CASA: Classification-based Adjusted Slot Admission Control for Query Processing Engines

Tim Zeyl
Cloud BU
Huawei Technologies
Markham, Canada
timothy.zeyl@huawei.com

Harshwin Venugopal
Cloud BU
Huawei Technologies
Markham, Canada
harshwin.venugopal@huawei.com

Calvin Sun
Cloud BU
Huawei Technologies
Markham, Canada
calvin.sun3@huawei.com

Paul Larson
Cloud BU
Huawei Technologies
Markham, Canada
paul.larson@huawei.com

***Abstract*—We propose CASA, a classification-based admission control strategy for use in distributed query processing systems. CASA uses coarse-grained predictions of CPU and memory utilization of incoming queries, based solely on the query text, and maps these predictions to an adjusted number of *slots* that the query will occupy; queries with large resource consumption will occupy more slots than queries with small resource consumption. CASA's mapping of predictions to slots enables automatic throttling of query admission, reducing contention and improving throughput and latency. At the core of CASA is a novel feedback mechanism that controls the magnitude of slot adjustments, allowing increased query queueing when workload increases or when the predictor is biased to under-predict, and increased query admission when the workload is light. We compare CASA to default admission control in a production query processing engine and find CASA improves throughput by 48%, reduces tail latency by 50.6%, and eliminates failed queries.**

***Index Terms*—admission control, query processing, prediction**

I. Introduction

Resource contention in query processing systems is costly and inefficient. It is detrimental for both users, who have to wait longer for compute jobs, and for providers, who must provision more resources to avoid contention, which can lead to idle resources when workloads are light.

Workload scheduling and admission control are two ways contention can be mitigated without over-provisioning. In query processing systems, workload scheduling refers to the assignment of tasks, stages, or operators to workers with available capacity. This assignment can be done for individual queries irrespective of other queries currently running in the system (as in [1], [2]), or in a centralized manner where a global scheduler selects the appropriate work unit from any running query to be assigned next (as in [3], [4]). Centralized scheduling may have better potential to reduce contention, but is necessarily more complex.

Admission control is a simpler technique for addressing contention, complementary to workload scheduling. In admission control, an arriving query is either admitted, queued or failed based on current cluster utilization. Common admission control schemes are largely reactive [1], [2], limiting admission of queries when the current or recent history of the cluster resource usage is high, instead of proactively considering the resource needs of incoming queries. Reactive approaches can fail to account for large resource consumption by incoming queries, which can overload a cluster, causing contention.

Machine learning is a potential solution to predicting the resource needs of incoming queries so they can be incorporated into an admission control strategy. Most predictive admission control solutions apply regression based models which predict the resource usage as a scalar value [5], [6], which can be added to the current resource consumption before comparing to a threshold at which queries should be queued. This allows admission control systems to account for the resource needs of incoming queries, however, this approach can be overly sensitive to model accuracy. For example, if the prediction model is biased to over-predict the resource needs of incoming queries, then fewer queries will be admitted than the cluster can support, leading to idle resources. If the prediction model is biased to under-predict the resource needs of incoming queries, then too many queries may be admitted, leading to contention.

In this paper, we propose CASA, a classification-based adjusted slot admission control strategy for query processing engines that is designed to reduce sensitivity to prediction model bias. CASA makes use of a simple ordinal classifier as opposed to a regression model to predict a coarse grained resource usage class for each query. Our approach normalizes the predicted class, making the system more robust to model bias, and uses it to make feedback-controlled adjustments to the number of slots each query should occupy, imposing a hard limit on the total number of slots and resulting in appropriate admission control. Under this scheme, queries predicted to use a small amount of resource get smaller slot-adjustments, making it easier for small queries to enter the system relative to large queries. Our focus is on how to make use of predicted query resource demand for improved admission control, rather than improving the predictions themselves.

Our contributions are two-fold:

- We demonstrate that mapping a normalized predicted resource class to an adjusted number of occupied slots can result in improved admission control, throughput and latency in a query processing engine.
- We devise a feedback mechanism that determines the overall level of slot adjustment in real time, considering the current queueing levels to avoid overshoot.

In the remaining sections of the paper we describe related work (§II), existing admission control in a production query processing engine (§III), details of our proposed predictive admission control extension (§IV), followed by experimental analysis (§V) and finish with concluding remarks (§VI).

II. Related work

Query performance prediction: Machine learning has been explored as a solution for query performance prediction and query resource demand prediction, which can be applied to predictive admission control. Recent work in performance prediction has focused on using features derived from the optimized query plan [7], [8] with some work including features from all concurrently running query plans [9]. CASA uses features derived only from the query string so admission control can take place before query optimization.

More recently, several works have moved from using statistical models to deep learning models in performance prediction and for use in query optimization, while still requiring plan-based features [10]–[12]. A plan-based tree-convolutional network was used in [11] to select a query plan by iterative value function evaluation. A similar model architecture was used in [13] to select the best optimizer hint set, making the selected plan more robust to performance regressions. Lero [14] made use of the tree-convolutional architecture to learn pairwise plan rankings, allowing it to select among several plans generated with different cardinality estimates. QueryFormer [12] moved away from the tree-convolutional architecture to a directed acyclic graph-style transformer, and showed this can be used for query optimization as well as cardinality estimation and index recommendation. CASA favors simplicity, speed and robustness, and so employs the prediction model from [16], which uses XGBOOST [15] with features derived only from the query string.

Predictive scheduling and admission control: In addition to applications in cardinality estimation, index recommendation and query optimization, query performance prediction can also be used to guide workload scheduling and admission control. For scheduling, [17] forecasts task-level resource usage for assigning tasks to workers with available capacity; forecasts are made using historical execution statistics for recurrent queries or user estimated resource consumption for new queries. LSched [18] uses a deep learning architecture for scheduling operators across all running queries and selecting thread counts for each running query, but instead of manifesting an intermediate prediction of query performance, they use this model as an encoder to make scheduling decisions directly.

For admission control, Q-cop [5] estimates the expected runtime of an incoming query and rejects it if that will push the average query response time beyond a threshold. Redshift [19] uses XGBOOST as a regression model with plan-based features to predict query resource requirements, which are then used to decide the queue to which a query should be routed. Queries with short predicted durations are executed on dedicated resources while other queries enter a queue to be placed in an available slot on either the main or a concurrency scaling cluster. An admitted query will occupy one slot, provided it has sufficient resources with respect to the predicted demand, and an adaptive algorithm is used to adjust the total number of available slots. CASA takes a different approach of dynamically adjusting the number of slots each query occupies subject to its predicted resource demand. CASA also uses normalized predicted ordinal classes as opposed to a regression model to reduce the sensitivity to model bias. Indeed, we adopt the model described by [16], who propose using the predicted resource classes for routing queries between clusters, but do not provide details on the routing logic.

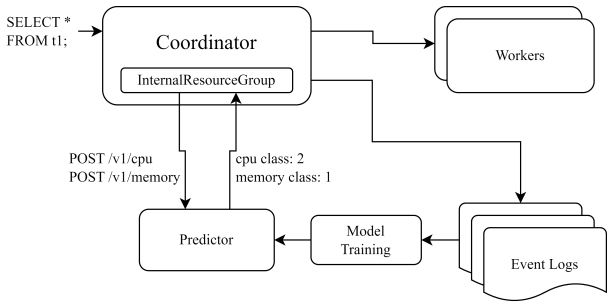

Fig. 1. The admission control logic of CASA is integrated directly into the InternalResourceGroup class, which fetches predictions over HTTP from a prediction server. Prediction models are currently trained offline on data extracted from event logs.

III. Admission control in Presto

We integrated our admission control system (Fig. 1) into a version of Presto[1] [1], a distributed SQL query processing engine that can operate on multiple data sources, typically serving OLAP workloads.

Admission control in Presto is achieved through configured limits on the resource group to which the query is sent. Resource groups can form a hierarchy to achieve desired cluster sharing across multiple tenants within a single organization. Each resource group monitors the number of running queries, CPU usage, and takes periodic snapshots of instantaneous memory usage. Configurable parameters are the `hardConcurrencyLimit`, the `hardCpuLimitMillis`, and the `softCpuLimitMillis`. Each of these is an independent limit for the current resource usage, beyond which submitted queries will queue. The resource group also defines a `softCpuLimitMillis`. If the current CPU usage surpasses the `softCpuLimitMillis`, then the `hardConcurrencyLimit` is adjusted down with the aim of admitting fewer queries if CPU usage is high.

There are two main issues with this form of parameterized admission control. First, the resource demands of incoming queries are not accounted for, so the cluster could become

[1]Specifically, we integrated our work into a Huawei-driven fork [20] of Trino [2], itself a fork of Presto. We refer to this baseline as *Presto* throughout this document for simplicity.

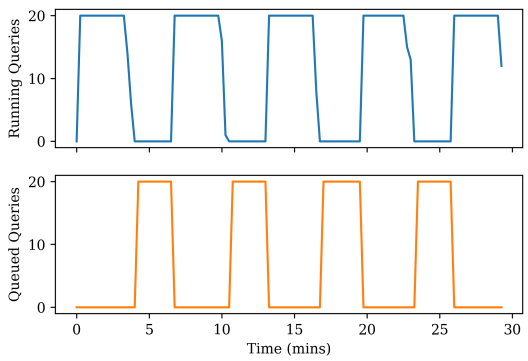

Fig. 2. Baseline admission control in Presto can lead to a sawtooth pattern between queued and running queries leading to idle cluster periods. Due to the the nature of delayed CPU bookkeeping, `cpuUsageMillis` can exceed capacity for long periods causing queries to queue and the cluster to become idle. When `cpuUsageMillis` finally decays enough to admit new queries, too many queries are admitted because their expected CPU usage is not accounted for and `cpuUsageMillis` is relatively low while queries are running. When these running queries begin to complete, `cpuUsageMillis` can be driven up again, causing this sawtooth cycle to repeat itself.

overloaded by admitting a large query. Second, the current memory and CPU usage represent a delayed measure of the current cluster state; this is especially true for CPU. In Presto, a periodic CPU quota is decremented from the current `cpuUsageMillis` and the amount of CPU each query uses is only added to current `cpuUsageMillis` *after* the query finishes. We have observed that this delayed bookkeeping can lead to sawtooth behavior in the admitted queries, making the cluster over-committed in some periods and underutilized in others (Fig. 2).[2]

IV. Predictive Admission Control

A. Predictions

We used the method of [16] for classifying the incoming query into ordinal classes for each resource dimension (CPU and memory) separately. The method uses a term frequency-inverse document frequency (TF-IDF) featurization of the query string with 100 elements, followed by an XGBOOST classifier. We make the simple extension of concatenating the schema name to the query string before featurization to differentiate tables that may have the same name across multiple schemas. We also added tools to derive class boundaries for each resource dimension by dividing training data into quartiles.

There are two key benefits that led us to using this simple classification model. First, it requires only the query string as input; this means that predictive admission control can be done before query parsing, rewriting and cost based optimization.

[2]Recent patches [21], [22] in Trino have remedied this sawtooth behavior by updating `cpuUsageMillis` every 100ms in addition to the time each query finishes, however incoming query resource utilization remains unaccounted for. We port the improved `cpuUsageMillis` bookkeeping patches from Trino to our Presto baseline for a stronger comparison.

When integrating into Presto, we did not need to modify where admission control was performed. Second, the model is both quick to train and, perhaps more importantly, provides low-latency inference, which means there is little additional overhead of using a predictive model.

B. Slot adjustments

We integrated our predictive admission control directly into Presto's resource group. To avoid complications from combining the predicted ordinal CPU class with a delayed measure of current `cpuUsageMillis`, we mapped the predicted class along each resource dimension to an adjusted number of slots that the incoming query should occupy. Instead of occupying one slot, each query can now occupy more than one slot based on its predicted resource demand.

We set the adjusted number of slots, $\widehat{\text{slots}}_q$, that incoming query q occupies to

$$\widehat{\text{slots}}_q = \alpha\left(\frac{1}{2}\frac{l_{\hat{c}_{\text{cpu}}}}{l_{c_{\text{cpu,max}}}} + \frac{1}{2}\frac{l_{\hat{c}_{\text{mem}}}}{l_{c_{\text{mem,max}}}}\right) + 1 \tag{1}$$

where $l_{\hat{c}_{\text{cpu}}}$ is the lower bound of the predicted cpu class, $l_{c_{\text{cpu,max}}}$ is the lower bound of the maximum cpu class, and similar parameters are defined for the memory resource, mem. In this way, slot adjustments are proportional to the normalized lower bound of the query's predicted class, ensuring smaller queries will occupy fewer slots than larger queries. α is a controllable real-valued parameter governing the overall magnitude of slot adjustments. If α is large, slot adjustments will be greater, leading to more throttling of submitted queries. Note that if query q is predicted to be in the lowest class for both CPU and memory, it will occupy exactly one slot, while if q is predicted to be in the highest class on both dimensions, it will occupy $1 + \alpha$ slots. The value of α has a large effect on the overall level of query throttling observed.

We define α in terms of the `hardConcurrencyLimit` and a new hidden parameter `numBigQueries` as

$$\alpha = \frac{\texttt{hardConcurrencyLimit}}{\texttt{numBigQueries}} - 1. \tag{2}$$

The parameter `numBigQueries` corresponds to the number of queries from the largest class that should be allowed into the cluster at the same time (if, theoretically, no other queries were running). This parameter is automatically adjusted through a custom feedback mechanism, which allows CASA to find an appropriate level of query throttling. If unspecified, we initialize `numBigQueries` using the middle of the largest memory class (obtained during model training) and the configured `softMemoryLimit`.

Predictions are fetched from a prediction server over HTTP when a query enters the system, the adjusted slots, $\widehat{\text{slots}}_q$, are computed, then the resource group checks whether this slot adjustment can fit within the configured `hardConcurrencyLimit`. If so, the query is admitted and the total number of slots occupied by running queries is incremented by $\widehat{\text{slots}}_q$. If not, the query queues until there is sufficient slot availability. We allow `hardConcurrencyLimit`

to remain manually configurable so administrators can retain some control over the number of running queries in the system; $\widehat{\text{slots}}_q >= 1$, so at most `hardConcurrencyLimit` queries will be admitted, but CASA may choose to throttle based on cluster conditions. A simple extension could make `hardConcurrencyLimit` automatically configured based on training data.

C. Feedback mechanism

CASA relies on feedback, not only to adapt to changes in prediction model accuracy, but also to respond to workload changes and to determine the overall aggressiveness of query throttling. Feedback tunes the value of `numBigQueries`, which in turn adjusts α, so that the overall number of slots a query occupies reacts with changing conditions. As α is adjusted, the relative number of slots a large query occupies remains greater than the number of slots a small query occupies due to the normalized predicted class terms in (1).

We defined a composite target metric to provide an error signal for feedback. Our feedback error is composed of `taskError`, which attempts to keep the cluster-wide number of tasks at twice the cluster-wide number of processing threads, and `memoryError`, which prevents too much memory from being used. The `taskError` is defined as

$$\texttt{taskError} = \frac{\texttt{numTasks} - \texttt{numThreads}}{\texttt{numThreads}} - 1 \qquad (3)$$

where `numTasks` is the number of tasks associated with all running queries and `numThreads` is the number of processing threads. `taskError` typically ranges between -2 and 4 so we defined `memoryError` to be a piece-wise linear function that ramps from 0 to 2 when current memory usage is between $1/4$ `softMemoryLimit` and `softMemoryLimit`, then from 2 to 4 when current memory usage is between `softMemoryLimit` up to the cluster capacity. Feedback error is a simple sum of `taskError` and `memoryError`, so the goals of feedback are to admit enough queries so the cluster processing threads remain busy, but not overworked, while limiting memory footprint.

When introducing feedback into a system it is important to limit oscillations. We achieve this goal by factoring in the current admission and queuing rates to our adjustment as an admission factor, F_a. For example, if the feedback error is positive and large, there are either too many tasks in the system or too much memory is being used, and queries should be enqueued at a higher rate. If we are already enqueueing many queries relative to the amount we are admitting, we should only make a small adjustment (or none at all) to α because we are already acting in close accordance to what the feedback error is indicating. Whereas, if we are currently admitting a lot of queries, we can make a relatively large adjustment to α so that we take quick corrective action. F_a is formulated to enable this behavior and incorporating it into our feedback mechanism ensures smooth adjustments to α with minimal oscillations. Concretely, when the feedback error is positive, as in the scenario described above, we make the adjustments according to

$$F_a = \frac{R_e}{R_e + R_a + 1} \qquad (4)$$

$$\Delta = e_f F_a \qquad (5)$$

$$\texttt{numBigQueries} \Leftarrow \texttt{numBigQueries} - \Delta \qquad (6)$$

where R_e and R_a are smoothed estimates of the rate we are enqueuing and admitting queries, respectively and e_f is a smoothed version of our composite feedback error. A complementary adjustment is made when feedback error is negative.

V. Experimental analysis

A. Cluster

We ran our experiments on a Presto cluster with one coordinator (24 cores, 16G Java heap size) and two workers (96 cores, 78G Java heap size). We configured a single resource group to achieve admission control with a `hardConcurrencyLimit` of 150, a `softMemoryLimit` of 80%, a `hardCpuLimit` of 192s and a `softCpuLimit` of 96s. This configuration was used both for the baseline Presto admission control and for CASA; `numBigQueries` was determined automatically unless otherwise stated.

B. TPC-H queries

All our workloads make use of TPC-H queries derived from 22 templates with randomized parameter substitution according to the specifications defined in [23]. We apply these queries to five different schemas (sf0.1, sf0.5, sf1, sf5, sf10) supplied by Presto's TPC-H connector, which correspond to five different scale factors of the TPC-H database, where scale factor 1 (sf1) is approximately 1G. In Presto's TPC-H connector, the underlying table data are generated on-demand and in-memory.

C. Model training

TABLE I
Prediction class lower bounds.

Class	CPU	Memory
0	0	0
1	16.9s	20MB
2	74s	56MB
3	8.6min	344MB

It is important to avoid contention when generating training data so that resource demands of queries reflect a scenario where admission control is working properly. To generate training data, we ran 25000 TPC-H queries with randomized parameter substitution and target schema size under low concurrency (≤ 5) to avoid contention. Resource usage data were split into 80% for model training and 20% for model testing. Class boundaries were derived using training data quartiles and are shown in Table I. The maximum CPU observed during

training was 32.8min, and the maximum memory observed during training was 7GB.

After training our XGBOOST predictor, accuracy on the test set was 95.74% for the CPU resource and 95.18% for the memory resource. In this work, we use an offline training process and the same fixed prediction model was used across all experiments. We suggest the prediction model be retrained periodically and automatically in production. Predictor training takes about 10s and when a trained model is deployed using an HTTP interface, the median inference time was found to be 7.4ms while the 99th percentile inference time was 15.97ms.

D. Production trace

In this experiment we applied our predictive admission control on a workload trace derived from one tenant in our cloud service based on Presto. To maintain confidentiality of the tenant's sql queries, we had access to query resource usage and start and stop times, but not the actual query string. As such, we mapped each of the 4999 production queries to TPC-H queries from our training data based on shortest Euclidean distance in terms of CPU and memory usage. This allowed us to build a schedule with unique TPC-H template and scale factors for each query, from which we could generate random parameter substitutions during testing. This trace was submitted to our cluster at eight times speed to provide enough stress that admission control would be required.

Results are shown in Table II, which compares CASA to the baseline Presto admission control across all experiments. For the production trace experiment, CASA achieves the best metrics (shown in bold) for elapsed time (in seconds), throughput (in queries per second), and the 99th percentile tail query latency (in seconds). Throughput increased by 48% and tail latency was reduced by 50.6% when using CASA compared to Presto admission control. We also noticed that a considerable number of queries failed when using Presto, which we attribute to an overloading of the worker nodes and long periods of garbage collection causing communication with the coordinator to time out, resulting in query failures. No query failures were observed when using CASA.

We also compared the mean memory used as tracked by the root level resource group and found that CASA reduced the memory utilization by 47.4% on average. Total cumulative CPU hours for the workload are also shown in Table II. In this case, the baseline admission control used less overall CPU time, but note that CASA may have actually done more work given the 445 failed queries with Presto.

E. Dynamic table sizes

Next, we also tested our system under the condition of dynamically changing table sizes to ensure feedback could react to the condition where the the predictor would be intentionally biased. We modulated the size of each schema of the TPC-H connector according to a sinusoid with a period of one hour and uniform noise. Each schema scale factor was modulated with a unique sinusoid amplitude (A) and zero centered uniform noise amplitude (n) as follows:

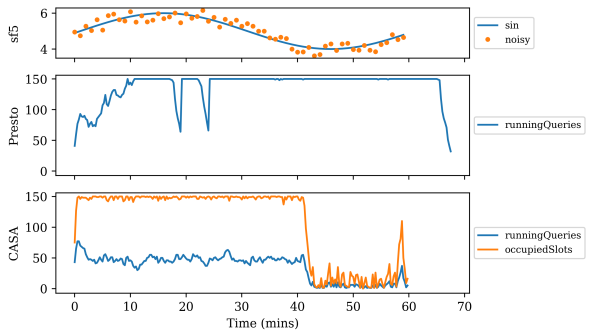

Fig. 3. The number of running queries and total slot adjustments (occupiedSlots) for Presto (middle) and CASA (bottom) during the dynamic table sizes experiment. CASA throttles queries during the early portion of the noisy sinusoidal workload (example schema sizes for sf5 are shown in top panel) when table sizes are larger by making relatively large total slot adjustments. As the table sizes decrease in the latter portion of the sinusoidal workload, and after any queued queries are completed, queries begin to have a quick turnaround time and the cluster has relatively low load. Presto is not able to throttle the admission of queries appropriately, leading to a full cluster for the entire duration of the workload and longer makespan.

- sf0.1:$\{A = 0.01,\ n = 0.05\}$
- sf0.5:$\{A = 0.1,\ n = 0.1\}$
- sf1:$\{A = 0.3,\ n = 0.4\}$
- sf5:$\{A = 1,\ n = 1\}$
- sf10:$\{A = 1,\ n = 1\}$

The base schema size string was concatenated as usual to the query string as input to our classifier, so the classifier would have no input features indicating the fluctuating table sizes, resulting in intentional model bias[3]. We then submitted our randomized TPC-H queries according to a Poisson process with constant rate of 1.0 queries per second (QPS) over a duration of one hour.

With CASA, the tail latency was reduced relative to Presto by 68.4% (see Table II) and by throttling query admission in the first half hour, the workload was able to keep up with the sinusoidal pattern of the fluctuating table sizes (Fig. 3). Presto fell behind, so its makespan lasted much longer than the 3600 second period over which queries were being submitted. CASA also used less memory on average and required fewer cumulative CPU hours to complete the workload, even though Presto had failed queries, so may have done less work overall.

F. Stepped query submission rate

To ensure that feedback would be able to handle changing loads we tested a workload with a time-varying submission rate. We submitted randomized TPC-H queries and schema sizes according to a Poisson process with average rate of 0.8 QPS for 5 minutes, then 1.6 QPS for 5 minutes, then back to 0.8 QPS for 5 minutes. The baseline Presto admission control was able to handle the initial average 0.8 QPS submission rate

[3]Even the base schema size strings do not convey any notion of size to the model, which interprets these schema names as strings. Instead, the model might learn that the string "sf10" is associated with larger query resource usage. In production, the schema name need not include any indication of data size.

TABLE II
COMPARISON OF PRESTO AND CASA ADMISSION CONTROL ACROSS THREE EXPERIMENTS.

	Production trace		**Dynamic tables**		**Stepped rate**			
	Presto	*CASA*	*Presto*	*CASA*	*Presto*	*CASA*	*CASA (nBQ=1)*	*CASA (nBQ=150)*
Elapsed time (s)	5428	**3436**	4076	**3609**	1331	**1062**	1066	1139
Throughput (qps)	0.98	**1.45**	0.88	**1.01**	0.61	**0.91**	0.90	0.85
Tail latency (s)	858	**423**	841	**266**	341	253	**247**	351
Failed queries	445	**0**	32	**0**	150	**0**	**0**	**0**
Successful queries	4554	**4999**	3603	**3635**	814	**964**	**964**	**964**
Mean Memory (GiB)	85.20	**44.87**	66.68	**14.01**	57.76	22.23	**19.42**	30.35
Cumulative CPU[a] (h)	143.86	160.18	180.21	157.55	47.26	49.72	49.20	52.69

[a]We do not embolden the "best" CPU as Presto contained failed queries, rendering total work incomparable.

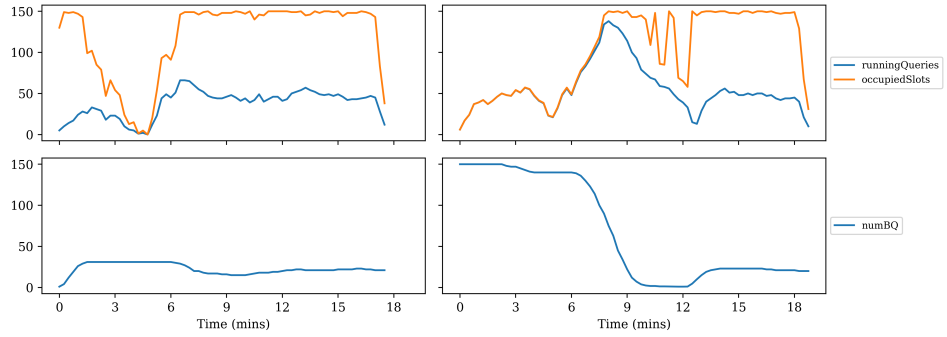

Fig. 5. We show the dynamically changing `numBigQueries` (bottom) along side the number of running queries and total cluster slot adjustments (top) for two poor manual initializations of `numBigQueries` when using CASA. When `numBigQueries` is initialized to one (left), the relative slot adjustment to running queries at the workload start is too large; feedback detects that more queries should be admitted and because the cluster starts out in a state where many queries would be enqueued, `numBigQueries` quickly ramps up to a larger value, admitting more queries. As the workload increases at the five minute mark, feedback then lowers `numBigQueries` to limit contention. When `numBigQueries` is initialized to 150 (right), `numBigQueries` stays relatively constant during the first 5 minutes because the workload is not heavy enough for feedback to detect that this value is too admissive. After the 5 minute mark, the workload increases and `numBigQueries` quickly decreases to throttle the number of newly admitted queries.

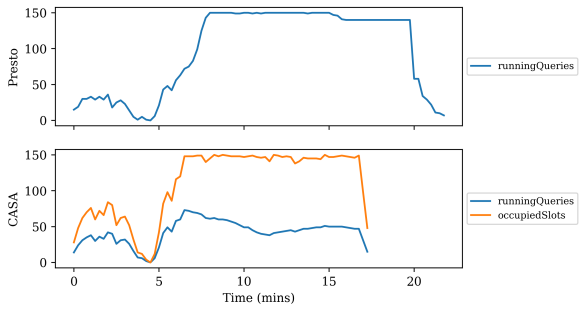

Fig. 4. The number of running queries and total slot adjustments (occupiedSlots) for Presto (top) and CASA (bottom) during the stepped rate experiment. During the first five minute low-load period of the workload, Presto and CASA admit roughly the same number of queries. Thereafter the total slot adjustments for CASA increase causing the number of admitted queries to be throttled. Presto admits up to the `hardConcurrencyLimit` (150) which causes increased query latency and increased workload makespan.

(Fig. 4) but admitted too many queries when the rate stepped to 1.6 QPS, causing contention and increased latency for all admitted queries. CASA, in contrast, restricted the number of admitted queries when the submission rate stepped up to 1.6 QPS, which increased the throughput by 49.2%, reduced tail latency by 25.8%, reduced average memory usage by 61.5%, and reduced makespan by 20.21% relative to Presto (Table II).

G. Effect of poor initialization

Finally, we tested whether feedback could recover from a poorly initialized value of `numBigQueries` (and therefore α). We repeated our stepped query rate experiment (§V-F) where `numBigQueries` was either initialized to a value of one, meaning we should only admit a single query if it was predicted to be the largest class on both resource dimensions, or initialized to the `hardConcurrencyLimit` (150), in which case no adjusted slot throttling would be performed. Fig. 5 shows that `numBigQueries` can be corrected to a good value using our feedback mechanism quickly and without much oscillation. Table II confirms that even poor initializations of `numBigQueries` (shown as nBQ=1 and nBQ=150) outperform the baseline Presto admission control.

VI. CONCLUSION

Admission control is a simple technique to reduce contention in query processing systems, but it must consider the resource needs of incoming queries to be successful. We presented CASA, a predictive classification-based adjusted slot admission control strategy that is robust to prediction model bias and can react quickly and appropriately to dynamic workloads. CASA's simple feature space allows it to be

integrated into any query processing system before query planning takes place. We demonstrated that mapping coarse-grained predictions of CPU and memory utilization to an adjusted number of occupied slots is an effective approach to predictive admission control that is robust to changing table sizes, workload volume, and misconfiguration and results in improved throughput and accuracy compared to Presto. We showed that even with a biased classification model, CASA outperformed Presto, suggesting our feedback mechanism can successfully reduce the need for model retraining; an in-depth analysis of the relative importance of an accurate model compared to an effective feedback mechanism is left for future work. On a production-derived workload trace, CASA improved throughput by 48% and reduced tail latency by 50.6%.

REFERENCES

[1] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte *et al.*, "Presto: SQL on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1802–1813.

[2] Trino. [Online]. Available: https://trino.io/

[3] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min *et al.*, "Dremel: A decade of interactive SQL analysis at web scale," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3461–3472, 2020.

[4] J. Aguilar-Saborit, R. Ramakrishnan, K. Srinivasan, K. Bocksrocker, I. Alagiannis, M. Sankara, M. Shafiei, J. Blakeley, G. Dasarathy, S. Dash *et al.*, "POLARIS: the distributed SQL engine in Azure Synapse," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3204–3216, 2020.

[5] S. Tozer, T. Brecht, and A. Aboulnaga, "Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 2010, pp. 397–408.

[6] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig *et al.*, "Amazon redshift re-invented," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2205–2217.

[7] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 592–603.

[8] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based query performance modeling and prediction," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 390–401.

[9] J. Duggan, O. Papaemmanouil, U. Cetintemel, and E. Upfal, "Contender: A resource modeling approach for concurrent query performance prediction," in *Proceedings of the 17th International Conference on Extending Database Technology*, 2014.

[10] R. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1733–1746, 2019.

[11] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1705–1718, jul 2019.

[12] Y. Zhao, G. Cong, J. Shi, and C. Miao, "Queryformer: a tree transformer model for query plan representation," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1658–1670, 2022.

[13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1275–1288.

[14] R. Zhu, W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, and J. Zhou, "Lero: A learning-to-rank query optimizer," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1466–1479, 2023.

[15] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SigKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[16] C. Tang, B. Wang, Z. Luo, H. Wu, S. Dasan, M. Fu, Y. Li, M. Ghosh, R. Kabra, N. K. Navadiya *et al.*, "Forecasting SQL query cost at Twitter," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 154–160.

[17] C. Cayiroglu, Z. Zhuang, B. Chattopadhyay, P. Pandian, and C. Li, "Presto pack: Resource-aware workload placement for higher performance and reliability," in *2022 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 2022, pp. 267–274.

[18] I. Sabek, T. S. Ukyab, and T. Kraska, "LSched: A workload-aware learned query scheduler for analytical database systems," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1228–1242.

[19] G. Saxena, M. Rahman, N. Chainani, C. Lin, G. Caragea, F. Chowdhury, R. Marcus, T. Kraska, I. Pandis, and B. M. Narayanaswamy, "Auto-wlm: Machine learning enhanced workload management in amazon redshift," in *Companion of the 2023 International Conference on Management of Data*, ser. SIGMOD '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 225–237.

[20] openLooKeng. [Online]. Available: https://openlookeng.io/en/

[21] P. Desai. (2019) Implement periodic cpu usage tracking in resource groups. Trino. [Online]. Available: https://github.com/trinodb/trino/pull/1128

[22] Lars. (2020) Reduce idle cpu consumption in the resource group manager. Trino. [Online]. Available: https://github.com/trinodb/trino/pull/3990

[23] TPC Benchmark H (TPC-H). [Online]. Available: http://www.tpc.org/tpch/