EASY: Efficient Segment Assignment Strategy for Reducing Tail Latencies in Pinot

Seyyed Ahmad Javadi, Harsh Gupta, Robin Manhas, Shweta Sahu, Anshul Gandhi
PACE Lab, Department of Computer Science, Stony Brook University
{sjavadi,hagupta,rmanhas,shsahu,anshul}@cs.stonybrook.edu

***Abstract*—Customer facing online services, such as LinkedIn and Uber, rely on scalable and low-latency data stores to maintain acceptable query tail latencies. An important challenge for managing the performance of these systems is the assignment of newly created data segments to data nodes to balance load. Given the rate at which these services are accessed (thus generating new data), the segment assignment problem is particularly important. This paper presents EASY, an efficient segment assignment strategy that leverages analytical modeling to predict the future load induced by data segments, thus allowing for long-term balancing of load across data nodes. Our implementation and evaluation of EASY on Pinot shows that we can significantly reduce query tail latencies in the presence of dynamically generated data segments.**

***Keywords*—segment assignment strategy, tail latency, Pinot**

I. Introduction

Large-scale and real-time Online Analytical Processing (OLAP) is a major requirement for customer facing companies. A popular distributed near-realtime OLAP solution is Pinot [13], that is extensively used at LinkedIn and Uber for serving user queries (such as the Profile View functionality of LinkedIn) and for internal analysis.

Pinot leverages a simple architecture where every table is divided into data "segments" distributed among worker nodes. Every segment typically contains information for a period of time (e.g., one hour or one day). An incoming query from a client to Pinot is run simultaneously across workers hosting the target segments. The end-to-end response time of a query in Pinot depends on the longest query latency among target workers, as all individual (per-worker) results need to be integrated by the broker node(s) before sending the response back to the client.

In such distributed data store systems, the Segment Assignment Strategy (SAS) has a significant impact on query latency. SAS dictates the placement of new segments on worker nodes; new segments are created dynamically as time passes. Naive SAS such as round-robin can result in hotspots, severely impacting query tail latencies (see Section IV).

Existing SAS in production systems often employ a decentralized and scalable utility function (or cost function) approach whereby each server is assigned a cost that can be easily computed; incoming segments are then assigned to the lowest cost server, whose cost is then updated. While popular open source OLAP solutions such as Pinot and Druid [17] have their own cost-based SAS, these default strategies have their shortcomings. The Pinot SAS aims to balance the number of segments across workers. Our experimental results show that this SAS leads to unbalanced load and high tail latencies. Druid implements a more advanced SAS by taking the time range of segments into account. However, as we show in Section IV, there is much scope for improvement, especially when there are multiple tables in the data store.

We propose a new load-aware SAS, EASY (Efficient segment Assignment StrategY), that outperforms existing SAS solutions in terms of load distribution among workers and, importantly, in terms of query tail latency. EASY works by first *passively computing* the server load created by segments as queries operate on them. Then, EASY models this segment load and *predicts, at run time*, the future load induced by a segment during its remaining (finite) lifetime. This task is complicated by the fact that load depends critically on the age of a segment; we find that, as time passes, the popularity *and* load contribution of a segment decreases non-linearly.

We implement EASY on top of Pinot and experimentally evaluate our SAS using a custom LinkedIn-like data and query set (guided by the first author's understanding of LinkedIn's Pinot system while he was interning there); we open source all our implementation and code [11]. Our results show that EASY significantly improves the load balance among worker nodes, reducing query tail latencies by up to 6–21% when compared to the default SAS of Pinot and Druid. Importantly, EASY requires few changes and creates negligible overhead.

In summary, the contributions of this paper are:

- We present a novel and efficient load-aware SAS for Pinot.
- We design and implement a realistic data set and benchmark for evaluating Pinot, and open source it [11].
- We implement our SAS on Pinot (publicly available [11]), and experimentally evaluate it by comparing with the default SAS of Pinot and Druid.

II. Background and Prior Work

This section provides an overview of Pinot, and then discusses important prior work on SAS to put our work in context.

A. Background on Pinot

Pinot is a distributed near-realtime OLAP (On-Line Analytical Processing) data store that is used at LinkedIn for various user-facing functions and internal analysis. Pinot has been open source since 2015 and is currently being used by other companies such as Uber. Pinot is designed to be able

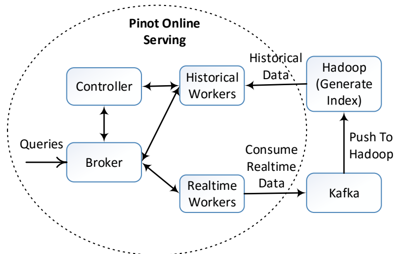


Fig. 1. Overview of Pinot's architecture.

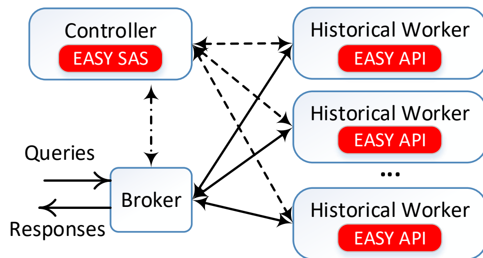


Fig. 2. EASY's solution architecture. Components we add are shaded in red.

to process 100 Million SQL-like queries for 100 Billions of records in 10s of ms latency. While this paper focuses on Pinot, our proposed techniques can be applied to other OLAP systems as well.

Figure 1 illustrates the Pinot architecture including the three main components: (1) controller, (2) broker, and (3) worker nodes. The controller is responsible for cluster-wide coordination and segment assignment to worker nodes (SAS). The broker (or brokers) receives queries from clients, distributes them among workers, and integrates the results from the workers and sends the final result back to clients; the end-to-end query response time can be obtained at the broker. The worker nodes host data segments and respond to query sub-requests that originate from the broker. Query logs are maintained at workers.

Pinot processes recent data (e.g, a few days old) using Realtime Workers and older data (may overlap with realtime data) using Historical Workers, as shown in Figure 1. Realtime data is pushed to Historical Workers as time passes (e.g., daily) or when a given number of records have been ingested; the data is pushed via Kafka and HDFS. In this paper we focus on Historical Workers, that store the bulk of the data.

Historical Workers store data in the form of a pre-built index called *segment*; every table has its own segments. Segments store contiguous data for a given time range; there is a row of data columns for each time interval within the range. Every segment thus has an associated start time and end time for its data (in the Time column). Note that there may be a table-specific expiry time that dictates how long segments should be retained by Historical Workers. Once the expiry time, say 3 months, elapses, the associated segments are deleted.

B. Prior work

We now discuss important prior work on SAS; other related works are discussed in Section V. We implement EASY on top of Pinot by modifying Pinot's SAS. The default SAS for Pinot balances the number of segments across workers. By contrast, EASY aims to minimize query tail latencies by reducing the load imbalance between workers; we show in Section IV that EASY significantly outperforms the default Pinot SAS.

The closest systems to Pinot are Druid [17] and ClickHouse [1]. Druid's SAS [17] is similar to EASY, except that Druid's cost function depends only on the time range of a segment and not its load. As we show in Section IV, EASY outperforms Druid by specifically taking segment load into account. ClickHouse [1] is also an OLAP system but does not employ time-ranged segments, like Pinot. Data is distributed over workers based on weights that must be manually assigned by cluster administrators.

Getafix [8] uses a modified bin packing approach to distribute incoming segments across workers based on their popularity. The authors define segment popularity in terms of access count of the segment, and popularity is aged exponentially. Likewise, Copeland et al. [2] distribute data segments to worker nodes so as to balance the access frequency of resident data objects. Furtado [7] proposes a data placement schema based on hash-partitioning to favor most frequently accessed keys for a relational database. BlowFish [9] maintains a request queue per segment and uses queue length as an estimator of segment load; this queue length information is then used to distribute segments across servers. However, the access frequency or outstanding requests for a segment may not directly correlate with the segment load. For example, a less popular segment may still contribute significantly to server load because of its size or its structure (e.g., number of columns). By contrast, EASY models popularity based on its estimated load, which is a more direct indicator of the cost of a segment than its access frequency.

III. System Design and Implementation

We now present the design of EASY, followed by the cost function for SAS, and finally the implementation details of EASY on Pinot.

A. Solution architecture

Figure 2 shows the solution architecture of EASY; the components that make up EASY are shown in red. Since SAS is managed by the controller, we implement our EASY SAS in the controller; the mathematical details of our SAS are presented in the next subsection. When a new segment is generated, the controller sends a request to all workers. Each worker, in turn, computes its cost function and returns the value to the controller via an API call. The controller then picks the r workers that have the smallest cost values, and places r replicas of the incoming segment on these workers. r is a user-specified value; we set $r = 1$ in our implementation.

To facilitate the computation of the cost function, each worker logs the total cpu time spent, cpu_time_Q, and the total number of rows scanned, row_scan_Q, by each query Q. Note that Q will likely span multiple segments; we thus also log a list of segments scanned by Q. However, we do not log

segment-level information, such as segment-level cpu time and rows scanned, as this information logging requires significant overhead and may be computationally infeasible. Instead, we estimate segment-level information from cpu_time_Q and row_scan_Q, as discussed next.

B. EASY's cost function

Recall that Pinot selects the lowest cost worker nodes for each incoming segment. The default cost function in Pinot simply assigns one unit of cost for each segment in a worker node, thus assigning an incoming segment to the r workers with the lowest number of segments. Unfortunately, this cost function does not take into account the server load that each segment contributes and may contribute in the future. The cost function for EASY is specifically designed to efficiently address this shortcoming.

High-level idea. The high-level idea behind EASY's cost function is to estimate the server load that each segment will induce *during its remaining lifetime*. The server load contribution of a segment is challenging to compute as it depends on several factors, including (i) the popularity of the segment, (ii) the size of the segment (number of rows), (iii) the query mix that typically targets the segment and its relative complexity, and (iv) the structure of the segment (number of columns and their content). Worse, predicting the load that a segment may contribute to in the future requires an understanding of how induced load changes with time. Clearly, modeling all of these factors will require significant time and effort, leading to inefficient SAS design.

Instead, EASY simply models the total server load contribution of a segment of a given table based on previously observed data. Specifically, we compute the *total cpu time spent by all queries actively scanning a segment*, and use this as a proxy for load contribution. We find that this cpu time per segment *per* query decreases with the age of a segment, possibly because of caching. We thus also model this decaying trend of cpu time as a function of the *segment age* (difference between current time and segment start time).

To enable predictions of future load that a segment may induce, we learn the *cpu_time per row as a function of segment age* for a typical segment of each table. Then, for any segment of a table, we predict its cpu_time contribution based on its number of rows during its entire lifetime as it ages (since segments expire after some expiry time).

Our approach differs from existing approaches since we predict the *future load* induced by any segment. Further, we model the actual load induced by a segment as opposed to only modeling its popularity or frequency of access, which are not accurate enough estimators of load (see Section IV).

1) Passive model training: EASY passively computes its estimates of load per segment based on the measured load induced by incoming queries on existing segments. Further, to account for changes in workload, EASY periodically updates its estimates in each interval (one hour, in our implementation).

Computing cpu_time per segment. As discussed in Section III-A, we track the total cpu_time of each query Q, say cpu_time_Q, at each worker. To determine the contribution of individual segments to this cpu time, we also keep a track of the segments, and the specific time range within the segments, that each query scans. Let S_Q be the set of segments scanned by query Q, and let t_s be the time range, in hours, of segment $s \in S_Q$ that Q scans (obtained via the WHERE clause of Q). In our implementation of Pinot, each segment represents one day, and so the fraction of segment s that is scanned by Q is $f_s = t_s/24$. We now estimate the number of rows of s scanned by Q (not directly available via Pinot) as $f_s \times row_count_s$, where row_count_s is the total number of rows in segment s and is already known to Pinot. Finally, we estimate the contribution of segment $s \in S_Q$ to cpu_time_Q as:

$$cpu_time_Q^s = cpu_time_Q \times \frac{f_s \times row_count_s}{\sum_{x \in S_Q} f_x \times row_count_x} \quad (1)$$

The total cpu_time contribution of s based on all queries observed in the past interval is then estimated as:

$$cpu_time^s = \sum_{observed\ Q} cpu_time_Q^s \quad (2)$$

Computing row_scan per segment. We use a similar approach to estimate the number of rows scanned for segment s by all queries in the past interval as:

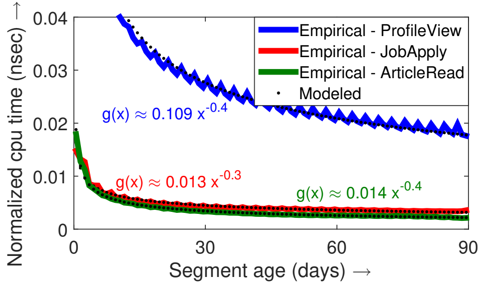
$$row_scan^s = \sum_{observed\ Q} \frac{row_scan_Q \times f_s \times row_count_s}{\sum_{x \in S_Q} f_x \times row_count_x}, \quad (3)$$

where row_scan_Q (logged by EASY) is the total number of rows, across all segments, scanned by query Q.

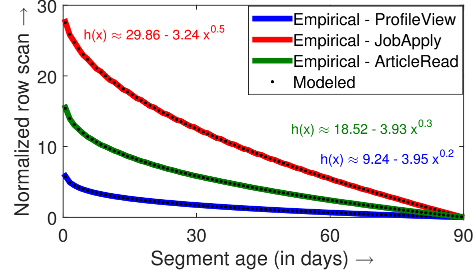
Load modeling as a function of age. We now model the load induced by any segment based on its age; this will allow us to online predict the future load created by a segment in Section III-B2. To enable load prediction for any segment size, we normalize cpu_time^s by row_scan^s; we refer to this as:
normalized cpu_time: the total cpu time per scanned row of segment s incurred by all queries in the last interval.
Likewise, we normalize row_scan^s by row_count^s to get:
normalized row_scan: the total rows scanned per row contained in segment s by all queries in the last interval.

Figure 3 shows our empirical results for normalized cpu_time and row_scan for three different Pinot tables (see Section IV-A for details on our experimental setup). We see that both values decrease non-linearly with segment age; the decrease for row_scan is to be expected as segment popularity drops with time (older segments are queried less frequently compared to newer segments).

To enable efficient predictions for new segments, we model the empirical observations. Given that popularity for segments is Zipf distributed, we fit the empirical values as $c_0 + c_1/x^{\alpha}$, where c_0 and c_1 are coefficients to be learned and α is a parameter. Our regression results for these models are shown as dotted lines in Figure 3 along with the modeled equations; we find that setting $c_0 = 0$ does not significantly affect the modeling accuracy for normalized cpu_time, and so we simplify this model accordingly. The regression fit is very



(a) Normalized cpu_time as a function of segment age.



(b) Normalized row_scan as a function of segment age.

Fig. 3. Empirical and modeled estimates for normalized cpu_time and row_scan for segments of three different tables. Also shown are the regression fit model equations for each case. The mean modeling error is less than 5% for cpu_time and less than 3% for row_scan for all tables.

close to the empirical observations, thus the dotted lines coincide with the solid (empirical) lines in the figure. The modeling error for cpu_time (g(x) in Figure 3(a)) is 3.24%, 4.11%, and 2.75% for ProfileView, JobApply, and ArticleRead tables, respectively. The modeling error for row_scan (h(x) in Figure 3(b)) is 2.94%, 1.16%, and 0.97% for ProfileView, JobApply, and ArticleRead tables, respectively.

2) Online load prediction: To predict the future load induced by a segment, EASY leverages the above described models of $g()$ and $h()$, and integrates the predicted load over the remaining lifetime of the segment. In particular, at time t, for a segment s with segment start time $start_s$ and row_count_s total rows, EASY predicts its future load as:

$$load_s(t) = row_count_s \times \int_{t-start_s}^{expiry} g(x)\ h(x)\ dx, \quad (4)$$

where $t - start_s$ is the age of s and $expiry$ (3 months in our implementation) is the expiration duration of s. Note that $g(x) \times h(x)$ represents total cpu time per row of segment s, and thus multiplying this quantity with row_count_s gives us the total cpu time for segment s; integrating over the remaining lifetime gives us the predicted load induced by s.

Since our accurate models for normalized cpu time, $g(x) = a \cdot x^{\alpha}$, and row scan, $h(x) = b + c \cdot x^{\beta}$, are relatively easy to express (where a, b, c, α, and β are regression coefficients, as shown in Figure 3), we can obtain Eq. (4) in closed-form as:

$$load_s(t) = row_count_s \cdot \Big(\frac{ab}{(\alpha+1)} \left(expiry^{\alpha+1} - (t - start_s)^{\alpha+1}\right) + \frac{ac}{(\alpha+\beta+1)} \left(expiry^{\alpha+\beta+1} - (t - start_s)^{\alpha+\beta+1}\right)\Big) \quad (5)$$

Given this closed-form expression, computing the segment load under EASY is computationally efficient; hence the name EASY (Efficient segment Assignment StrategY).

3) Putting it all together: We are now ready to define our cost function. For a worker w with current set of segments S_w at time t, the EASY cost is:

$$cost(w, t) = \sum_{s \in S_w} load_s(t) \quad (6)$$

Finally, for an incoming segment at time t, EASY selects the r workers with lowest $cost(w, t)$ for placement.

C. Implementing EASY on Pinot

We implement EASY in Java for integration with Pinot (also written in Java). On the controller side, we implement EASY SAS with ∼200 lines of code. On the worker side, we implement the EASY RESTless API and Pinot logging extensions with ∼500 lines of code. The API is used to compute the $cost(w, t)$ function at each worker w and return the value to the controller. We record cpu_time for each query via java.lang.management.ThreadMXBean; we verified the correctness of our cpu_time implementation with engineering staff at LinkedIn (when the first author was interning at LinkedIn). We also expose the list of segments being targeted by a query in the final log. The overhead of EASY is negligible in practice, especially since we integrate our logging efforts with the efficient LogFactory class used by LinkedIn in their production Pinot implementation. For reference, we have open sourced our EASY-equipped Pinot implementation [11].

IV. Evaluation

We first describe our experimental setup and evaluation methodology, and then present our evaluation results comparing EASY to Pinot SAS and Druid SAS.

A. Experimental setup

We use 7 servers for our experiments, with 1 controller, 2 brokers, and 4 worker nodes. All servers are identical with 4 cores (Intel Xeon CPU E3-1231) and 16GB of memory (of which 12GB is assigned to Pinot Java processes). Servers are connected through 1GB network links.

Data store. The data on worker nodes is divided into tables, and each table has its own segments; each segment is made up of rows and columns, with each row corresponding to information for a given time period. To mimic the LinkedIn functionality, we create the following (self-explanatory) tables: ProfileView, JobApply, and ArticleRead. The total number of rows for ProfileView, JobApply, and ArticleRead are around 2.7M, 1.8M, and 0.9M, respectively. Each table has several columns; for example, ProfileView has columns: Time, ViewerProfileId, ViewerWorkPlace, WereProfilesConnected, etc.

Workload and benchmark. We implement a query generator benchmark for Pinot based on our tables. For each table, we create several relevant queries. An example query for the ProfileView table is "SELECT * FROM ProfileView WHERE

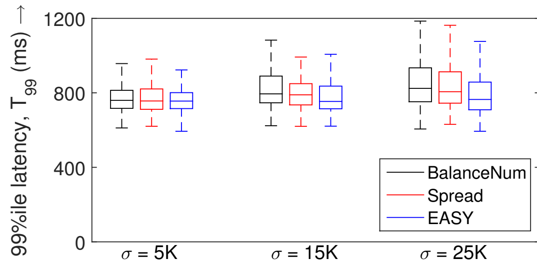


Fig. 4. Boxplot illustrating the T_{99} for different SAS as a function of increasing standard deviation of segment size (σ). For $\sigma = 5K, 15K$, and $25K$, EASY reduces T_{99} by 1%, 5%, and 6% when compared to BalanceNum and by 1%, 4%, and 5% when compared to Spread.

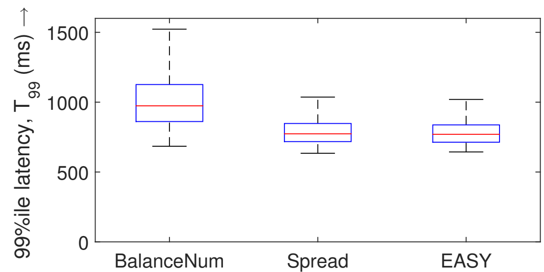


Fig. 5. Boxplot illustrating the T_{99} under different SAS for the scenario where a worker node is added. EASY reduces T_{99} by 21.55% and 1.61% when compared to BalanceNum and Spread, respectively.

ViewStartTime $> t_1$ AND ViewStartTime $< t_2$", where t_1 and t_2 are (randomized) query parameters. Every query requests data from a table with time range length (based on WHERE clause) being Zipf distributed and end time being the wall clock time when the query is issued.

Our benchmark is implemented in ∼2000 lines of code and schema files. The table and query design is guided by our understanding of the Pinot system used by LinkedIn (based on the first author's internship at LinkedIn). All implementation details, including code, tables, and queries, have been open sourced for reference [11].

B. Evaluation methodology

Metrics. We evaluate SAS in terms of two metrics:
(i) $\mathbf{T_{99}}$: 99%ile query tail latency as seen by the broker(s), a metric that LinkedIn uses internally [10]; and
(ii) $\mathbf{CPU_\sigma}$: standard deviation of the CPU usage across workers, a metric we aim to minimize to, in turn, reduce T_{99}.

Baselines. We compare EASY with the following SAS:
(i) **BalanceNum**: This default Pinot SAS aims to balance the number of segments across workers. An incoming segment is assigned to the worker with the least number of segments.
(ii) **Spread**: This is the Druid SAS in use at Metamarkets which aims to avoid hotspots by spreading apart segments that are closer in time as they are likely to be queried together [5]. For segments X and Y, Spread defines:

$$cost(X, Y) = \int_{x_0}^{x_1} \int_{y_0}^{y_1} e^{-\lambda |x-y|} \, dx \, dy, \quad (7)$$

where $[x_0, x_1)$ and $[y_0, y_1)$ is the time range of X and Y, respectively, and λ is the decay rate. For an incoming segment X, Spread selects the worker k which results in minimum $\sum_{y \in S_k} cost(X, Y)$, where S_k is the set of segments on k. The intuition behind this cost function is to place X at a worker that does *not* contain too many segments which are likely to be queried together with X (have neighboring time ranges) to minimize contention.

C. Results

We illustrate evaluation results under various scenarios. In each case, we use normalized cpu_time and row_scan information about segments from the past interval (one hour) to guide the SAS, as described in Section III-B.

SAS for different segment size variability. We first consider a scenario where 90 segments (for 90 days of data) are assigned to four worker nodes via the specified SAS. We then run our benchmark and generate queries over these 90 segments for the next 30 minutes. This experiment uses the ProfileView table; segment sizes (row count) are Normally distributed with mean $\mu = 30K$ and varying standard deviation, σ.

Figure 4 shows the boxplot (including median and first and third quartiles) for our experimental results for T_{99} under BalanceNum, Spread, and EASY. We find that EASY reduces T_{99} moderately by around 1-6% when compared to BalanceNum and Spread. The improvement is larger for higher variability in segment sizes. This is to be expected as BalanceNum and Spread do not explicitly take segment size into account, while EASY implicitly takes the segment size into account when learning the load contributions of segments (see Section III-B).

SAS when adding workers. We next consider the more challenging scenario where a new worker node is added to scale capacity and accommodate new segments. Specifically, we start with three worker nodes which are assigned 60 segments via their SAS. Then, a fourth worker node is added and 30 new segments are assigned (across all workers). We monitor query latencies from this point onwards for the next 30 minutes. This experiment uses the ProfileView table; segment sizes are Normally distributed with $\mu = 30K$ and $\sigma = 1K$.

Figure 5 shows our experimental results for T_{99} under BalanceNum, Spread, and EASY. We find that EASY reduces T_{99} by 21.55% and 1.61% when compared to BalanceNum and Spread, respectively. Likewise, EASY improves query throughput (not shown) by 13.38% and 1.04% when compared to BalanceNum and Spread, respectively. Finally, EASY reduces CPU_σ by 18.38% and 3.51% when compared to BalanceNum and Spread, respectively.

The above results show that the improvement afforded by EASY over BalanceNum is significant. This is because BalanceNum assigns most of the 30 new segments to the fourth (empty) worker node, resulting in a hotspot as newer segments are queried more often. By contrast, both EASY and Spread take recency of segments into account, thus providing better load balancing.

SAS with multiple tables. We now experiment with segments from all three tables (see Section IV-A). We assign 28 segments (for the month of February) for each table to

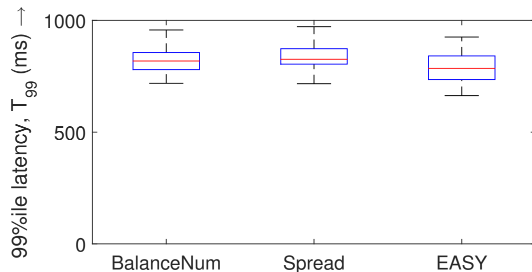


Fig. 6. Boxplot illustrating the T_{99} under different SAS for the case of multiple tables. EASY reduces T_{99} by 4.93% and 6.33% when compared to BalanceNum and Spread, respectively.

4 workers; assignment follows the specified SAS. We assign segments chronologically – segments for a given day for all tables, and then segments for the next day for all tables.

BalanceNum tries to balance the number of segments for *each* table across workers. Spread considers segments from all tables on a worker node, but assigns a higher cost in Eq. (7) (by a factor 2×) if a pair of segments belong to the same table as they are then more likely to be queried together [5]. EASY does not use a pair-wise cost function (as in Spread), and easily extends to the case of multiple tables by considering segments from all tables on a worker ($s \in S_w$ in Eq. (6) can be from any table) when computing the cost for a worker.

Figure 6 shows our experimental results for T_{99}. This time, EASY reduces T_{99} by 4.93% and 6.33% when compared to BalanceNum and Spread, respectively. These results show that EASY affords moderate improvements over Spread as well. Spread performs poorly in this case as it does not take into account the relative difference in the load contributed by segments of different tables. That is, segments of different tables with the same age are treated equally, even though they may induce different loads on the workers due to differences in their structure and content as well as incoming query rate and pattern. By contrast, EASY learns these differences over time and thus treats segments from different tables differently.

V. Related Work

We now discuss related work apart from those already discussed in Section II-B. Curino et al. [3] propose a resource estimation technique to better consolidate multiple online data processing workloads on physical servers. However, they do not take the time range of data into account, which is an important factor in accurately estimating segment load.

Wong et al. [16] consider the subset of segments required to service a relational database query, and use this information to consolidate segments onto servers. However, under Pinot, since segments are created over time, the subset of segments required by a query changes dynamically.

Ozmen et al. [12] address the problem of generating an optimized layout for a given set of database objects by formulating it as a non-linear program. The resulting layout both balances load and avoids interference. By contrast, EASY's approach is much more efficient and only relies on load and popularity estimates, which can be easily obtained.

Pinot partitions data based on timestamps as queries are expected to apply to a particular range of time. This is not the case for general OLAP where all dimensions may have equal importance. VOLAP [4] migrates data shards among OLAP workers to reduce load imbalance.

There are also related works that address the problem of tenant placement in Database-as-a-Service deployments (e.g., STeP [15] and Pythia [6]) or placement of different databases across servers (e.g., Schaffner et al. [14]). While similar, the SAS problem is distinguished by the concept of time-ranged segments which complicates the load distribution challenges.

VI. Conclusion

We present EASY, an efficient SAS (Segment Assignment Strategy) for OLAP systems, such as Pinot [13] and Druid [17]. The key idea in EASY is to model the cpu time contribution of each segment, and leverage this modeling to predict the future load induced by segments of a server. Experimental results show that SAS based on our accurate model predictions provides significantly lower query tail latencies when compared to the SAS of Pinot and Druid.

Acknowledgements

This work was supported by NSF CNS grants 1617046, 1622832, 1717588, 1730128, and 1750109.

References

[1] ClickHouse — Open Source Distributed Column-oriented DBMS. https://clickhouse.yandex.

[2] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *SIGMOD'88*, pages 99–108, 1988.

[3] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware Database Monitoring and Consolidation. In *SIGMOD'11*, pages 313–324, 2011.

[4] F. Dehne, D. Robillard, A. Rau-Chaplin, and N. Burke. VOLAP: A Scalable Distributed System for Real-time OLAP with High Velocity Data. In *IEEE Cluster'16*, pages 354–363, 2016.

[5] Distributing Data in Druid at Petabyte Scale. https://metamarkets.com/2016/distributing-data-in-druid-at-petabyte-scale.

[6] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *SIGMOD'13*, pages 517–528, 2013.

[7] P. Furtado. Experimental Evidence on Partitioning in Parallel Data Warehouses. In *DOLAP'04*, pages 23–30, 2004.

[8] M. Ghosh, L. Xu, X. Qian, T. Kao, I. Gupta, and H. Gupta. Getafix: Workload-aware Distributed Interactive Analytics. Technical report, University of Illinois Urbana-Champaign, 2016.

[9] A. Khandelwal, R. Agarwal, and I. Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *NSDI'16*, pages 485–500, 2016.

[10] Who Moved My 99th Percentile Latency? https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency.

[11] PACELab/pinot. https://github.com/PACELab/pinot.

[12] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware Storage Layout for Database Systems. In *SIGMOD'10*, pages 939–950, 2010.

[13] Pinot — A Realtime Distributed OLAP Datastore. https://github.com/linkedin/pinot.

[14] J. Schaffner, D. Jacobs, T. Kraska, and H. Plattner. The Multi-Tenant Data Placement Problem. In *DBKDA'12*, 2012.

[15] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. DeWitt. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *SOCC'16*, pages 388–400, 2016.

[16] E. Wong and R. H. Katz. Distributing a Database for Parallelism. In *SIGMOD'83*, pages 23–29, 1983.

[17] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A Real-time Analytical Data Store. In *SIGMOD'14*, pages 157–168, 2014.