Getafix: Workload-aware Distributed Interactive Analytics

Mainak Ghosh, Le Xu, Xiaoyao Qian, Thomas Kao, Indranil Gupta
University of Illinois, Urbana Champaign
{mghosh4, lexu1, qian13, tkao4, indy}@illinois.edu

Himanshu Gupta
Yahoo! Inc
himanshg@yahoo-inc.com

Abstract

Distributed interactive analytics engines (Druid, Redshift, Pinot) need to achieve low query latency while using the least storage space. This paper presents a solution to the problem of replication of data blocks and routing of queries. Our techniques decide the replication level of individual data blocks (based on popularity, access counts), as well as output optimal placement patterns for such data blocks. For the static version of the problem (given set of queries accessing some segments), our techniques are provably optimal in both storage and query latency. For the dynamic version of the problem, we build a system called Getafix that dynamically tracks data block popularity, adjusts replication levels, dynamically routes queries, and garbage collects less useful data blocks. We implemented Getafix into Druid, the most popular open-source interactive analytics engine. Our experiments use both synthetic traces and production traces from Yahoo! Inc.'s production Druid cluster. Compared to existing techniques Getafix either improves storage space used by up to 3.5× while achieving comparable query latency, or improves query latency by up to 60% while using comparable storage.

1. INTRODUCTION

Real-time analytics is projected to continue growing annually at a rate of 31% [13]. Apart from stream processing engines, which have received much attention [4, 5, 6, 22, 34], real-time analytics now also includes the burgeoning area of interactive data analytics engines such as Metamarkets' Druid [45] (used by Yahoo! Inc.), Amazon's Redshift [1], and LinkedIn's Pinot [17]. These systems have been widely adopted [2, 20] in companies like Yahoo! [21], LinkedIn [16], and Pinterest [19], etc. Yahoo! internally uses Druid for 35 applications spanning usage analytics, revenue reporting, spam analytics, ad feedback, and Flurry [11] SDK reporting. Yahoo!'s deployment covers more than 2000 hosts, and each application indexes 75 Billion events per day, and it processes 2.5 Million queries per day [7].

In such interactive data analytics engines, data is ingested from many data analytics pipelines including batch and streaming sources, then it is indexed and stored in a data warehouse. This data is immutable. The data warehouse resides in a backend tier, e.g., HDFS [38] or Amazon S3 [3]. In parallel, queries arrive and are run on multiple compute nodes that reside in a frontend tier (cluster). These compute nodes have several orders of magnitude less storage space (Terabytes) compared to the warehouse (Exabytes). Thus, the compute nodes need to access data from the warehouse in such a way so as to: i) minimize query execution time, and ii) maximize storage utilization.

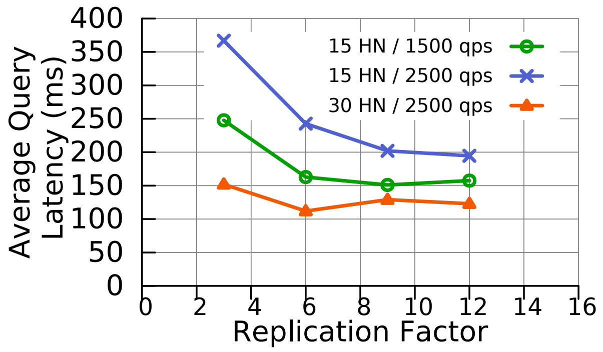


Figure 1: **Varying replication factor as number of compute node and query injection rate changes. 'HN' stands for the number of compute nodes, and 'qps' stands for injected operations (queries) per sec.**

To achieve high performance, these systems exploit parallelism at query execution level, i.e., for a query that accesses multiple *segments* (data blocks), it is run in parallel on each segment, and then the results are collected and aggregated. Segments need to be placed carefully, e.g., two "popular" segments accessed by many queries should not be colocated. Segments can be replicated to increase the number of choices in assigning queries to nodes. However, full replication is prohibitive because of limited storage space in the frontend tier and the high network bandwidth incurred.

Today's systems use simple strategies for managing data at the compute nodes which include: 1) uniformly replicating all segments and 2) a system administrator manually creating storage tiers with different replication factors and then assigning segments based on his/her estimate of popularity. The former approach is sub-optimal because some data is more popular than others [24]. For instance, we analyzed traces from Yahoo!'s Druid cluster and we found that top 1% of data is an order of magnitude more popular than the bottom 40% (see Figure 2a). Uniform replication constrains the benefits of parallelism for such data. The tier-based replication scheme (approach (2)) is laborious, and cannot adapt in real time to changes in query patterns and/or cluster configuration.

The right amount of replication needed to achieve good performance depends on the query injection rate and the cluster size. Figure 1 illustrates this via a few combinations of number of compute nodes (HNs) and query rates (qps). The "knee" of a given line determines the minimum replication level that achieves almost low latency as full replication. With 15 compute nodes, the knee of the curve occurs at around 6 replicas for 1500 queries per sec, but the knee becomes higher (9 replicas) when the input rate increases to 2500 queries per sec. The knee also falls with an increase in the number of compute nodes from 15 to 30.

Some techniques in this class of systems replicate based on data popularity, but they approximate popularity via recency of data [45]. That is, they assume that most queries will touch data that was ingested recently and is only a few hours to days old. However, our analysis of Yahoo!'s Druid cluster traces across multiple days shows that even older data can be popular, either transiently or for long periods. For instance, Figure 3 shows 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. Equating recency may result in a popular old data becoming colocated with a recent data, overloading that compute node with many queries and prolonging query completion times. Other techniques like Scarlett [24] replicate solely based on query injection rate. This can lead to good query performance but incurs large storage costs.

We present a new system called Getafix [1] to address these challenges. The key is to use query injection rate, current cluster capacity, and measured popularity of data, to dynamically decide replication factor for each data block in the system. We start by addressing the static version of the problem, where we are given a fixed set of queries and the segments each needs to access. We present an algorithm for placing segments and scheduling queries at the compute nodes, and show that this algorithm achieves optimality in *both* run time (of the query set) as well as storage utilization (number of segment replicas across all nodes).

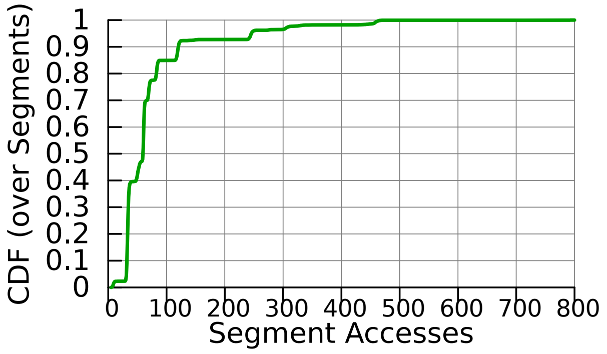
Next we develop techniques to solve the dynamic version of the problem wherein data and queries are streaming in continuously. Our dynamic solution, implemented in our Getafix system, combines a segment popularity metric based on segment access counts, along with a best fit strategy for loading segments. It contains techniques for loading segments as well as for routing incoming queries. Getafix also minimizes the volume of network transfer required for loading data from the backend warehouse by using a bipartite matching approach.

We implemented Getafix and integrated it into Druid [45], which is one of the most popular open-source interactive data analytics engines in use today. We present experimental results comparing Getafix to the closest systems: 1) base Druid system with fixed replication strategies, and 2) Scarlett [24], which solves a similar problem but solely for batch processing systems like Hadoop [14], Dryad [31], etc. Getafix reduces storage usage by up to 3.5× compared to Scarlett [24], while still providing comparable query latency performance. Getafix improves median query latency by 60% compared to uniform replication when using similar amount of storage.

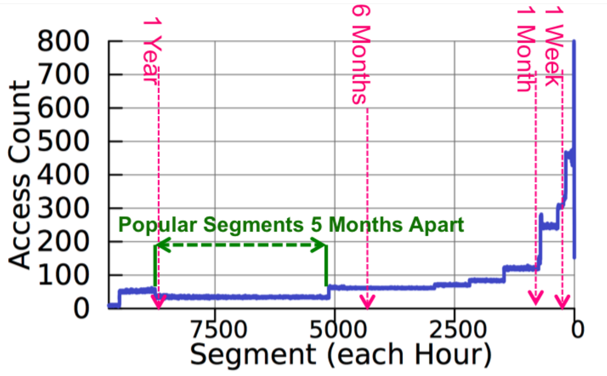
In summary, our main contributions are:

- We propose new algorithms and techniques for data management in interactive data analytics engines.
- In the static version of the problem, our algorithm provably minimizes both storage requirement as well as query running time (§3).

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



(a) **CDF of Segment Popularity.**



(b) **Segment Popularity In the Past.**

Figure 2: **Yahoo! Workload Analysis.**

- We solve the dynamic variant of the segment placement and query routing problem (§4).
- We design and implement our system Getafix into Druid, a popular interactive data analytics engine that is open-source (§4).
- We evaluate Getafix using both real-world workloads from Yahoo! (§5) and synthetic traces.

2. BACKGROUND

2.1 System Model

We present a general architecture of an interactive data analytics engine. To be concrete, we borrow our terminology from a popular system in this space, Druid [45].

An interactive data analytics engine receives results 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, also identified by the time interval it was collected in, 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 this term in the rest of the paper.

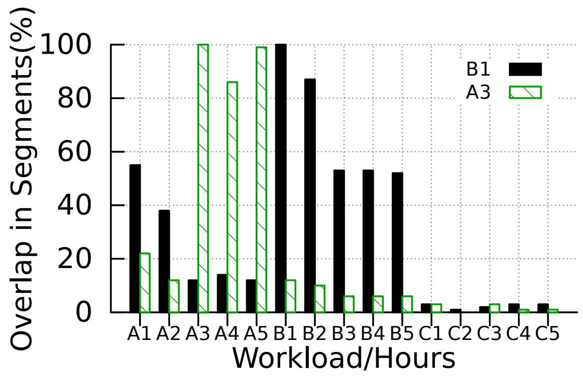


Figure 3: **Yahoo! Workload: Overlap in segment accesses across different hours of 3 workloads. Each workload identified with the workload name (A/B/C: see Table 1) and the ith hour.**

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 replicas of a segment. Once loaded, the HNs can start serving queries which access this segment.

Queries are sent 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 compute nodes in parallel. The broker then 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 [30] cluster. Druid also uses MySQL [18] for storing metadata from segments and failure recovery. This makes the broker, coordinator, and historical nodes, all stateless. This enables fast recovery by spinning up a new machine.

2.2 Workload Insights

Name	Period	Total Segments	Total Accesses
A	5 months old	0.6K	65K
B	1 month old	9.3K	0.8M
C	1 week old	1.3K	64K

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

We analyze Yahoo!'s Druid cluster workloads spanning several hundreds of machines, and many months of segments (segments are hourly). We use three diverse workload traces (shown in Table 1). We draw two useful observations that will feed into our design decisions:

Segment Access is skewed, and recent segments are generally more popular: Figure 2a plots the CDF of the access counts for workload B (other workloads 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 [24], 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 2b shows the number of times a segment was accessed in the workload trace B. That is, the 4000th data point shows the total access count for the segment created 4000 hours before this workload was created. We observe that segments are most popular 3 to 8 hours after creation, and this popularity is about 2 $\times$ more than segments which are a week old. However, a few select very old segments (e.g., bumps at about a year ago) continue to stay popular. This is usually due to interesting events such as Thanksgiving or holiday weeks.

Some (older) segments always stay popular: Figure 3 shows the level of overlap between a segment accessed during an hour of the Yahoo! workload (shown on the horizontal axis), and a reference hour (B1 or A3). Here, "overlap" is defined as 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 workloads spread across 5 months suggests that there must be relatively old segments that are being accessed again. This confirms again that some select old segments may be popular (for a while) far in the future 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 PLACEMENT PROBLEM

We formally define the static problem (§3.1), our solution (§3.2), and give a proof of optimality (§3.3).

3.1 Formulation: Static Version of Problem

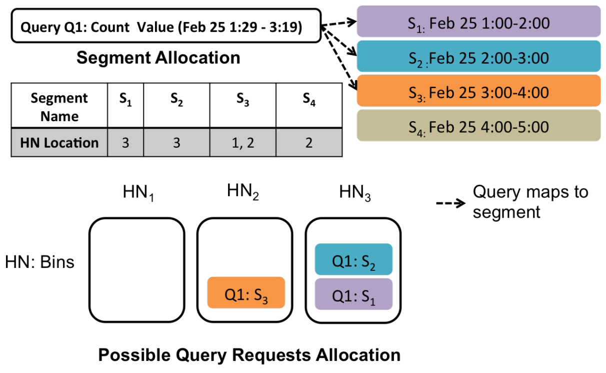
Given m segments, n historical nodes (HNs), and k queries that access a subset of these segments, our goal is to find a segment allocation (segment assignment to HN(s)) that both: 1) ensures query load balance, and 2) minimizes the total replication required. For simplicity we assume: a) each query takes unit time to process each segment it accesses, and b) the HNs are all empty. Our implementation (later sections) relaxes these assumptions.

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 compute nodes automatically gives a time-optimal schedule; this is because it is load balanced across all HNs, and 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 completion* time 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 was 7. The minimum number of replicas for this example however 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$\}$.

Formally, the input to our 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 $\lceil \frac{\sum_i C}{n} \rceil$. We wish to find:

Allocation $X = \{x_{ij} = 1, \text{if segment } i \text{ is replicated at HN } j\}$, such that it minimizes $\sum_i \sum_j x_{ij}$.



Segment Name	S_1	S_2	S_3	S_4
HN Location	3	3	1, 2	2

Figure 4: **Query Q1, Current Segment Allocation, and one possible routing strategy for Q1.**

This problem has similarities to the bin packing problem [8]. A segment-query 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, our version of the problem is solvable in polynomial time.

Figure 4 shows an example of a query Q1 that requests segments S_1, S_2 and S_3. Based on the segment allocation shown in the table, one possible query routing strategy is to route the requests from Q1, to both HN_2 and HN_3. However, this needs to be done in a way that takes other queries and their requirements into account.

3.2 Solution

We first present a generalized template algorithm (Algorithm 1). This algorithm can be instantiated with any of three heuristics that we describe next, via the ChooseHistoricalNode routine.

Algorithm 1 maintains a priority queue of segments, sorted in decreasing order of popularity (i.e., number of queries accessing the segment). The algorithm iteratively extracts the next segment S_i from the head of the list, and allocates the segment-query pairs which access it to a HN (selected based on a heuristic). If the selected HN's current capacity is insufficient to take all these pairs, then: a) all available slots in that HN are filled with the segment-query pairs for that segment, and b) the segment's count is updated to reflect remaining segment-query pairs, and it is re-inserted back into the priority queue at the appropriate position (via binary search).

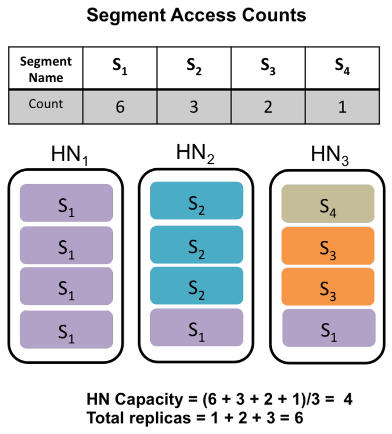
The total number of iterations in this algorithm equals the total number of replicas created across the cluster. The algorithm thus takes time $O((\sum_{i=1}^{m} c_i) \cdot log(m))$, i.e., it is linear in the number of query-segment pairs. This bound is loose and the actual typical performance is much better.

There are three options for the ChooseHistoricalNode routine, all inspired by segmentation strategies from traditional operating systems [40]:

First Fit: We choose the next HN (lowest HN id) that is not yet full, i.e., $argmin_{HN_*}\ binCap[HN_i] \le capacity$. The intuition here is to fill up bins in a FIFO way and reduce external fragmentation [12]. If the available HN capacity is insufficient, we fill up that HN as much as we can, and return the segment with a reduced query count back to the queue.

```
input: $C$: Access counts for each segment
       $nodelist$: List of HNs
Algorithm ModifiedFit($C, nodelist$)
    $n \leftarrow$ Length($nodelist$)
    $capacity \leftarrow \lceil \frac{\sum_{C_i \in C} |C_i|}{n} \rceil$
    $binCap \leftarrow$ Allocate($n, capacity$)
    $priorityQueue \leftarrow$ 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($priorityQueue, (segment, left)$)
        end
    end
```

Algorithm 1: Generalized Allocation Algorithm.



Segment Access Counts
HN1
HN2
HN3
S1
S2
S4
S3
HN Capacity = (6 + 3 + 2 + 1)/3 = 4
Total replicas = 1 + 2 + 3 = 6

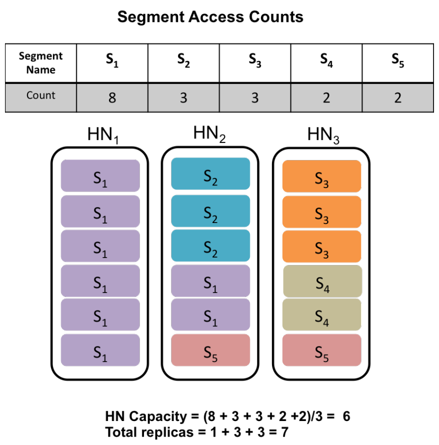
Segment Name	S_1	S_2	S_3	S_4
Count	6	3	2	1

Figure 5: **An example execution of First Fit and expected final configuration.**

Figure 5 considers our running example where C is $\{S_1$:6, S_2:3, S_3:2, S_4:1$\}$. First fit picks segment S_1 and places it in HN_1 as it is the first bin with available slots, returning the remaining 2 queries for S_1 to the queue. Then, it picks segment S_2 and assigns it to HN_2. At this point, HN_2 is not full. The next segment in the queue S_1 (count 2, tie with S_2 broken via lower segment id) is assigned to fill up HN_2. Since segment S_1 has count of 2 but HN_2 has only one slot, S_1 is re-inserted into the queue with a count of 1. Continuing this way, the final state reached is shown in the figure. This strategy may be sub-optimal in replication count, e.g., the above example creates 6 total replicas, but we showed earlier in §3.1 that the optimal was 5 replicas. So First Fit is not optimal.

Largest Fit: We choose the HN with the largest available capacity leftover, i.e., $argmax_{HN_*}(binCap)$. The intuition is to create large enough *holes* so that segments picked in later iterations with smaller counts (fewer queries needing them) can fully fit in them. If the HN's capacity is insufficient in an iteration, we fill up the HN and return the segment with reduced count back to the queue.

Figure 6 shows a different example and the final allocation due to Largest Fit (ties broken via lower HN and segment ids). This


Segment Access Counts

Segment Name	S_1	S_2	S_3	S_4	S_5
Count	8	3	3	2	2

HN Capacity = (8 + 3 + 3 + 2 +2)/3 = 6
Total replicas = 1 + 3 + 3 = 7

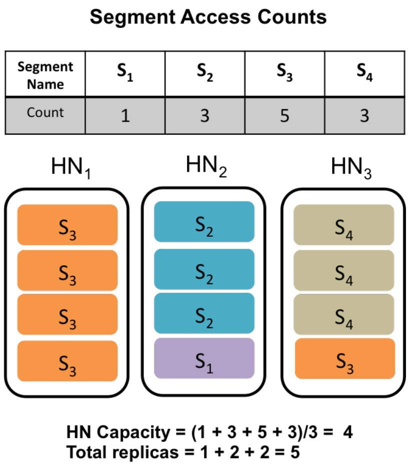
Figure 6: **An example execution of Largest Fit and expected final configuration.**

creates 7 replicas. However the optimal is 6 replicas and an optimal allocation is: $HN_1 = \{S_1{:}6\}$, $HN_2 = \{S_2{:}3, S_3{:}3\}$, $HN_3 = \{S_4{:}2, S_1{:}2, S_5{:}2\}$. Thus Largest Fit is not optimal either.

Best Fit: We choose, in each iteration, the next HN which would have the least slots remaining after accommodating all the queries in the current segment, i.e., $argmin_{HN_*} max\{0, (binCap[HN_i]$ - number of queries for this segment$)\}$, with ties broken by lower HN id. The intuition is to pack the number of queries in the slots more effectively. If none of the nodes have sufficient capacity (for the segment at the queue head), we default to Largest Fit for this iteration, i.e., we choose the HN with the largest available capacity (ties broken by lower HN id), fill it as much as possible, and return queries (if any, with updated counts) to the queue.

Figure 7 shows an example execution for Best Fit. The reader can verify that the final assignment is correct (ties broken via lower HN and segment ids). This allocation in fact reaches the minimum number of replicas.

Next we formally prove that the MODIFIEDBESTFIT (Algorithm 1 using Best Fit for CHOOSEHISTORICALNODE) strategy in fact is optimal in the amount of replication.


Segment Access Counts

Segment Name	S_1	S_2	S_3	S_4
Count	1	3	5	3

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

Figure 7: **An example execution of MODIFIEDBESTFIT and expected final configuration.**

3.3 Optimality Proof

We now formally prove that MODIFIEDBESTFIT minimizes the amount of replication among all load balanced assignments. (This section can be skipped without loss of understanding the rest of the paper.)

3.3.1 *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 ($p \le m$) 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.

3.3.2 *Proofs*

LEMMA 3.1. *In a balls and bins arrangement using* MODIFIEDBESTFIT *algorithm, no pair of bins can have more than 1 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, lets define an important operation called *swap*, which will be used later in our proof:

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 3.2. *A $k-way$ swap 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. □

We now define an important term that improves any assignment:

Successful swap: This is 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.

LEMMA 3.3. *No successful swap is possible for the* MODIFIEDBESTFIT *algorithm.*

PROOF. Since a $k-way$ swap is equivalent to k $2-way$ swaps (Lemma 3.2), we prove the theorem by showing that there is no successful $2 - way$ swap. For the rest of proof, we use the term "swap" to denote only a $2 - way$ swap.

We prove this by contradiction. Lets say a successful swap is possible. From Lemma 3.1, 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 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 1:** $n_1 < n_2$**:** 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 existence 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, again 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 (in color red). This cannot be a successful swap.

□

THEOREM 3.4. MODIFIEDBESTFIT *minimizes the amount of replication as well as achieves the lowest completion time for a query set (maximizes throughput).*

PROOF. By Lemma 3.3, MODIFIEDBESTFIT generates load balanced allocation that minimizes the sum of unique color balls across all bins, which in turn minimizes replication. Load balanced allocation of query-segment pairs implies the completion time is minimized. □

4. GETAFIX: SYSTEM DESIGN

In this section, we discuss the design of Getafix. First, we give an overview of how we use MODIFIEDBESTFIT for replication, placement and routing (§4.1). Then we present the key design and implementation decisions of Getafix.

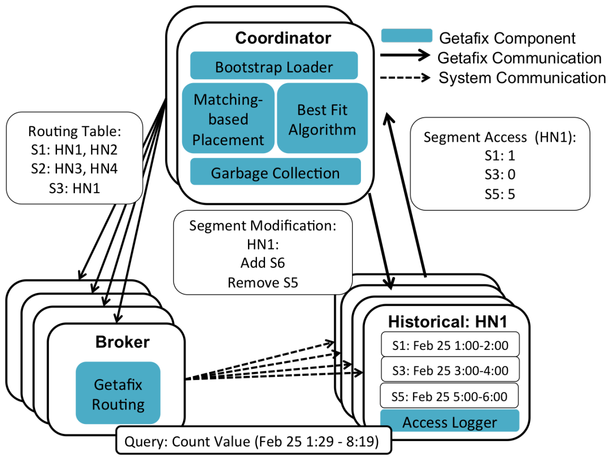


Figure 8: **Getafix Architecture.**

4.1 Overview

The static version assumes complete knowledge of all segments and queries up front. In reality, both segments and queries are dynamically streaming in all the time. Our approach divides the execution time into small windows and incrementally computes the amount of replication required for each segment. This approach works because as discussed in Section §2.2 segment popularity persists for a while.

Figure 8 shows the general architecture of our Getafix system. The historical nodes (HNs) are tasked with collecting the total number of segment accesses. The coordinator periodically (every minute) queries the HNs for the segment access information. It aggregates the responses from multiple HNs. A HN resets its counter after sending out this information. The coordinator runs MODIFIEDBESTFIT to calculate the segment popularity. Segment popularity is aged exponentially as: $\sum_{i=1}^{K}(C_{ij} \times \frac{1}{2^{i-1})}$ where C_{ij} is the actual access count for segment s_j in window i.

At the end of a run, the output of MODIFIEDBESTFIT is used for both segment placement and query routing.

Segment Placement: To ensure query load balance, segments need to be colocated according to the results of MODIFIEDBESTFIT. In our running example (Figure 7), S_3 should occupy one HN, S_3 and S_4 should be stored in another node, and finally, S_1 and S_2

should occupy the last node. After receiving this information the coordinator sets up the HNs to load or remove respective segments.

Query Routing: Apart from colocating segments, HNs should also serve the right number of queries, so that the number of segment requests routed to the HN matches the segment-query pairs on that HN. In our example (Figure 7), segment S_3 has 5 query-server pairs of which 4 are in HN_1 and 1 is in HN_2. This means the replica at HN_1 serves 80% of queries accessing while the remaining 20% is served from HN_2. The resulting routing table looks like:

S_1	0	0	100
S_2	0	0	100
S_3	80	20	0
S_4	0	100	0

Table 2: **Routing Table for Figure 7.**

The broker nodes periodically poll the coordinator for the routing table. In our implementation, this period is 20 seconds. We use a shorter duration than the frequency at which coordinator runs to ensure routing table at broker end is not stale. The broker also maintains a view of segment assignments. So, it can detect when the routing table is inconsistent. At this point, it routes queries by randomly picking a HN which has the segment based on its own view. If collisions are detected that cause hostpots, the frequency of polling is increased.

4.2 Segment Loading

To serve queries right away, when a new segment is created (at a realtime node), Getafix immediately and eagerly replicates once at a random HN, independent of whether some queries are requesting to access it. Later, our replication may create more replicas (depending on segment popularity). This is preferable to letting the real time nodes handle queries for fresh segments (the approach used in today's Druid system), which overloads the real time node. This early bootstrapping also allows segment count calculation to start early.

Getafix retries a query if it failed because the segment was not loaded into a HN. 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.

4.3 Matching-based Placement

To optimize the network transfer volume of placing segments in MODIFIEDBESTFIT, Getafix models this as a *configuration assignment* problem. This problem is treated as a bipartite graph shown in Figure 9 where vertices on the left side represent expected configurations (E_i) and vertices on the right represent HNs (HN_i) with a current set of replicas. An edge represents an assignment and the cost of the assignment is calculated as the number of data transfers required for the HN (HN_i) to have all the segments in expected configuration (E_i).

Minimizing data transferred in this graph can be achieved by finding the minimum cost bipartite matching. In the example in Figure 9, the final assignment is shown with thick lines. $E_1 \Rightarrow HN_1$, $E_2 \Rightarrow HN_3$, $E_3 \Rightarrow HN_2$. The total number of network transfers required is 2. As opposed to this, a naive allocation where HN HN_i chooses to have segments in the expected configuration E_i, would have required 3 data transfers (segment S_4 and S_3 to HN_2, and segments S_2 to HN_3).

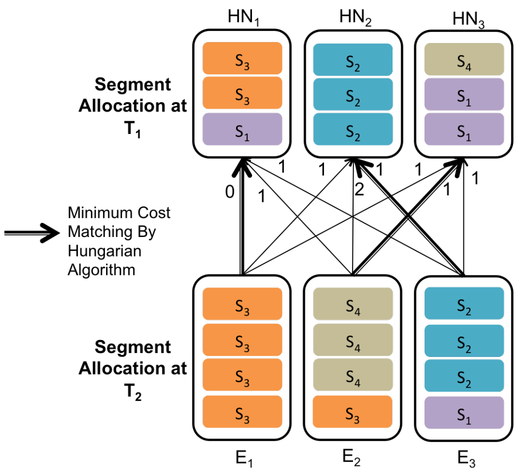


Figure 9: **Configuration Assignment problem from Figure 7 represented as a bipartite graph.**

Getafix's replication policy constructs a bipartite graph (similar to Figure 9) by comparing the results from MODIFIEDBESTFIT with the current segment assignments to HNs. We use the classical Hungarian Algorithm [15] to find the minimum matching. The coordinator uses the results to set up data transfers for the segments to their appropriate HNs.

4.4 Lazy Deletion and Garbage Collection

MODIFIEDBESTFIT also tells us which replicas are no longer required. This is because incoming queries are no longer accessing the segments, or the segment counts have dropped. For instance, in Figure 9, segment S_1 is not needed in HN_1 and HN_3 after configuration change.

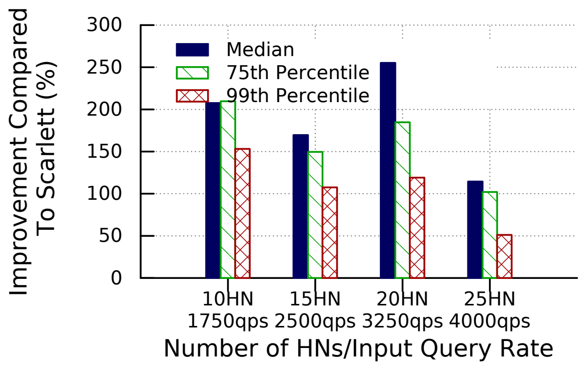
However we do not delete such segments eagerly, instead deferring their deletion. We implement lazy deletion by rate-limiting how many replicas of a segment are deleted during each MODIFIEDBESTFIT run: currently it is set to 1. This lazy approach retains segments in case their popularity increases again, and avoids network IO. Natural fluctuation in segment accesses causes popularity to fluctuate, and lazy deletion is tolerant to making hasty decisions.

To limit storage utilization, we implemented our own garbage collector for segments in the coordinator. The garbage collector evicts unused segments from the frontend tier. The coordinator periodically checks the storage pressure and if this exceeds a threshold, it marks unpopular segments for removal until the pressure is alleviated. Typically, the threshold is set as a fraction of the total storage available, currently set to 80%. Since we already maintain segment popularity to run MODIFIEDBESTFIT, we reuse this information to choose which segments to remove.

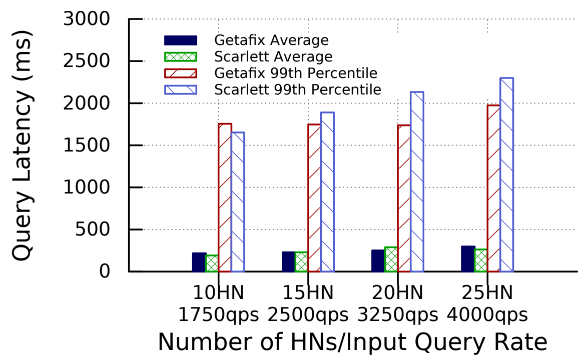
4.5 Fault Tolerance

Entities like the broker, HN, and coordinator are stateless and after a failure can be spun up within minutes. Metrics like segment counts are kept in HNs by Getafix, but also fetched by the coordinator and translated into routing tables at the brokers. After a HN failure that loses some of these segment counts, Getafix can continue routing with slightly stale data until segment counts catch back up. This typically happens within minutes.

To avoid the effect of losing segment counts on the coordinator we periodically checkpoint it to a MySQL table every 1 minute.

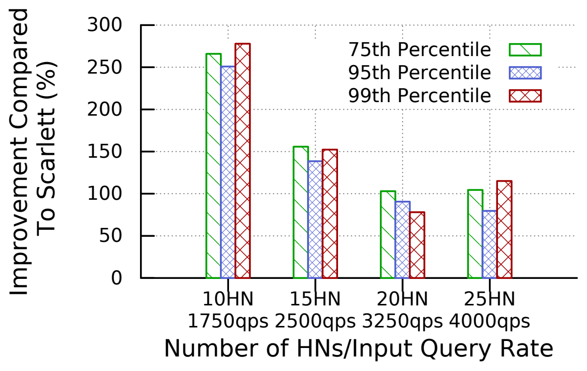


(a) **Improvement in Storage Space compared to Scarlett.**

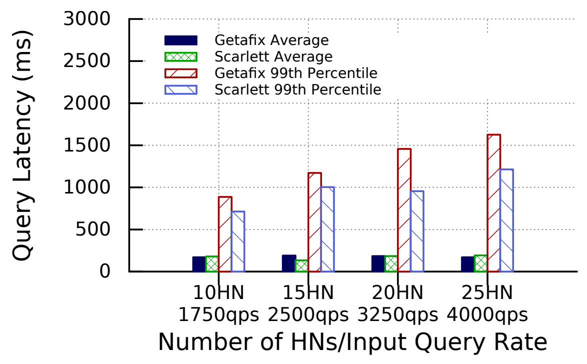


(b) **Query Latency: Average and Tail.**

Figure 10: **Comparison of Getafix vs. Scarlett under Synthetic Workload.**



(a) **Improvement in Storage Space compared to Scarlett.**



(b) **Query Latency: Average and Tail.**

Figure 11: **Comparison of Getafix vs. Scarlett under Yahoo! Production Workload.**

We do a full update instead of an incremental update as MySQL is optimized for bulk writes.

5. EVALUATION

We evaluate Getafix using deployments on a 50 machine cluster. Our experiments are based on both synthetic data, and workload traces from the Druid production cluster at Yahoo!. We summarize our results here:

- Compared to the best existing strategy (Scarlett), Getafix reduces median and 99th percentile storage space by 2× - 3.5×, while producing comparable query latency, across both synthetic and Yahoo! workloads, and in both batch and streaming settings.
- Compared to fixed replication (a common strategy used today in Druid) using similar amount of storage, Getafix improves median query latency by 20% - 60%.
- Getafix's garbage collector improves 95th percentile query performance by 25% - 55%.
- Getafix's routing strategy improves tail query latency by 35% compared to a random scheme.

5.1 Methodology

Experimental setup. We deployed Getafix on a 50 machine cluster consisting (from Emulab [43]) of d430 [10] machines each with two 2.4 GHz 64-bit 8-Core processor, 64 GB RAM, connected using a 10Gbps network. We deployed Druid on dedicated machines as well as on Docker [9] containers (to constrain memory). The 50 machines are used to run up to 25 compute nodes (HNs), 16 brokers, Zookeeper and Kafka nodes, realtime nodes, and colocated clients.

Workloads. We generated data using a custom schema, and streamed via Kafka into a Druid realtime node. We used two query workloads:

- **Synthetic Workload:** Typically, Druid queries summarize results collected in a time range. In other words, each query has a start time and an interval. We generate query workloads by picking values for the start time and interval from a specified distribution. By default, we use Latest as the start time distribution and Zipfian as our interval distribution. Our choice of these parameters are driven by two observation – 1) recent data is more popular, and 2) queries with small interval range are more common (e.g., query over hours is more popular than days, etc). To enable fast experiments, we use 5 minute-sized segments, but we ran the experiments in such a way that our results hold for any segment size (e.g., hourly).
- **Production Workload:** We use the Yahoo! traces described earlier in §2.2. We scaled down the workload to fit our cluster.

Scaling Query Rate. We inject queries at a fixed rate (250 queries/s) between each client and the broker. 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 the number of clients and brokers.

Metrics. Our main metrics measure cost (storage) and performance (query latency). We measure averages as well as tail values.

Baselines. We compare Getafix using two baselines:

1. **Scarlett:** We implemented Scarlett's [24] techniques into Getafix (around 2000 lines of code). Scarlett is the closest system we found that handles skewed popularity of data. While Scarlett was implemented for Hadoop, it is intended to work generally.

   We implemented Scarlett's round-robin based algorithm [2]. The round-robin algorithm counts the number of concurrent accesses to a file to determine how many replicas to assign. The intuition is to alleviate hotspot in the system by providing more replicas. 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 chose not to do it for Scarlett too (for fairness in comparison).

2. **Uniform:** We compared our system to the simple (but popular in Druid deployments today) approach where all segments are uniformly replicated.

5.2 Comparing Getafix with Scarlett

We increase the compute capacity (number of HNs) and scale the query load proportionally. Simultaneous scaling ensures proportional load on the HNs. Concretely, for 10 HNs we use 7 clients, and for every additional 5 HNs we add 3 clients. We perform two types of experiments: 1) For *streaming experiments*, we ingest data for 30 minutes and simultaneously send queries at a specified input rate; 2) For *batch experiments*, we ingest for 30 minutes and then generate query workload for an additional 15 minutes.

Storage Space Savings. Figure 10 shows the improvement in memory and absolute latencies for the synthetic workload, while Figure 11 shows corresponding results for the production workload. For the synthetic workload, Getafix reduces median storage space usage by 2× - 3.5× compared to Scarlett and 99th percentile storage space by 2× - 3.5× (Figure 10a). For experiments with the production workload (Figure 11a), we see 2× - 3.5× reduction in 75th percentile storage and similar improvements for 99th percentile. Scarlett performance suffers because it decides to replicate solely based on segment popularity. Getafix minimizes replicas by providing popular segments with a larger share of the compute in a single machine. As a result for the popular segments, Getafix creates fewer replicas than Scarlett. As the number of HNs increase, Getafix has more choices to replicate, and so the performance improvement plateaus to about 2×.

Query Latency. We achieve these storage savings with little to no impact on query performance. The average query latencies observed for both synthetic (Figure 10b) and production (Figure 11b) workloads are stable across Scarlett and Getafix. The tail latency (99th percentile) for Getafix is marginally higher than Scarlett's in the production case, by 25% to 30%. This is because Getafix creates fewer replicas for the popular segment and under heavy query load, Druid throttles queries at the HNs. The tail latencies can be increased if Druid's underlying throttling mechanism were less aggressive.

5.3 Storge-Latency Tradeoff

Figure 12 shows the cost-performance tradeoff curve, plotting median storage used (cost) against the average latency observed (performance). We compare using results from one of the synthetic workloads running on 15 HNs and receiving 2500 queries/s. We ran 4 uniform replication experiments starting with a replication factor of 3 and going upto 12. Getafix: a) uses 4× less storage space than Scarlett while giving comparable average latency, and b) it is well below the envelope of the fixed replication strategy. Essentially, we found that Getafix was creating a number of replicas that is just at the knee of the curve in Figure 1 (§1).

[2] 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.

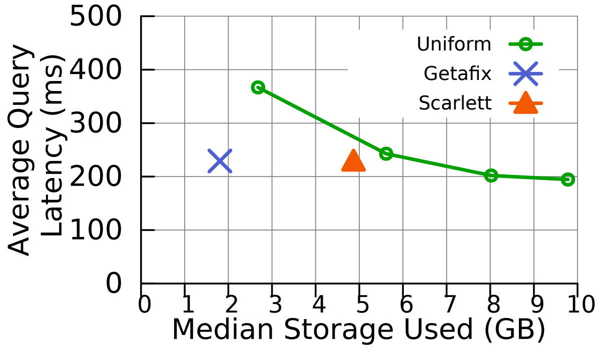


Figure 12: **Storage-Latency Tradeoff.**

5.4 Getafix Vs. Uniform Replication Under Similar Storage

Since uniform replication has a choice of storage space, we compare it against Getafix when both use similar storage. The results are shown in Figure 13. We carefully choose a replication factor for uniform such that the total storage space used by both the schemes is similar. (It turns out that the following heuristic works best to calculate uniform's number of replicas to achieve similar storage space as Getafix: calculate 75th percentile of total segment count from Getafix, and divide by the total number of segments.) We ran synthetic streaming workloads for 30 minutes with data ingestion and query generation.

We observe that both median and average latencies see an improvement between 20% - 60%. The tail latency also improves. Essentially the popular segments in the uniform approach are replicated infrequently compared to Getafix, leading to a longer queueing time at HNs hosting popular segments and increasing the median query response.

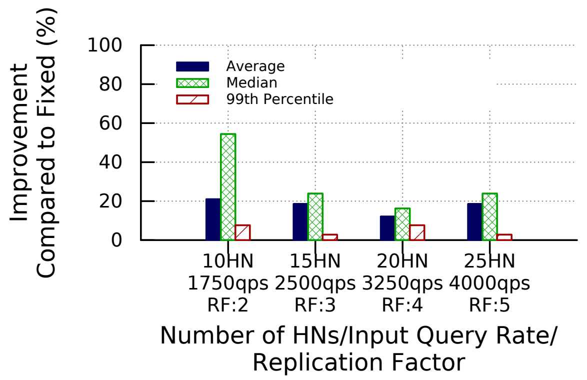


Figure 13: **Comparing performance of Uniform Replication with Getafix. Replication factor (RF) chosen such that storage is in the ballpark of Getafix.**

5.5 Benefit From Garbage Collection

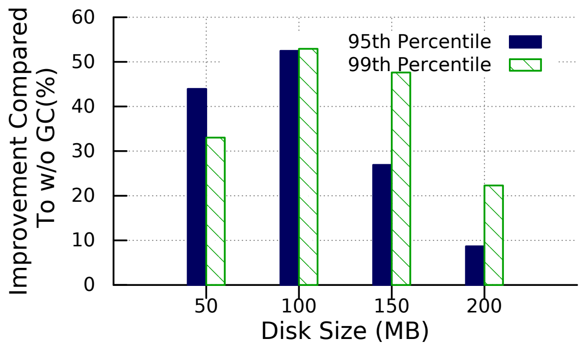


Figure 14: **Improvement in tail latency for Getafix with GC compared to without GC.**

Since the frontend tier has far less storage space than the backend tier, garbage collection of unused segments from the frontend tier is critical for performance (to allow newly popular segments to continue being loaded into the frontend). We evaluate this by scaling down disk space and the workload so that the experiments run in a reasonable amount of time. We chose 3 HNs with a combined space ranging from 150 MB to 600 MB. We ingest data for the first 30 minutes along with queries and then run another batch workload for 30 minutes.

Figure 14 plots the tail latencies (95th and 99th percentile) observed as we increase the disk size. We compare against Getafix without garbage collection. The tail latencies improve by about 25% to 55% when garbage collection is enabled. As we increase the storage size, the frontend tier becomes less saturated, thus reducing the marginal gain from garbage collection. Thus we recommend that garbage collection be enabled especially when the frontend tier is expected to be saturated, e.g., when there is a large differential between backend and frontend storage sizes, or when queries have less locality.

5.6 Broker Polling Period

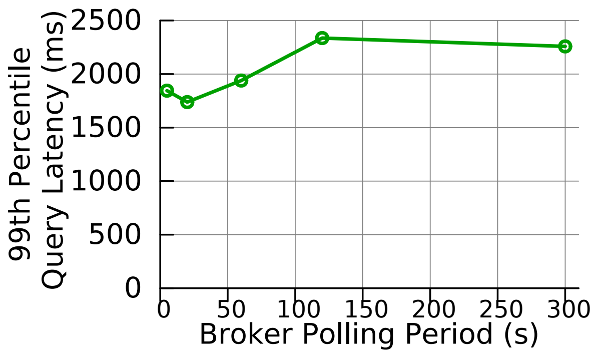


Figure 15: **Tail latencies affected by Broker polling period**

As discussed in §4.1, brokers poll latest routing information periodically to update segment routing tables. We justify why we chose 20 s as the default polling period. Figure 15 investigates the effect of changing the polling period (Getafix synthetic workload with 2500 qps on 15 HNs with Getafix ModifiedBestFit run every 1 minute. Data points at 5 s, 20 s, and 1, 2, 5 mins). Polling periods at or below 60 s give comparable latencies. Polling periods longer than this threshold cause an increase in tail latency because of a mismatch with ModifiedBestFit frequency (once a minute), and the resulting stale routing tables at the broker. We chose 20 s as the default broker polling period because it is about midway through the stable range.

5.7 Benefit of Getafix Routing Strategy

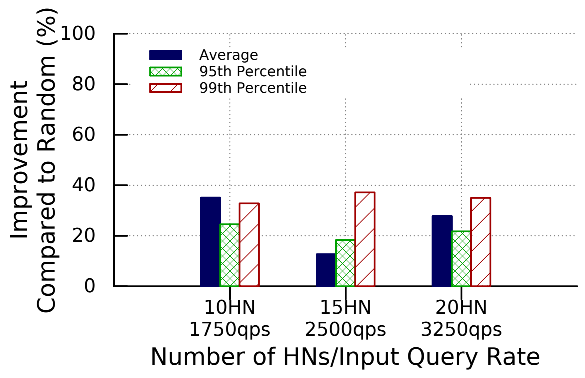


Figure 16: **Comparing Getafix Routing Strategy with Random.**

Getafix uses a custom routing scheme (§4.1) based on the result generated by ModifiedBestFit algorithm. We now evaluate the benefit of using tailored routing strategy. To compare, we replaced Getafix's query routing with a random scheme where we pick a random node from all nodes hosting a segment replica to route a query, however the replication strategy stays the same as original Getafix. We ran streaming experiments with a 30 minute ingestion and query generation.

Figure 16 plots the overall improvement observed in tail latencies and average latency for vanilla Getafix compared with the modified Getafix running the random scheme. We observe an improvement of 35% at the tail as we increase both cluster capacity and query injection rate. We conclude that it is important to use a routing strategy that is coupled with the replication location strategy. Our competing random routing is unaware, and hence leads to query bottlenecks at multiple nodes, prolonging latency especially at the tail.

6. RELATED WORK

Current distributed analytics engines [17, 23, 25, 33, 35, 45] largely decouple data management from the query routing, instead focusing on query optimization techniques. Newer systems like Druid [45], Pinot [17], etc., use a tiered storage architecture, where effectively utilizing the storage in frontend compute nodes is paramount to achieving good query performance. Our system does intelligent segment placement (using popularity), and couples query routing with it.

Data popularity for replication has been looked at before. Nectar [28] 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. Scarlett [24] uses popularity for deciding replication of files in batch processing systems, and we have implemented its techniques and shown Getafix performs better. Other works [39, 42] have used performance SLAs to determine the number of replicas and how to assign them among different hosts; Getafix could be extended to satisfy SLAs (but this is beyond our scope here).

Adaptive schemes have been used for replicating read/write objects to improve operation latency in databases [44]. They have also been implemented for memory caching systems [29] to achieve better cache performance under skewed data popularity. In interactive data analytics engines, since data is immutable, our adaptive replication tries to optimize storage and network overheads involved in loading and storing this data. Facebook's f4 [36] 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 [32], 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.

Workload-aware data partitioning and replication has been explored in Schism [27], whose techniques minimize cross-partition transactions in graph databases. 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. As mentioned earlier we believe this approach is ad-hoc and that an adaptive strategy like Getafix presents a system that is easier to manage. There are other works which look at adaptive partitioning for OLTP systems [37] and NoSQL databases [26] respectively, however they do not explore Druid-like interactive analytics engines.

7. ACKNOWLEDGMENTS

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

8. SUMMARY

We have presented techniques intended for interactive data analytics engines like Metamarkets/Yahoo!'s Druid, Amazon Redshift, and LinkedIn's Pinot. Our techniques use latest (running) popularity of data segments to determine their placement and replication level at compute nodes, and route queries to appropriately to these nodes. Our solution to the static query/segment placement problem is provably optimal in both query latency and total storage space used. Our system, called Getafix, realizes the solution to the dynamic version of the problem, and effectively integrates adaptive and continuous segment placement/replication with query routing and garbage collection. We implemented Getafix into Druid, the most popular open-source interactive analytics engine. Our experiments use both synthetic traces and production traces from Yahoo!'s production Druid cluster. Compared to the best existing techniques (Scarlett) Getafix either improves storage space at both the median and tail by 2× to 3.5× while achieving comparable query latency. Compared to the default strategies that use uniform replication, Getafix is 20-60% faster under the same storage constraints.

9. REFERENCES

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

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

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

[4] Apache Flink. https://flink.apache.org. visited on 2017-2-26.

[5] Apache Samza. http://samza.apache.org. visited on 2017-2-26.

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

[7] 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.

[8] Bin Packing Problem. https://en.wikipedia.org/wiki/Bin_packing_problem. visited on 2016-3-2.

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

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

[11] Flurry. https://developer.yahoo.com/flurry/docs/. visited on 2017-2-26.

[12] Fragmentation (computing). https://en.wikipedia.org/wiki/Fragmentation_(computing). visited on 2017-3-1.

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

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

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

[16] LinkedIn. http://linkedin.com. visited on 2017-2-26.

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

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

[19] Pinterest. http://pinterest.com. visited on 2017-2-26.

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

[21] Yahoo! https://www.yahoo.com. visited on 2017-2-26.

[22] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. Distributed operation in the Borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 882–884, New York, NY, USA, 2005. ACM.

[23] 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.

Proc. VLDB Endowment, 8(12):1792–1803, Aug. 2015.

[24] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed content popularity in Mapreduce clusters. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 287–300, New York, NY, USA, 2011. ACM.

[25] 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. *Proc. VLDB Endowment*, 8(4):401–412, Dec. 2014.

[26] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça. Met: Workload aware elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA, 2013. ACM.

[27] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endowment*, 3(1-2):48–57, Sept. 2010.

[28] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[29] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[30] 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, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[32] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[33] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, 7th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[34] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[35] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endowment*, 3(1-2):330–339, Sept. 2010.

[36] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 383–398, Berkeley, CA, USA, 2014. USENIX Association.

[37] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[38] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[39] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 89–102, New York, NY, USA, 2006. ACM.

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

[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. *Proc. VLDB Endowment*, 8(3):245–256, Nov. 2014.

[42] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[43] 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 *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[44] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, June 1997.

[45] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.