Database-Agnostic Workload Management

Shrainik Jain
University of Washington
shrainik
@cs.washington.edu

Jiaqi Yan
Snowflake Computing
jiaqi
@snowflake.com

Thierry Cruanes
Snowflake Computing
thierry.cruanes
@snowflake.com

Bill Howe
University of Washington
billhowe
@cs.washington.edu

ABSTRACT

We present a system to support generalized SQL workload analysis and management for multi-tenant and multi-database platforms. Workload analysis applications are becoming more sophisticated to support database administration, model user behavior, audit security, and route queries, but the methods rely on specialized feature engineering, and therefore must be carefully implemented and reimplemented for each SQL dialect, database system, and application. Meanwhile, the size and complexity of workloads are increasing as systems centralize in the cloud. We model workload analysis and management tasks as variations on query labeling, and propose a system design that can support general query labeling routines across multiple applications and database backends. The design relies on the use of learned vector embeddings for SQL queries as a replacement for application-specific syntactic features, reducing custom code and allowing the use of off-the-shelf machine learning algorithms for labeling. The key hypothesis, for which we provide evidence in this paper, is that these learned features can outperform conventional feature engineering on representative machine learning tasks. We present the design of a database-agnostic workload management and analytics service, describe potential applications, and show that separating workload representation from labeling tasks affords new capabilities and can outperform existing solutions for representative tasks, including workload sampling for index recommendation, user labeling for security audits and error prediction.

1. INTRODUCTION

Extracting patterns from a SQL query workload has enabled a number of important features in database systems, including workload compression [3], index recommendation [2], modeling user and application behavior [31, 9, 35], query recommendation [1], predicting cache performance [29, 5], and designing benchmarks [35]. These techniques can be used as part of a more comprehensive approach to automate database administration [26].

However, the diversity of applications have led to a diversity of solutions, each relying on specialized feature engineering. For example, workload summarization for index recommendation uses the structure of join and group by operators as features [3], query recommendation may pre-process a query into fragments before making recommendations [13], and security audits may require user-defined functions to enforce particular policies [32].

In fact, the features and the algorithms to extract them tend to be the significant contributions in the papers in this space. But the state of the art in a variety of applications is to learn features automatically. For instance, Natural Language Processing applications previously relied on parsing and labeling sentences as a pre-processing step, but now use learned vector representations almost exclusively [6, 28]. This approach not only obviates the need for manual feature engineering and pre-processing, but also has the potential to significantly outperform more specialized methods.

We see three trends motivating an analogous role for generalized workload representations. First, workload heterogeneity is increasing, making it difficult to maintain SQL parsers and feature extraction routines. The number of SQL-like languages is increasing, with inconsistent support and syntax for even relatively common features such as outer joins. Second, workload scale is increasing. Cloud-hosted, multi-tenant database services including Redshift [8], Snowflake [4], BigQuery [23] and more receive millions of queries daily from thousands of customers using hundreds of schemas; relying on brittle parsers (or worse, manual inspection) to identify query patterns that influence administration decisions is no longer tenable. Third, new use cases for centralized workload management are emerging. For example, SQL debugging [7], database forensics [27], and data use management [32] motivate a more automated analysis of user behavior patterns, and cloud-hosted multi-tenant systems motivate a more automated approach to query routing and resource allocation.

In this work, we propose Querc, a database-agnostic system for mining and managing large-scale and heterogeneous workloads. We model workload management and analysis as a set of query labeling tasks. For instance, workload sampling can be reduced to labeling each query as present or absent in the sample, error prediction involves labeling

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

each query with an error type, query routing involves labeling each query with a cluster resource to which the query should be routed, and so on. Because our framework depends only on the query text (along with typical metadata such as arrival timestamp and userid issuing the query), it can be used with any DBMS and any SQL dialect. In fact, as we will show, features learned with a workload against a particular schema and SQL dialect can be effective even when used with a *different* schema and SQL dialect.

The weakness of this approach is that it requires enormous amounts of data to be effective. But as database products migrate to the cloud, service providers have access to workloads from a large number of customers, potentially even across different database products. Since the input is just the query text, these diverse workloads can be processed as one very large dataset. But the resulting vectors can still be used to train models to support specific applications, as we will show on two representative tasks: workload summarization for index selection, user prediction for security audits and routing, and query error prediction.

2. SYSTEM ARCHITECTURE

Figure 1 illustrates the architecture of Querc. There are three applications, X, Y, Z. Each application has its own database, DB(X), DB(Y), and DB(Z), though these may be logical instances in the same physical multi-tenant service. In this example, DB(X) and DB(Y) are tenants in the same service. Each application is also associated with a separate stream of queries (at left), where query(X,t) indicates a batch of queries arriving for application X at time instant t.

Each application is associated with one Qworker, but each Qworker operates multiple classifiers. Qworkers may not be entirely stateless, as some labeling tasks process a small window of queries. However, the state is assumed to be small such that the Qworkers do not need their own local storage and can be load balanced and parallelized in typical ways. Each classifier is a pre-trained (embedder, labeler) pair. The same trained embedder may be used across multiple applications. This split design is critical, because we want to learn features using a very large, combined workload, but an individual classifier may perform better when trained on an application-specific workload. In this example, application X and application Y both share the same embedder, EmbedderA, trained on the combined X and Y workloads, written EmbedderA(X,Y). This log sharing between customers may not always be permitted by customers for security reasons, and in this example, application Z uses only its own data. But there is some incentive for customers to pool their data as the additional signal can potentially improve accuracy, and some cloud providers support features to allow data sharing between customers.

The Labeler passes the query on to the database, but also transmits the query back to a central training module ("Training, Evaluation, and Offline Labeling" in Figure 1). The training module manages training sets, including the (parallel) execution of training and evaluation routines, then deploys trained models back to Qworkers. There is significant ongoing research in the database, systems, and ML communities on runtime architectures for training and deploying models (e.g., [21]); we do not discuss them further since our requirements are relatively modest.

Since Querc is specialized for query workload analytics

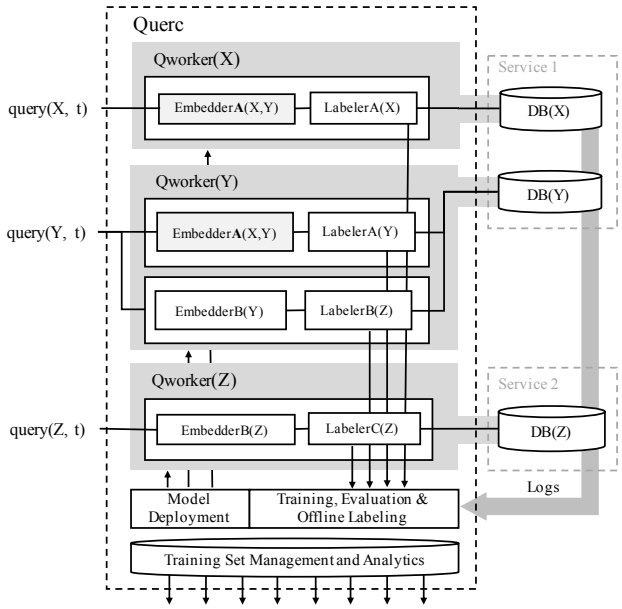


Figure 1: System architecture. Queries arrive for three different applications X, Y, and Z and are processed by one or more (embedder, labeler) pair before being sent on to the database, centralized for offline labeling tasks, or both.

rather than general machine learning, one data model can be shared among most applications. The only messages passed between components are labeled queries. A labeled query is a tuple $(Q, c_1, c_2, c_3, \dots)$ where c_i is a label. This simple model captures situations where a query arrives already equipped with a timestamp, a userid, an IP address, etc., but also captures more verbose query logs that are returned from the database.

The training module also records the queries with their predicted labels for retraining, evaluation, and to support offline analysis tasks. Offline tasks are those that do not require or do not allow processing each query separately, and can be implemented as typical batch jobs. For example, query clustering is important for workload summarization [16], but does not require real-time labeling of individual queries.

Training data is collected periodically from the databases in the form of query logs. These logs are (batched) sequences of labeled queries, but with additional labels to be used for training, such as runtime, memory usage, error codes, security flags, resource IDs. We do not specify the mechanism by which these logs are transmitted from the database to Querc, since most systems have robust means of exporting logs in appropriate forms.

In some applications, Querc may not be in the critical path for query execution to avoid any performance overhead or reduce dependencies. In these cases, queries will be forked to Querc. No change to the architecture is required in this case; queries come in, and labeled queries are collected in the training module. The query is simply not forwarded to the database.

This architecture is not designed for continuous learning, as the training is handled separately from real time query labeling. Not all algorithms can support fully continuous learning, and an important design goal is to support simple

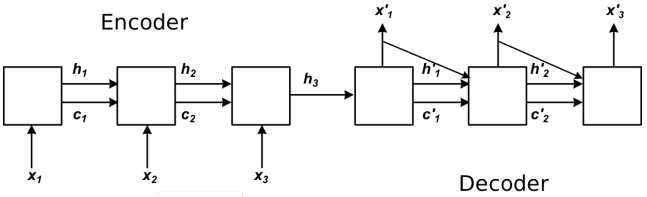


Figure 2: The LSTM Autoencoder network architecture learns to generate the input token in the decoding phase. Once trained, the encoder can be used to output a vector representation for the text of a query.

machine learning algorithms as labelers. Model training is therefore assumed to occur infrequently as a batch job.

3. LEARNING VECTOR REPRESENTATIONS

There are multiple choices for embedders; we describe two initial models we evaluate in this paper:

Context prediction models: Mikolov et al. [25, 24, 17] proposed learning a vector representation for words by predicting the next word in a context, and then deriving a vector representation for larger semantic units (sentences, paragraphs, documents) by adding a vector representing the paragraph to each context as an additional "word." The learned vector for this virtual context word is used as a representation for the entire paragraph. This "Doc2Vec" method has been shown to capture semantic relationships that work well for, say, sentiment classification and clustering tasks [14, 18]. This approach can be applied directly for learning representations of SQL queries: We can use fixed-size context windows to learn a representation for each token in the query, and include an identifier to learn a representation of entire query.

LSTM AutoEncoders: The paragraph vector approach in the previous section is viable, but it requires a hyper-parameter for the context size. There is no obvious way to determine a context size for queries, for two reasons: First, there may be semantic relationships between distant tokens in the query. Second, the length of queries vary widely in ad hoc workloads [12, 10]. To avoid setting a context size, we can use Long Short-Term Memory (LSTM) networks [36], which are modified Recurrent Neural Networks (RNN) that can automatically learn how much context to remember and how much of it to forget, thereby removing the dependency on a fixed context size. LSTMs have successfully been used in sentence classification, semantic similarity between sentences and sentiment analysis [30]. We use a standard LSTM encoder decoder network [37, 20] with architecture as illustrated in Figure 2.

An LSTM autoencoder is trained by sequentially feeding words from the query to the network one word at a time, and then attempting to reproduce the input. The LSTM network not only learns the encoding for the samples, but also the relevant context window associated with the samples. The final output of the encoder network gives us an encoding for the query. Once this network has been trained, an embedded representation for a query can be computed by passing the query to the encoder network, completing a forward pass, and using the hidden state of the final encoder LSTM cell as the learned vector representation.

There are multiple prior approaches in the NLP literature that compare the efficacy of these models and their relative performance [17, 22, 30]. For this paper, we consider context-based models (i.e., doc2vec) and LSTM AutoEncoders.

4. APPLICATIONS

The applications supported by this system reduce to *query labeling*, and general workflow consists of two machine learning models: a representation learner (an embedder) and a classifier. We split the task into two parts to allow the same representation to be used for multiple applications.

Workload summarization for index recommendation: The goal [3, 16] is to find a representative sample of the workload as input to further database administration, tuning, and testing tasks [3, 33]. In particular, workload summarization aids index recommendation, since the recommendation process is typically quadratic in the size of the workload [3]. While index recommendation systems are well-studied and ship with most production databases [3, 2], the quality of the representative sample determines the overall quality of the final recommendations. In Section 5, we show that a simple sampling procedure using learned features delivers a significant runtime improvement over the built-in sampling procedure in the SQL Server database system.

Enforcing query routing policies: Query Routing in a distributed database involves identifying the cluster resources on which to execute the incoming query. The policies that govern these routing decisions may involve customer SLAs, security considerations (e.g., certain applications must use a physically distinct cluster from other applications), auditing requirements (e.g., queries from certain accounts or those accessing certain tables must be logged for auditing purposes). Even in modern cloud-hosted database products such as Snowflake [4] and BigQuery [23], these policies tend to be manually encoded, and management of these policies as they evolve, while maintaining multiple heterogeneous clusters used by thousands of customers, is increasingly perceived as untenable. Under the hypothesis that queries that follow a particular policy tend to have similar features, Querc can help identify policy misconfiguration by detecting when a predicted routing decision differs from the assigned routing decision.

Error prediction: Particular syntax patterns in the workload may be associated with resource errors or bugs in the database system. In a multi-tenant, multi-database, and high-volume scenario, identification of the syntactic patterns that tend to trigger errors, either manually or with scripts, becomes untenable: there may be hundreds of error codes, each with hundreds of subtle patterns that tend to trigger them, across hundreds of tenant schemas. Using learned features, a classifier to predict errors from syntax is trivial to engineer. This prediction allows the query to be routed to a different runtime environment that is instrumented, equipped with more more memory per node, or running a more stable version of the database engine.

In figure 3, we show a clustering of error-generating SQL queries from a large-scale cloud-hosted multi-tenant database system [4]. Color represents the type of error; there are over twenty different types of errors ranging from out-of-memory errors to hardware failures to query execution bugs (the figure highlights three specific error types). The syntax pat-

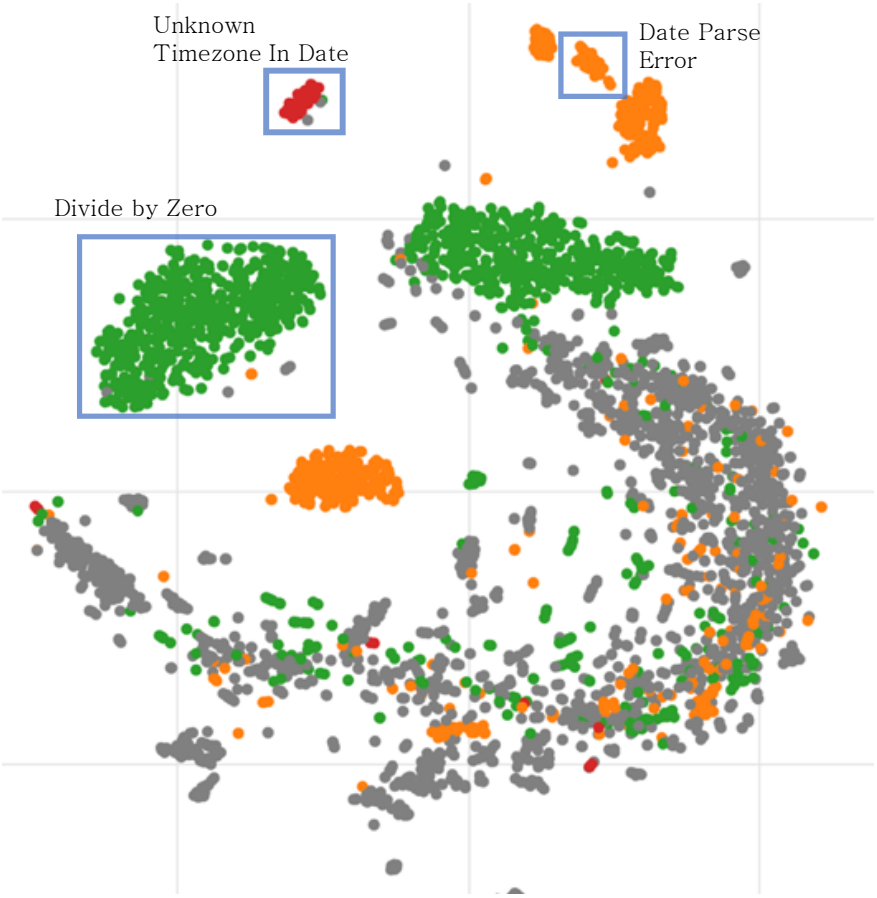


Figure 3: A clustering of error-generating SQL queries from a large-scale cloud-hosted multi-tenant database system. Each point in the plot represents a query. The color represents the type of error (the figure annotates three specific error types). The syntax patterns in the workload are complex as one would expect, but there are obvious clusters some of which are strongly associated with specific error types. The text in the annotated boxes represents the error type query clusters correspond to. For example, the rectangle on the top right corresponds to queries that resulted in an error while parsing an incorrectly formated DateTime field and the cluster with queries annotated in green corresponds to queries that generated divide-by-zero error.

terns in the workload are complex (as one would expect), but there are obvious clusters, some of which are strongly associated with specific error types. For example, the cluster at the upper right corresponds to errors raised when the compiler failed to parse a wrongly formatted DateTime field.

Using an interactive visualization based on these clusterings, the analyst can inspect syntactic clusters of queries to investigate problems rather than inspecting individual queries, for two benefits: First, the analyst can prioritize large clusters that indicate a common problem. Second, the analyst can quickly identify a number of related examples in order to confirm a diagnosis. For example, when we first showed this visualization to our colleagues at Snowflake [4], they were able to diagnose the problem associated with one of the clusters immediately.

Resource allocation: The structure of the query is not sufficient to accurately predict its runtime or memory footprint, but it can provide a hint that can be used for load balancing, scheduling, and as an input for optimization. If we can coarsely categorize queries as memory-intensive, long-running, etc. with some degree of accuracy, these labels can be used as a simple, database-agnostic way to speculatively allocate resources. Training data is readily available from the query logs themselves. We consider this application in a tech report companion to this paper [11] and leave a detailed analysis for future work.

Query recommendation: The query recommendation problem can be modeled as a prediction of the next query the user will submit to the database based on the recent history of queries [1]. This prediction is then shown to the user though an appropriate client application to assist in query authoring. Our framework can generate features that can be used to train query recommendation models. We consider this application in a tech report companion to this paper [11].

Security auditing: To the extent that users' individual workloads tend to follow predictable patterns, an anomalous query may be a sign that a user's account has been compromised. By formulating a prediction problem that tries to guess the user that submitted the query from the syntax alone, we can identify anomalous queries for security audits. In our framework, the labeler is a simple classifier $V \rightarrow user$.

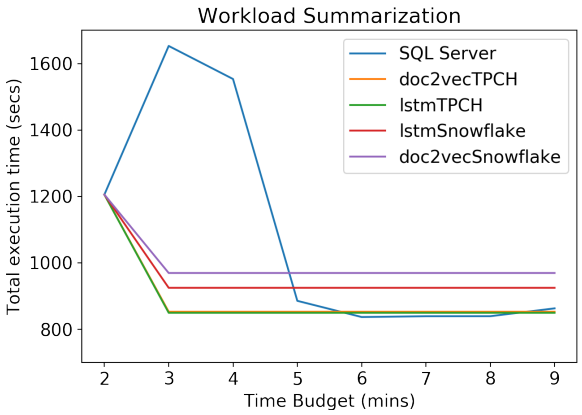


Figure 4: Workload runtime using indices recommended under various time budgets. For most time budgets, the workload summaries improve runtimes, even when the embedders were trained on an unrelated workload (lstmSnowflake and doc2vecSnowflake).

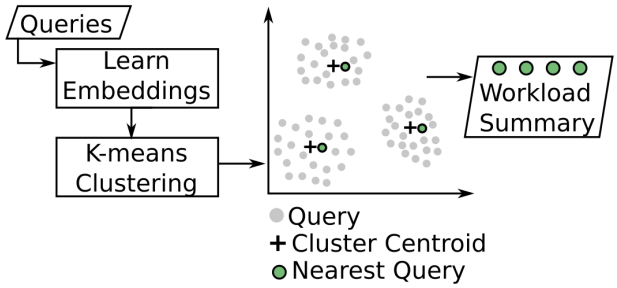


Figure 5: Workload summarization using learned query embeddings.

	Account Labeling	User Labeling
Doc2Vec	78.8%	39%
LSTMAutoencodder	99.1%	55.4%

Table 1: Query Labeling results

5. EXPERIMENTS

We consider two applications: Workload summarization for index selection, and labeling tasks for security audits and query routing.

5.1 Workload Summaries for Index Selection

The workload summarization task (with respect to index recommendation) is to find a subset Q_{sub} of a given query workload Q, such that the set of indices recommended based on Q_{sub} is similar to the the set of indices recommended for the overall workload Q. Previous solutions are primarily variants of the approach of Chaudhuri et al. [3], which uses K-medioids to cluster the queries and selects a witness query from each cluster. However, the authors emphasize that a custom distance function should be developed for specific workloads; our hypothesis is that generic representation learning approaches obviate the need for these custom distance functions.

In the Querc framework, this task is offline and does not require real-time labeling of queries. Instead, we perform the task as an offline unsupervised learning task. In our approach, we assign each query to a vector (using a suitably trained embedder), then simply use K-means to find K query clusters and pick the nearest query to the centroid in each cluster as the representative subset. To determine K, we use an intentionally simple method (the "elbow method" [15]) which runs the K-means algorithm in a loop with increasing K till the rate of change of the sum of squared distances from centroids plateaus. Although better methods exist, we highlight the effect of the learned vectors rather than the choice of K. We present this workload summarization algorithm in Figure 5.

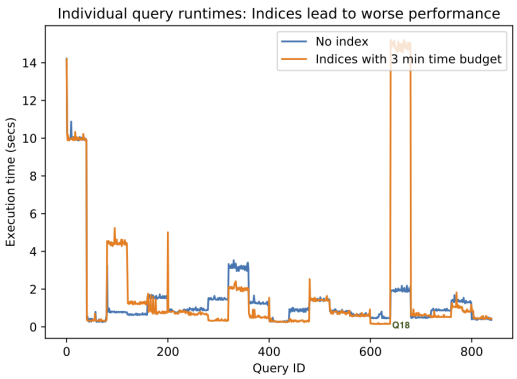
Setup: Following the evaluation strategy of Chaudhuri et al.[3], we first run the index selection tool on the entire workload Q, create the recommended indices, and measure the runtime t_{orig} for the original workload. We then run use the workload summarization algorithm to produce a reduced set of queries $\mathcal{Q}_{sub}$, re-run the index selection tool, create the recommended indices, and again measure the runtime t_{sub} of the entire original workload. We use SQL Server 2016 and the Database Engine Tuning Advisor, which performs its own summarization on the input according to the documentation. We use an $m4.large$ AWS EC2 instance as the server. We use TPC-H with scale factor 1 as the workload for comparison with previous results and to interpret the recommended indices, but we also show how the method performs when trained on a more complex Snowflake workload.

We pass the summarized workload to the tuning advisor, along with a time budget (a parameter supported by the tuning advisor). Each experiment involves clearing caches, generating indices, applying the indices, and running the full workload. We report the time running the workload; the time budget specifies the time limit under which the advisor must return a set of recommendations.

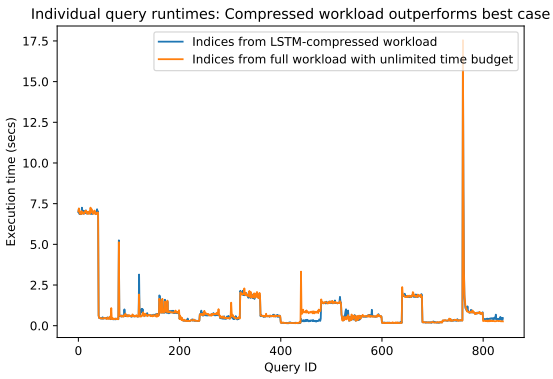
Results: Figure 4 shows the results. The x-axis is the time budget, and the y-axis is the runtime for the entire workload after building the recommended indices. For time budgets less than 3 minutes, the advisor does not produce any index recommendations for any method, and the runtime is constant at 1200 seconds. As we relax the time budget, different sets of indices are recommended, each associated with a separate runtime. The full workload (blue line) varies dramatically with the time budget, and surprisingly it gets worse before it gets better. For the summarized workloads, the workload is small enough that the runtimes are constant: Once three minutes have elapsed, the advisor has found the "optimal" set of indices, and allowing more time does not change the result.

We evaluate four trained embedders: two methods on two workloads. The two methods are Doc2Vec and the LSTMAutoencoder, and the two workloads are TPC-H itself, and a separate workload of 500,000 queries from the Snowflake service. When training the embedder on TPC-H (doc2VecTPCH and lstmTPCH), the advisor finds close-to-optimal indices in about three minutes as opposed to the six minutes the advisor requires on the full workload.

Surprisingly, under tight time budgets, the index recommendations made by the native system can actually *hurt* performance relative to having no indices at all! The optimizer chooses a bad plan based on the suboptimal indices.



(a) Runtime for each query under no indices and under indices recommended with a three-minute time budget. For a few specific queries (all instances of TPC-H Query 18), the presence of a recommended index results in significantly worse performance.



(b) Runtime for each query under our LSTM-based pre-compression scheme and the "optimal" indices recommended by SQL Server. The pre-compressed workload achieves essentially identical performance but with significantly smaller time budgets.

Figure 6: Comparing runtimes for all queries in the workload under different index recommendations.

In Figure 6a and Figure 6b, we show the sequence of queries in the workload on the x-axis, and the runtime for each query on the y-axis. The indices suggested under a 3 minute time budget result in all instance of TPC-H query 18 (queries 640-680 in Figure 6a) taking much longer than they would take when run without these indices. The key conclusion is that *pre-compression can help achieve the best possible recommendations in significantly less time, even though compression is already being applied by the engine itself.*

Transfer Learning: Figure 4 also illustrates the capacity for transfer learning using Querc: When training the embedder on the snowflake dataset — a completely unrelated workload to TPC-H workload in the *SQL Server dialect* — the summarized workload still outperforms native SQL Server for most time budgets. This transfer learning effect allows us to bootstrap new applications without waiting for a representative workload to accumulate, and to avoid having to repeatedly re-implement brittle parsers and feature extractors for each new dialect of SQL we encounter.

5.2 Labeling for Security Audits

We consider the conditions under which the learned features from query syntax are sufficient to predict username and customer account, where each customer has many users. When the predicted username differs from the actual username, we can potentially flag the query for an audit. Predicting username can help flag queries for security audits, account and cluster labels can identify misrouted queries. labels from query syntax using the two embedding methods described in Section 3 over the Snowflake dataset.

Setup: We use embedders pre-trained on 500000 Snowflake queries. The experiment itself is run on another dataset of 200000 Snowflake queries labeled with username, account_id and cluster_name for the cluster that ran the query. Next we train classifiers (randomized decision trees) for username and customer account.

Results: Table 1 shows the results for the labeling experiments. The numbers denote the 10-fold cross validation score on the respective task. We find that LSTM based embedder beats Doc2Vec on all tasks. The LSTM method achieves near perfect accuracy when predicting the customer account, which is because it automatically incorporates signal from the schema, and different customers use primarily different schemas (there are instances of shared schemas, but that is the less common case). The method was completely generic and knows nothing about schemas or queries. For user prediction, the task is more difficult, and the overall accuracy is lower at 55%. Upon further analysis we found that the user labeling task has $> 95\%$ accuracies for a majority of accounts (Table 2). The accounts that had poor accuracies for user labeling had one distinctive property: multiple users running the exact same query, making the users nearly indistinguishable. In the sample of workload that we were working with, there were two accounts that had a number of repetitive queries by different users (for instance, 69% percent of the 74000 queries in an account had more than one user label), and these two accounts also covered around 65% of the total queries, bringing down the overall accuracy of classifiers.

#queries	#users	accuracy
73881	28	49.3%
55333	10	37.4%
18487	46	31.8%
5471	21	96.2%
4213	6	58.5%
3894	12	99.7%
3373	9	99.8%
2867	6	99.8%
1953	15	89.1%
1924	4	98.1%
1776	9	95.2%
1699	5	99.8%
1108	12	98.2%

Table 2: Top accounts with user prediction accuracy.

5.3 Error Prediction

In figure 3 we showed how the syntactic patterns in query

workload correlate with errors. In this section, we perform a more quantitative experiment wherein we train a classifier to predict whether a query will raise an error or not. Such classifiers can be used in production to help quarantine error prone queries and run them on an instrumented debugging platforms.

Classifying multiple errors.

Setup: In this experiment we train a classifier which can predict an error type (or no_error) given an input Snowflake query. We use LSTMAutoencoder based embedder pre-trained on 500000 Snowflake queries. The classifier itself is trained using a dataset of 100000 queries from Snowflake. Next, we randomly split the learned query vectors (and corresponding error codes) into training (85%) and test (15%) sets. We use the training set to train a classifier. We present the performance of this classifier on the test set.

Error Code	Precision	Recall	f1-score	# queries
-1	0.986	0.992	0.989	7464
604	0.878	0.927	0.902	1106
606	0.929	0.578	0.712	45
608	0.996	0.993	0.995	3119
630	0.894	0.864	0.879	88
2031	0.765	0.667	0.712	39
90030	1.000	0.998	0.999	1529
100035	1.000	0.710	0.830	31
100037	1.000	0.417	0.588	12
100038	0.981	0.968	0.975	1191
100040	0.952	0.833	0.889	48
100046	1.000	0.923	0.960	13
100051	0.941	0.913	0.927	104
100069	0.857	0.500	0.632	12
100071	0.857	0.500	0.632	12
100078	1.000	0.974	0.987	77
100094	0.833	0.921	0.875	38
100097	0.923	0.667	0.774	18

Table 3: Performance of classifier trained using query embeddings for different error types (-1 signifies no error).

Results: We summarize the performance of the classifier on all error classes with more than 10 queries each in Table 3. The classifier performs well for the errors that occur sufficiently frequently, suggesting that the syntax alone can indicate queries that will generate errors. This mechanism can be used in an online fashion to route queries to specific resources with monitoring and debugging enabled to diagnose the problem. Offline, query error classification can be used for forensics; it is this use case that was our original motivation.

Although individual bugs are not difficult to diagnose, there is a long tail of relatively rare errors; manual inspection and diagnosis of these cases is prohibitively expensive. With automated classification, the patterns can be presented in bulk.

Classifying out-of-memory errors.

Setup: In this experiment, we compare the classification performance of our method for one type of error considered a high priority for our colleagues — queries running out of memory (OOM). We compare to a baseline heuristic method developed in collaboration with Snowflake based on their knowledge of problematic queries. We use a workload of 4491 Snowflake queries with a mix of queries with and without OOM errors to train a classifier to predict OOM errors. Following the methodology in the previous classification task, we use the pre-trained embedder to generate query representations for the workload, randomly split the learned query vectors into training (85%) and test (15%) sets, and present the performance on the test set.

Heuristic Baselines: We interviewed our collaborators at Snowflake and learned that the presence of window functions or joins between large tables in the queries tend to be associated with OOM errors. We implement four naïve baselines that looks for the presence of window functions or a join between at least 3 of the top 1000 largest tables in Snowflake. The first baseline looks for the presence of heavy joins, the second baseline looks for window functions, and the third baseline looks for the presence of either one of the indicators: heavy joins **or** window functions, and the fourth baseline looks for the presence of both heavy joins **and** window functions. The baselines predicts that the query will run out of memory if the corresponding indicator is present in the query text.

Results: Table 4 shows the results. We find that our method significantly outperforms the baseline heuristics, without requiring any domain knowledge or custom feature extractors. We do find that the presence of heavy joins and window functions in the queries are good indicators of OOM errors (specially if they occur together) given the precision of these baselines, however, the low recall suggests that such hard-coded heuristics would miss a other causes of OOM errors. Querc obviates the need for such hard-coded heuristics. As with any errors, this mechanism can be used to route potentially problematic queries to clusters instrumented with debugging or monitoring harnesses, or potentially clusters with larger available main memories. We see Querc as a component of a comprehensive scheduling and workload management solution; these experiments show the potential of the approach.

Method	Precision	Recall	f1-score
Contains heavy joins	0.729	0.115	0.198
Contains window funcs	0.762	0.377	0.504
Contains heavy joins OR window funcs	0.724	0.403	0.518
Contains heavy joins AND window funcs	0.931	0.162	0.162
LSTMAutoencoder	**0.983**	**0.977**	**0.980**
Doc2Vec	0.919	0.823	0.869

Table 4: Classifier performance for predicting OOM errors.

6. FUTURE WORK

Other methods: There are a variety of other methods for learning representations of text that we do not evaluate in this paper. Our goal is not to identify the best possible representation learning approach but rather to show that these methods can compete with and outperform classical approaches that rely on task-specific heuristics and feature engineering (extracting JOIN clauses, counting the number of attributes, etc.), and to organize the methods into a coherent system architecture.

Alternative methods can be roughly categorized into *non-neural-network* based methods and *neural-network*-based methods. The non-neural-network-based methods, including non-negative matrix factorization (NMF), bag-of-words representations, and LDA [22] have been shown to be less effective than neural-network-based-methods in a variety of contexts [25, 19]. Apart from the methods considered in this paper, there are more recent neural-network-based methods using Convolutional Neural Networks (CNNs) adapted for text data. However, Yin et al. [34] showed that RNN based methods (e.g., LSTMs) perform well and are robust in a broad range of tasks when compared to CNNs. However, we plan to extend the current work to include a rigorous comparison of the techniques not covered in this paper.

Publish pre-trained models: The results in Section 5 demonstrate that the proposed framework in this paper has potential to use pre-trained models on generic workloads to aid analytics for previously unseen query. In future work, we will build this framework as a service which is accessible by third parties. Given the workloads that we have access to from Snowflake [4], such a service could be really beneficial for researchers who do not have access to massive query workloads.

Other applications: We touched upon the potential utility of our framework for a variety applications in Section 4, future work would explore all of the applications that were not a part of experimental evaluation in this paper.

7. CONCLUSIONS

We presented the architecture for Querc, a database-agnostic workload analytics service that captures the structural and schema patterns present in the query workload automatically, largely eliminating the need for the specialized syntactic feature engineering that has motivated a number of papers in the literature. The proposed architecture provides a new way of organizing a variety of database administration and user productivity tasks, and provides a mechanism by which to automatically adapt database operations to specific query workloads. Our evaluation of this architecture showed that our general framework outperformed or was competitive with previous approaches that required specialized feature engineering, and also admitted simpler classification algorithms because the inputs are numeric vectors with well-behaved algebraic properties rather than result of arbitrary user-defined functions for which few properties can be assumed. The use of transfer learning in Querc allows workload analytics to be *SQL dialect independent* and enables the capability to bootstrap new analytics tasks and avoid re-implementing brittle codes paths.

8. ACKNOWLEDGEMENTS

We would like to thank our collaborators at Snowflake for valuable feedback and datasets which made this work possible. This work is sponsored by the National Science Foundation award 1740996 and Microsoft.

9. REFERENCES

[1] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman. Sql querie recommendations. *Proceedings of the VLDB Endowment*, 3(1-2):1597–1600, 2010.

[2] S. Chaudhuri, P. Ganesan, and V. Narasayya. Primitives for workload summarization and implications for sql. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 730–741. VLDB Endowment, 2003.

[3] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 488–499. ACM, 2002.

[4] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 215–226, New York, NY, USA, 2016. ACM.

[5] A. Dan, P. S. Yu, and J. Y. Chung. Characterization of database access pattern for analytic prediction of buffer hit probability. *The VLDB Journal*, 4(1):127–154, Jan. 1995.

[6] Y. Goldberg and O. Levy. word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[7] T. Grust and J. Rittinger. Observing sql queries in their natural habitat. *ACM Trans. Database Syst.*, 38(1):3:1–3:33, Apr. 2013.

[8] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923. ACM, 2015.

[9] S. Jain and B. Howe. Data cleaning in the wild: Reusable curation idioms from a multi-year sql workload. In *Proceedings of the 11th International Workshop on Quality in Databases, QDB 2016, at the VLDB 2016 conference, New Delhi, India, September 5, 2016*, 2016.

[10] S. Jain and B. Howe. SQLShare Data Release. https://uwescience.github.io/sqlshare//data_release.html, 2016. [Online;].

[11] S. Jain and B. Howe. Query2vec: NLP meets databases for generalized workload analytics. *CoRR*, abs/1801.05613, 2018.

[12] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 281–293. ACM, 2016.

[13] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.

[14] Y. Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[15] T. M. Kodinariya and P. R. Makwana. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[16] P. Kołaczkowski. Compressing very large database workloads for continuous online index selection. In *Database and Expert Systems Applications*, pages 791–799. Springer, 2008.

[17] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents.

[18] Y. LeCun. The mnist database of handwritten digits.

[19] O. Levy, Y. Goldberg, and I. Ramat-Gan. Linguistic regularities in sparse and explicit word representations. In *CoNLL*, pages 171–180, 2014.

[20] J. Li, M. Luong, and D. Jurafsky. A hierarchical neural autoencoder for paragraphs and documents. *CoRR*, abs/1506.01057, 2015.

[21] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014. USENIX Association.

[22] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

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

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[26] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. Self-driving database management systems.

[27] K. E. Pavlou and R. T. Snodgrass. Generalizing database forensics. *ACM Trans. Database Syst.*, 38(2):12:1–12:43, July 2013.

[28] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation.

[29] C. Sapia. Promise: Predicting query behavior to enable predictive caching strategies for olap systems. In *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, DaWaK 2000, pages 224–233, London, UK, UK, 2000. Springer-Verlag.

[30] D. Tang, B. Qin, and T. Liu. Document modeling with gated recurrent neural network for sentiment classification.

[31] Q. T. Tran, K. Morfonios, and N. Polyzotis. Oracle workload intelligence. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1669–1681, New York, NY, USA, 2015. ACM.

[32] P. Upadhyaya, M. Balazinska, and D. Suciu. Automatic enforcement of data use policies with datalawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 213–225, New York, NY, USA, 2015. ACM.

[33] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[34] W. Yin, K. Kann, M. Yu, and H. Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[35] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE Trans. Softw. Eng.*, 18(4):347–355, Apr. 1992.

[36] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[37] R. S. Zemel. Autoencoders, minimum description length and helmholtz free energy. NIPS, 1994.