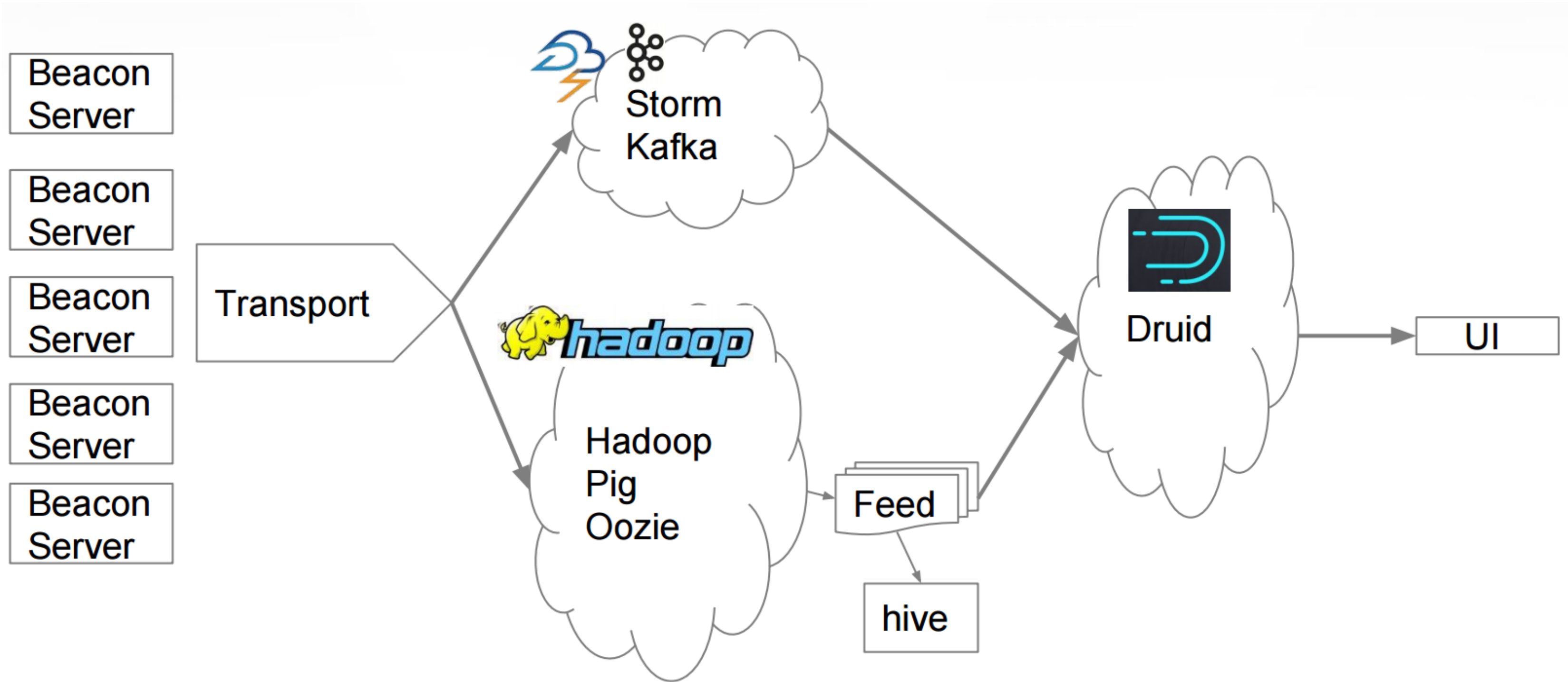
Getafix: Workload-aware Distributed Interactive Analytics

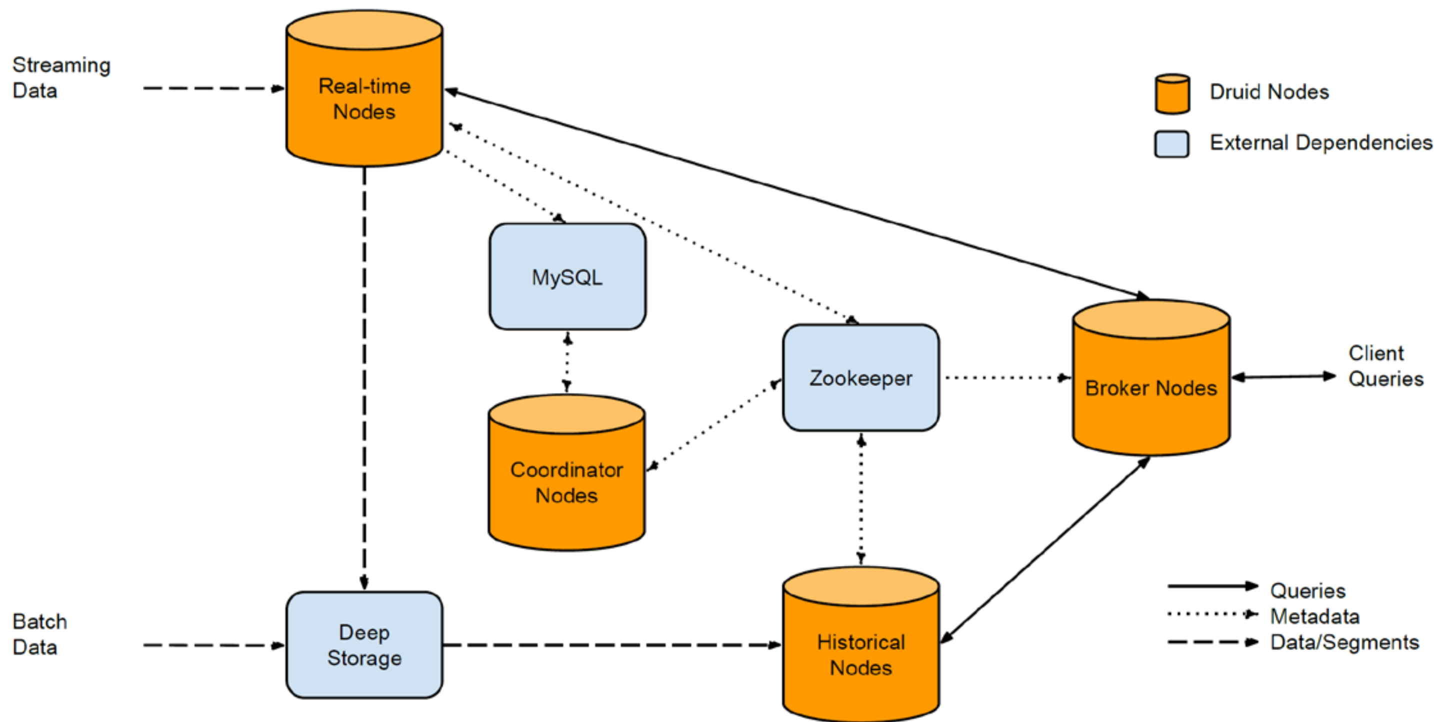
Presenter: Mainak Ghosh
Collaborators: Le Xu, Xiaoyao Qian, Thomas Kao, Indranil Gupta, Himanshu Gupta

Data Analytics



Picture borrowed from https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51640

System Model



Picture borrowed from https://github.com/druid-io/druid/blob/master/publications/whitepaper/figures/cluster.pdf

Interactive Data Analytics

- Data is indexed by timestamp
- Stored in batches (*segments*) -- e.g. hours worth of click logs
  - Segments are *immutable*
- Queries are *aggregation* (sum, count, etc.) over a time period's worth of collected data
  - Queries access multiple segments. Each *segment query pair* can be scheduled in parallel
- Data stored two separate tiers.
  - Backend tier is called *deep storage* (Amazon S3, HDFS). Stores all segments from the start of time.
  - Frontend tier (*historical nodes (HN)*) is the compute tier. Loads relevant segments from backend tier. Queries executed out of this tier.

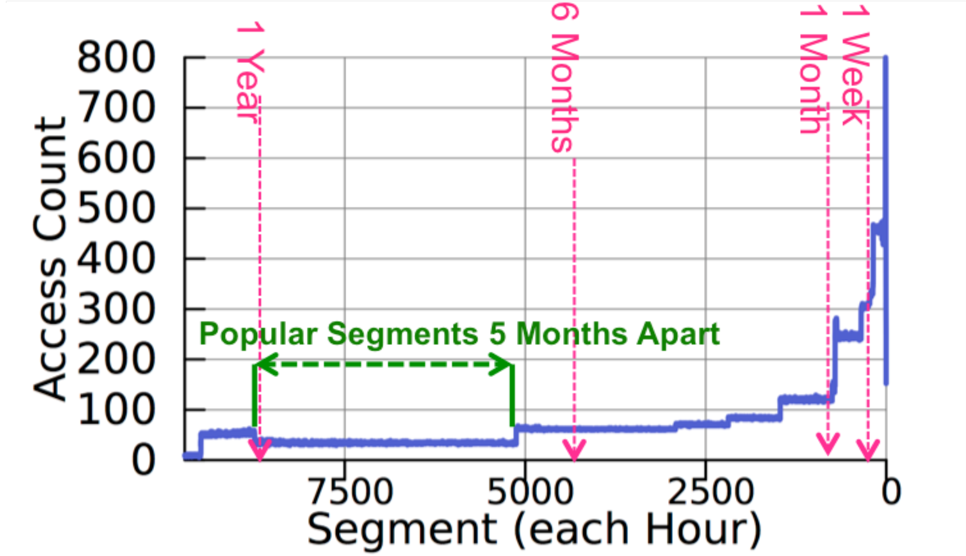
Importance of Data Management

- Goal is to achieve high query throughput
- Achieves it at query execution level by being ***massively parallel***
- Two popular segments should not be collocated (Placement)
- Segments should be replicated for better query load balancing (Replication)
- Full replication is prohibitive in a ***tiered store architecture***

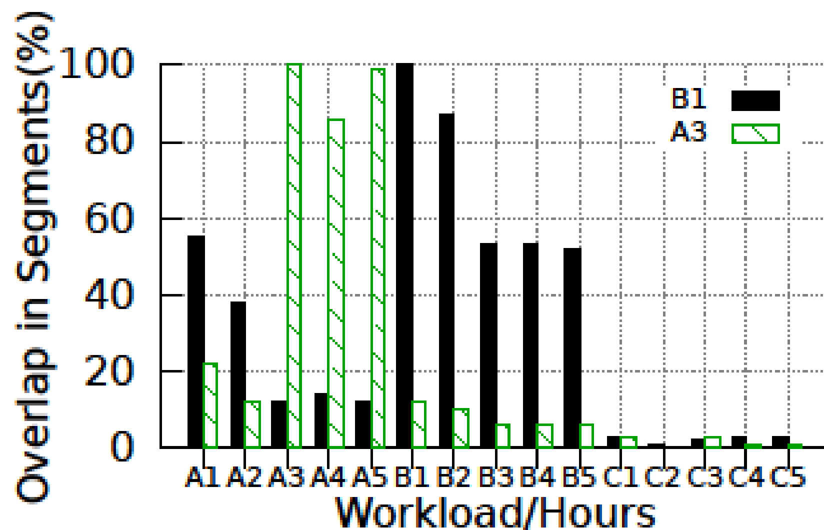
State-of-the Art

- Uniform Replication: Suboptimal since some data more popular than others
  - Traces from Yahoo!'s Druid cluster show that top 1% of data is an order of magnitude more popular than the bottom 40%
- Manual Tiering: A sys-admin manually creates storage tiers with different replication factors and assigns segments based on an estimate of popularity.
  - Laborious and cannot adapt in real-time to changes in query pattern and/or cluster configuration
- Based on recency (current Druid): Assumes that most queries will touch data that was ingested recently and is only a few hours to days old
- Based on Input Query Rate (Scarlett): Can lead to good query performance but incurs large storage costs

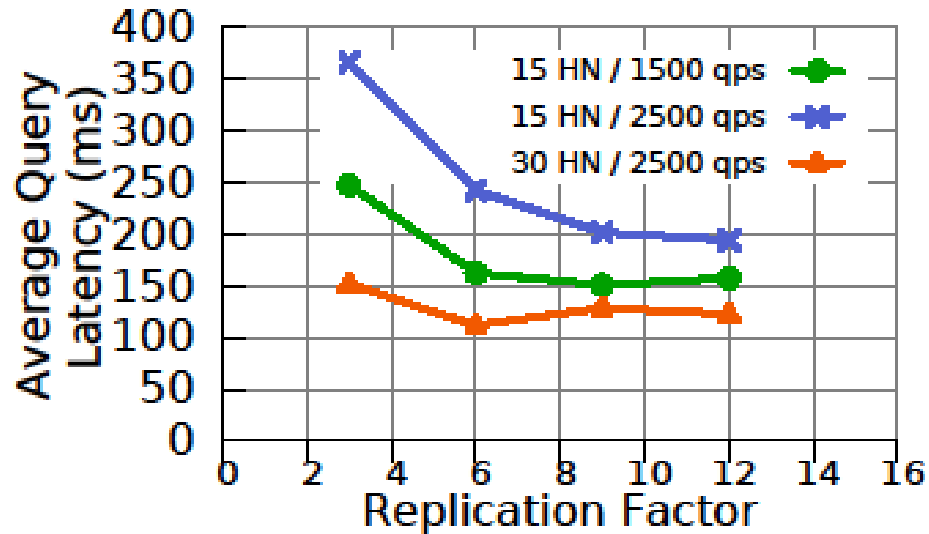
Old data is popular too



Old data is popular too



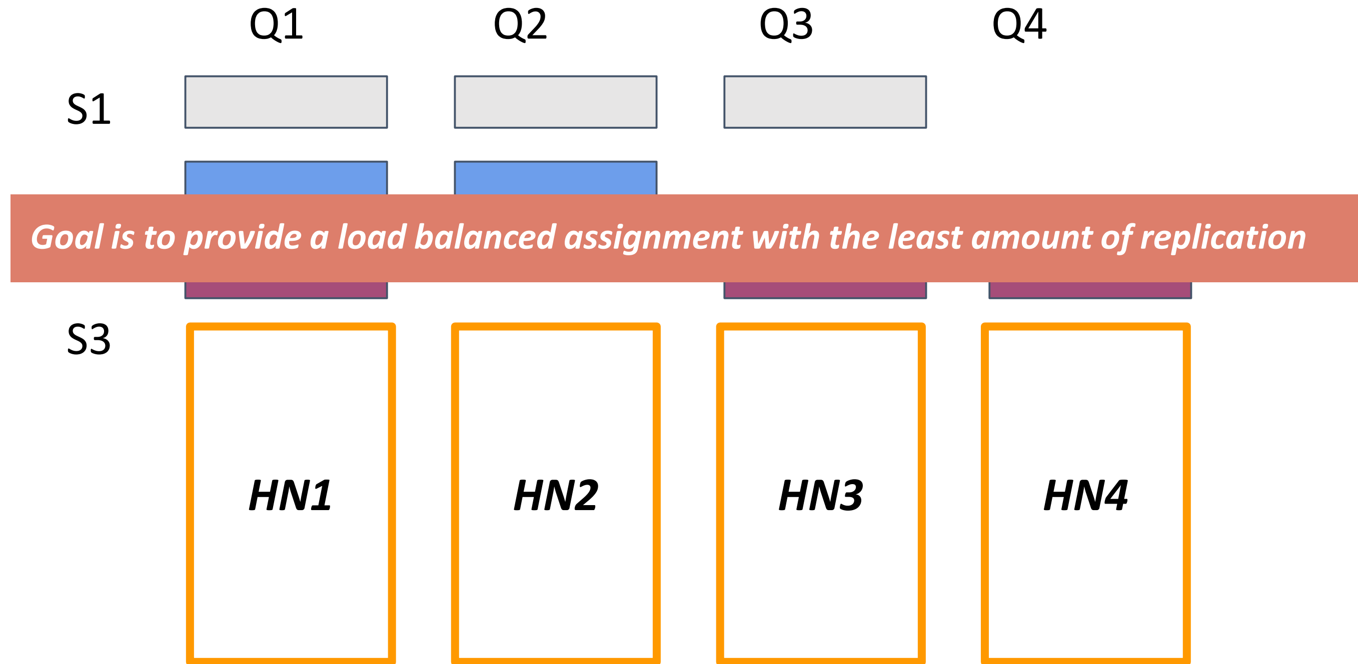
Input Query Rate Not Enough



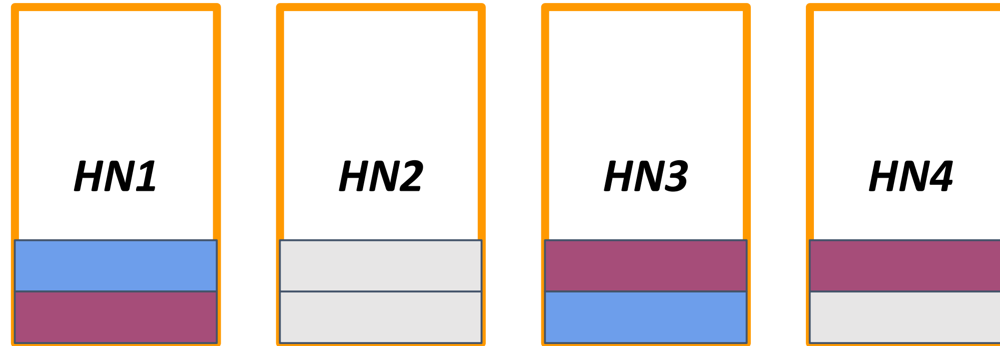
Contribution

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

Problem

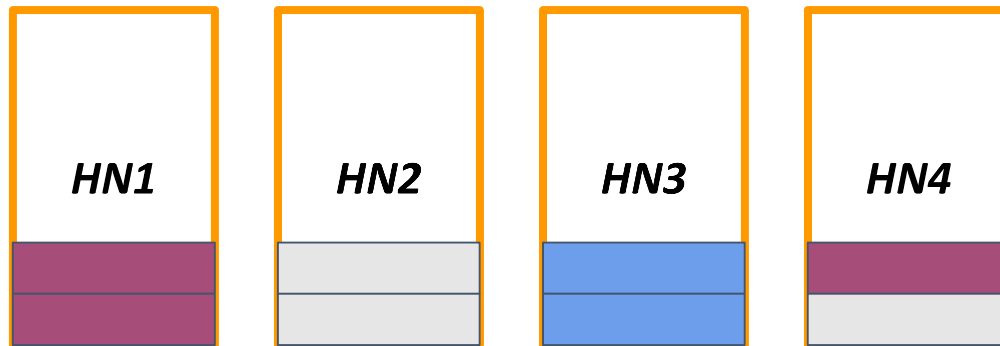


A possible solution...



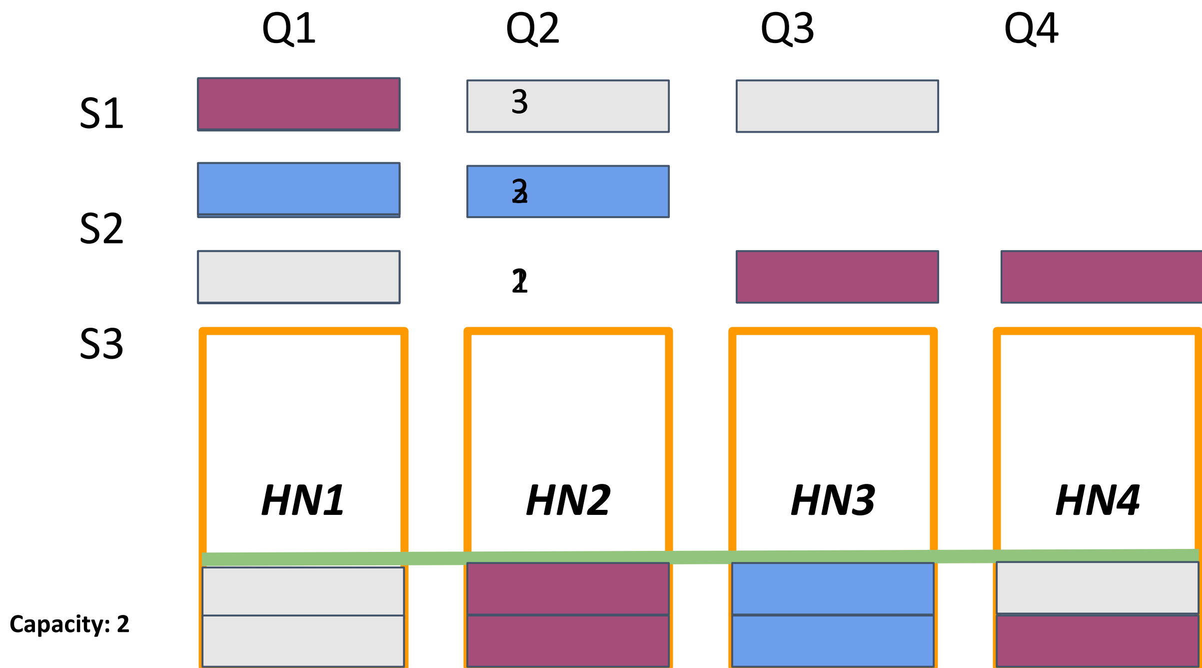
Load Balanced Assignment. Number of Replicas: 7

Solution which minimizes number of replicas



Load Balanced Assignment. Number of Replicas: 5

Algorithm



Assign bin capacity as total number of query segment pairs by number of historical nodes

Create a priority queue on segments by access count

Pick the highest priority segment, assign to historical node based on a policy and return the rest to the priority queue

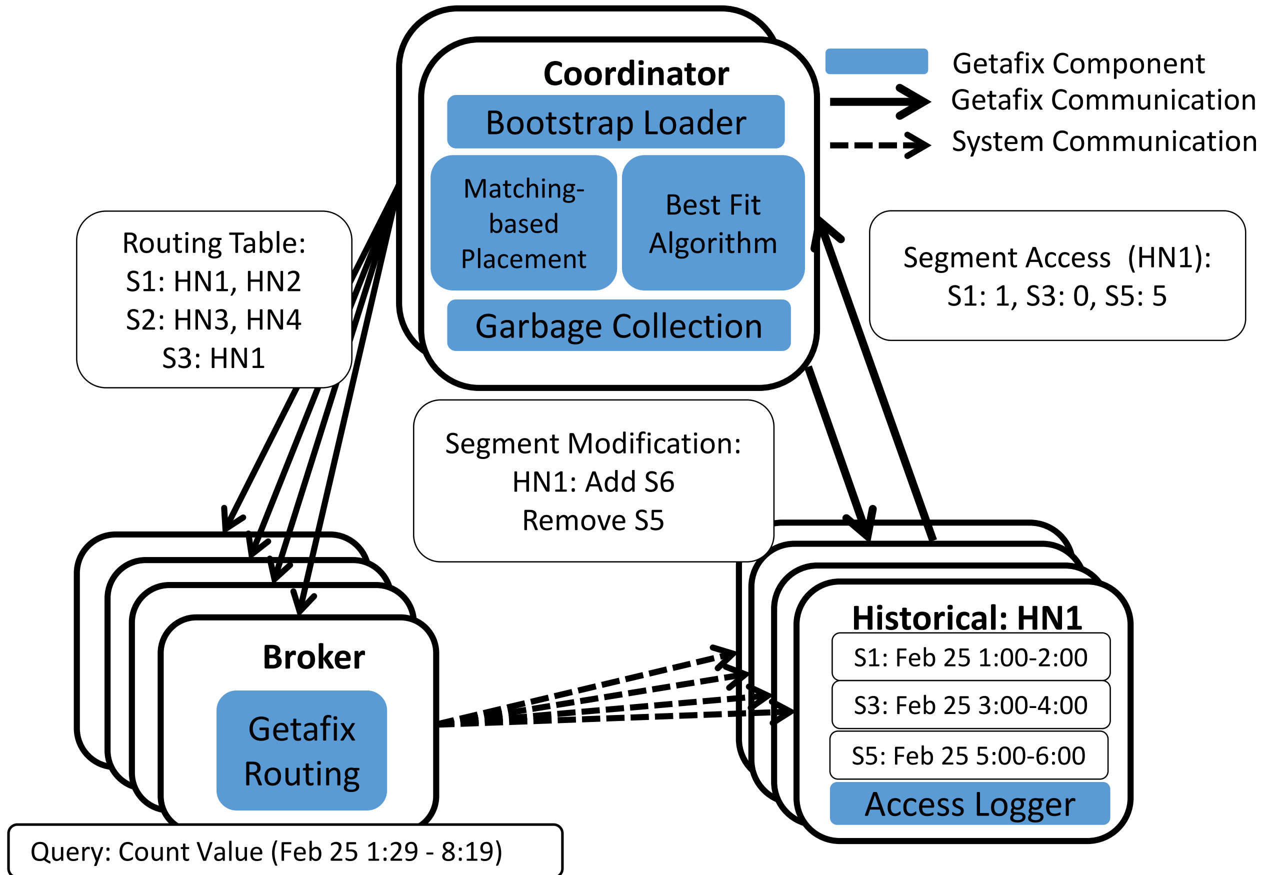
Continue till the queue is not empty

Policy

- ***First Fit***: We choose the next HN (lowest HN id) that is not yet full. Intuition is to fill up bins in a FIFO way and reduce external fragmentation
- ***Largest Fit***: We choose the HN with the largest available capacity leftover. Intuition is to create large holes for later.
- ***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, with ties broken by lower HN id. It gives better packing.

Best Fit Policy provides a load balanced assignment with the least amount of replication

Getafix: System Design



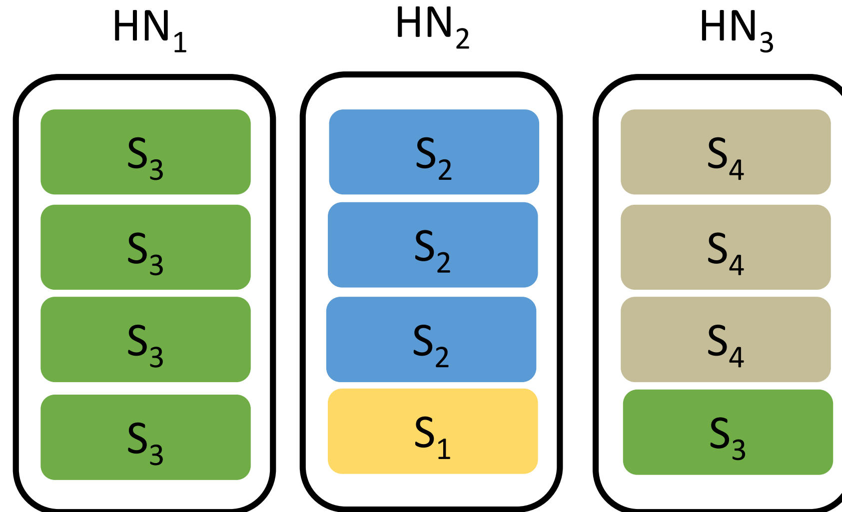
Algorithm

- In a dynamic system, we break up the execution over multiple time windows
- Coordinator collects the segment counts from historical nodes
- Popularity of a segment is the weighted average of its access counts over a fixed number of past windows
- Exponentially decay past window counts
- Run the best fit algorithm which returns the expected number of segment replicas in the system

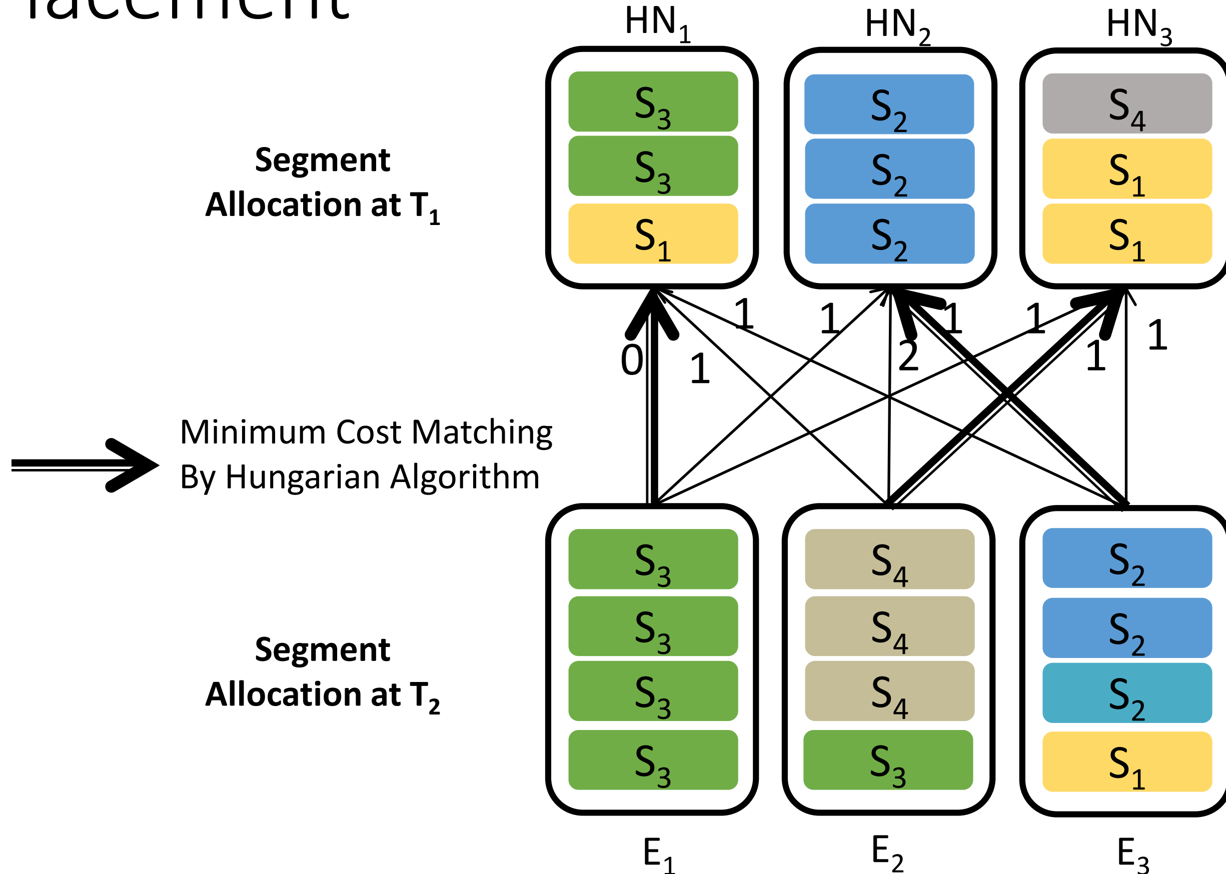
Example Execution

Segment Access Counts

Segment Name	S_1	S_2	S_3	S_4
Count	1	3	5	3



Data Placement



Query Routing

- Apart from colocating segments, HNs should also serve the right number of queries for load balance

S1	0	0	100
S2	0	0	100
S3	80	20	0
S4	0	100	0

- Broker periodically polls the routing table.
- Query for a segment is routed by prioritizing historical nodes based on the routing table

Other Optimizations

- Segment Load: Segment is eagerly replicated once at a random HN, independent of whether some queries are requesting to access it.
- Lazy Deletion: We rate limit the number of replicas that gets deleted for a segment. Retains segments in case their popularity increases again, and avoids network IO due to reloading
- Garbage Collection: In resource constrained scenarios, lazy deletion can saturate storage. We garbage collect unused segments. Gets triggered when the storage runs low.

Evaluation

- Setup: 50 d430 node cluster from Emulab. 2 2.4 GHz 64-bit 8 core processors, 64GB RAM, 10 Gbps network
- Workloads: Data generated using a custom schema and streamed via Kafka. Two query workloads:
  - Synthetic: 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.
  - Yahoo Production Trace: We scale down these workloads to suit our cluster size
- Metrics: Cost is storage and performance as query latency (average and tail).
- Baselines:
  - Scarlett: Implemented round-robin algorithm (in 2000 lines of code). Uses the number of concurrent accesses to determine how many replicas to assign.
  - Uniform

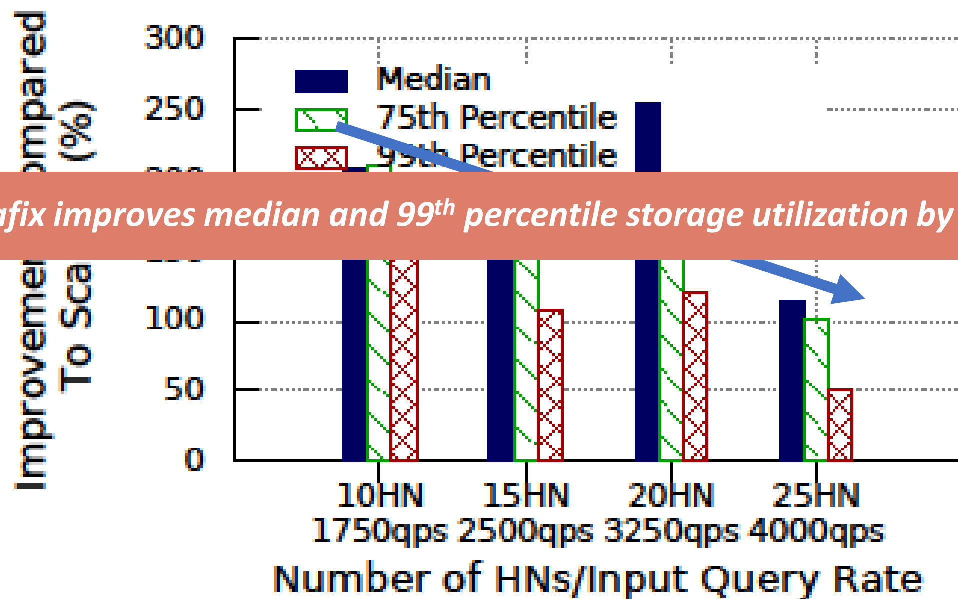
Result Summary

- Compared to the best existing strategy (Scarlett), Getafix reduces median and 99th percentile storage space by 2x - 3.5x, 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.

Result Summary

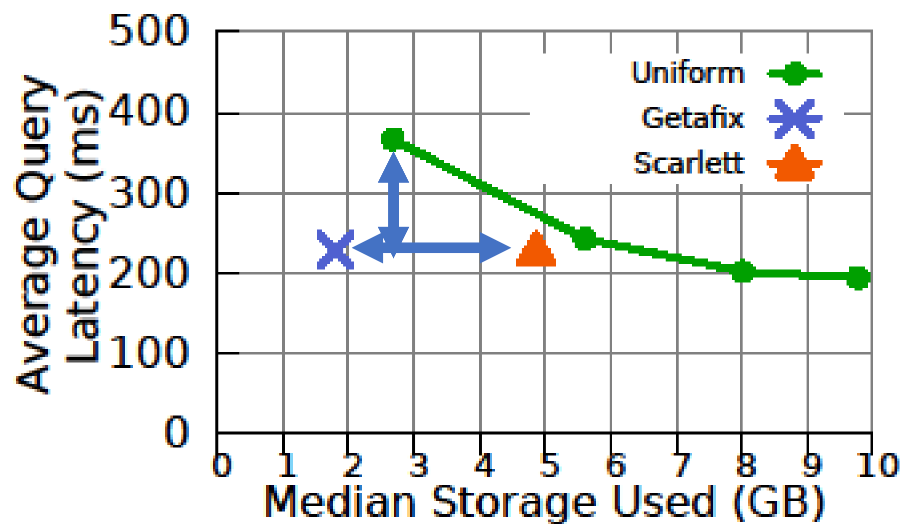
- Compared to the best existing strategy (Scarlett), Getafix reduces median and 99th percentile storage space by 2x - 3.5x, while producing comparable query latency, across both synthetic and Yahoo! workloads, and in both batch and streaming settings.
- Compared to fixed replication (a common strategy 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.

Storage Utilization Getafix v Scarlett



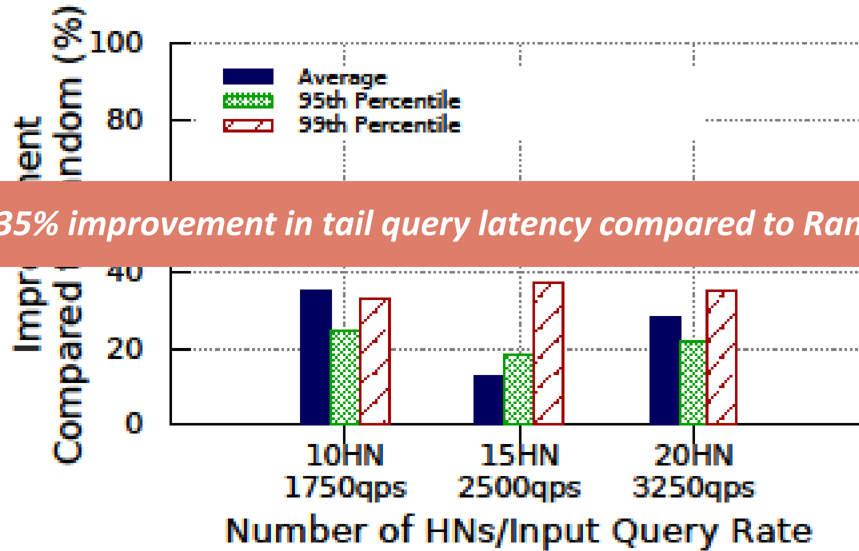
Getafix improves median and 99th percentile storage utilization by 2x – 3.5x

Storage Latency Tradeoff



20% improvement in average query performance compared to smallest uniform experiment

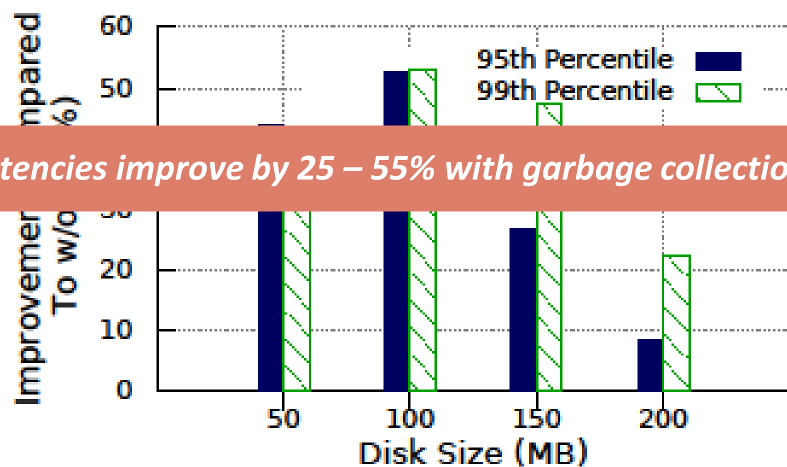
Benefit of Getafix Routing Strategy



35% improvement in tail query latency compared to Random

Validates our decision to couple query routing to data management

Benefit of Garbage Collection



Tail latencies improve by 25 – 55% with garbage collection enabled

Related Work

- Current analytics engines like Trill, Pinot, Dremel, Impala, etc decouple data management from query routing.

- Data popularity has been used by systems in the past.
  - Nectar does not store unpopular data and recomputes it on-the-fly. No intermediate data in interactive analytics engines
  - Scarlett uses it from deciding the number of replicas. Evaluation compares against Scarlett.

- Workload-aware adaptive schemes has been used for data management in databases [Wolfson et al], memory caching systems [Hong et al].

- Erasure coding of “warm” data (f4), and compression techniques (Blowfish) have also been explored in the literature. These techniques are orthogonal and can be used in combination with Getafix.

Summary

- We proposed techniques for data management in interactive data analytics engines
- We use segment popularity information to decide data placement and replication at the compute nodes and also to route queries
- 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, effectively integrates adaptive and continuous segment placement/replication with query routing in a dynamic setting.
- Compared to the best existing techniques (Scarlett) Getafix improves storage space at both the median and tail by 2x to 3.5x while achieving comparable query latency

Thank you

Problem

- We assume each query takes same amount of time (unit time) to process a segment it touches.
- Uniformly distributing these segment query pairs will give time optimal schedule
- May not be the least amount of replication
- Similar to *bin packing problem*
- In this problem, we have to place segment query pair (ball) to HN (bin) such that the sum of the number of unique segments in each bin is minimized

Algorithm

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

Algorithm 1: Generalized Allocation Algorithm.