhan Govindaraju. 2012. MARLA: MapReduce for Heterogeneous Clusters. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '12)*. IEEE Computer Society, Washington, DC, USA, 49–56. https://doi.org/10.1109/CCGrid.2012.135

[20] The Apache Software Foundation. 2014. Hadoop. (2014). Retrieved February 28, 2018 from https://hadoop.apache.org

[21] The Apache Software Foundation. 2015. Storm. (2015). Retrieved February 28, 2018 from https://storm.apache.org

[22] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, Berkeley, CA, USA, 75–88. http://dl.acm.org/citation.cfm?id=1924943.1924949

[23] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. 2016. Mesa: A Geo-replicated Online Data Warehouse for Google's Advertising System. *Commun. ACM* 59, 7 (June 2016), 117–125. https://doi.org/10.1145/2936722

[24] Himanshu Gupta. 2016. Beyond Hadoop at Yahoo!: Interactive analytics with Druid. Talk. (28 September 2016). Retrieved February 28, 2018 from https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51640

[25] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 13, 17 pages. https://doi.org/10.1145/2523616.2525970

[26] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[27] Docker Inc. 2018. Docker. (2018). Retrieved February 28, 2018 from https://www.docker.com/

[28] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[29] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2016. BlowFish: Dynamic Storage-performance Tradeoff in Data Stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 485–500. http://dl.acm.org/citation.cfm?id=2930611.2930643

[30] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[31] LinkedIn. 2015. Pinot. (2015). Retrieved February 28, 2018 from https://github.com/linkedin/pinot/wiki

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[33] Metamarkets. 2018. Powered by Druid. (2018). Retrieved February 28, 2018 from http://druid.io/druid-powered.html

[34] Microsoft. 2018. Blob Storage. (2018). Retrieved February 28, 2018 from https://azure.microsoft.com/en-us/services/storage/blobs/

[35] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Berkeley, CA, USA, 383–398. http://dl.acm.org/citation.cfm?id=2685048.2685078

[36] Oracle. 2018. MySQL. (2018). Retrieved February 28, 2018 from https://www.mysql.com

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

[38] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[39] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 315–330. https://doi.org/10.1145/3035918.3064052

[40] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient Queue Management for Cluster Scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/2901318.2901354

[41] Amazon Redshift. 2018. Customer Success. (2018). Retrieved February 28, 2018 from https://aws.amazon.com/redshift/customer-success/

[42] Research and Markets. 2015. Streaming Analytics Market by Verticals - Worldwide Market Forecast & Analysis (2015 - 2020). Report. (June 2015). Retrieved February 28, 2018 from https://www.researchandmarkets.com/research/mpltnp/streaming

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer

Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[44] William Stallings. 2005. *Operating Systems: Internals and Design Principles Edition: 5.* Pearson.

[45] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[46] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*. USENIX Association, Boston, MA, 255–270.

[47] Wikipedia. 2018. Bin Packing Problem. (2018). Retrieved February 28, 2018 from https://en.wikipedia.org/wiki/Bin_packing_problem

[48] Wikipedia. 2018. Hungarian Algorithm. (2018). Retrieved February 28, 2018 from http://en.wikipedia.org/wiki/Hungarian_algorithm

[49] Wikipedia. 2018. Jaccard index. (2018). Retrieved February 28, 2018 from https://en.wikipedia.org/wiki/Jaccard_index

[50] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. 1997. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems* 22, 2 (June 1997), 255–314. https://doi.org/10.1145/249978.249982

[51] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A Real-time Analytical Data Store. In *Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/2588555.2595631

[52] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, Berkeley, CA, USA, 29–42. http://dl.acm.org/citation.cfm?id=1855741.1855744