Query-based Workload Forecasting for Self-Driving Database Management Systems

Lin Ma
Carnegie Mellon University
lin.ma@cs.cmu.edu

Dana Van Aken
Carnegie Mellon University
dvanaken@cs.cmu.edu

Ahmed Hefny
Carnegie Mellon University
ahefny@cs.cmu.edu

Gustavo Mezerhane
Carnegie Mellon University
gangulo@andrew.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Geoffrey J. Gordon
Carnegie Mellon University
ggordon@cs.cmu.edu

ABSTRACT

The first step towards an autonomous database management system (DBMS) is the ability to model the target application's workload. This is necessary to allow the system to anticipate future workload needs and select the proper optimizations in a timely manner. Previous forecasting techniques model the resource utilization of the queries. Such metrics, however, change whenever the physical design of the database and the hardware resources change, thereby rendering previous forecasting models useless.

We present a robust forecasting framework called QueryBot 5000 that allows a DBMS to predict the expected arrival rate of queries in the future based on historical data. To better support highly dynamic environments, our approach uses the logical composition of queries in the workload rather than the amount of physical resources used for query execution. It provides multiple horizons (short- vs. long-term) with different aggregation intervals. We also present a clustering-based technique for reducing the total number of forecasting models to maintain. To evaluate our approach, we compare our forecasting models against other state-of-the-art models on three real-world database traces. We implemented our models in an external controller for PostgreSQL and MySQL and demonstrate their effectiveness in selecting indexes.

ACM Reference Format:
Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3196908

1 INTRODUCTION

With the increasing complexity of DBMSs in modern data-driven applications, it is more difficult now than ever for database administrators (DBAs) to tune these systems to achieve the best performance. Many DBAs spend nearly 25% of their time on tuning activities [15]. But personnel is estimated to be 50% of a DBMS' total cost [45]. If the DBMS could optimize itself automatically, then it would remove many of the complications and costs involved with its deployment [35, 40]. Such a "self-driving" DBMS identifies which aspects of itself should optimize without human intervention.

There are two reasons why new efforts to develop a self-driving DBMS are promising even though other attempts have been less than successful. Foremost is that the improved capabilities of modern storage and computational hardware enable the DBMS to collect enough data about its behavior and then use it to train machine learning (ML) models that are more complex than what was possible before. The second reason is that the recent advancements in ML, especially in deep neural networks and reinforcement learning, will allow the DBMS to continually improve its models over time as it learns more about an application's workload.

To be fully autonomous, the DBMS must be able to predict what the workload will look like in the future. If a self-driving DBMS only considers the behavior of the application in the past when selecting which optimizations to apply, these optimizations may be sub-optimal for the workload in the near future. It can also cause resource contention if the DBMS tries to apply optimizations after the workload has shifted (e.g., it is entering a peak load period). Instead, a self-driving DBMS should choose its optimizations proactively according to the expected workload patterns in the future. But the DBMS's ability to achieve this is highly dependent on its knowledge of the queries and patterns in the application's workload.

Previous work has studied database workload modeling in different contexts. For example, one way is to model the demands of resources for the system, rather than a direct representation of the workload itself [14, 44]. Other methods model the performance of the DBMS by answering "what-if" questions about changes in OLTP workloads [37, 38]. They model the workload as a mixture of different types of transactions with a fixed ratio. There is also work to predict how the workload will shift over time using hidden Markov models [28–31] or regressions [19, 36]. Earlier work has also modeled database workloads using more formal methods with pre-defined transaction types and arrival rates [48, 49].

All of these methods have deficiencies that make them inadequate for an autonomous system. For example, some use a lossy compression scheme that only maintains high-level statistics, such as average query latency and resource utilization [14, 37, 38, 44]. Others assume that the tool is provided with a static workload [48, 49], or they only generate new models when the workload shifts, thereby failing to capture how the volume of queries and the workload

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196908

	Admissions	BusTracker	MOOC
DBMS Type	MySQL	PostgreSQL	MySQL
Num of Tables	216	95	454
Trace Length (Days)	507	58	85
Avg. Queries Per Day	5M	19.9M	1.1M
Num of SELECT Queries	2541M [99.8%]	19.5M [98%]	0.97M [88%]
Num of INSERT Queries	1.8M [0.07%]	15K [0.8%]	14K [1.3%]
Num of UPDATE Queries	2.6M [0.1%]	22K [1%]	66K [6%]
Num of DELETE Queries	0.4M [0.02%]	3K [0.2%]	51K [4.7%]

Table 1: Sample Workloads – Summarization of the workload traces collected from the database applications described in Section 2.1.

trends change over time [19, 36, 48, 49]. Lastly, some models are hardware and/or database design dependent [28–31], which means the DBMS has to retrain them whenever its configuration changes.

In this paper, we present a method to succinctly forecast the workload for self-driving DBMSs. Our approach continuously clusters queries based on the their arrival rate temporal patterns. It seamlessly handles different workload patterns and shifts. It then builds models to predict the future arrival patterns for the query clusters. Such predictions are necessary to enable an autonomous DBMS's planning module to identify optimizations to improve the system's performance and apply them proactively [40]. The key advantage of our approach over previous forecasting methods is that the data we use to train our models is independent of the hardware and the database design. Thus, it is not necessary to rebuild the models if the DBMS's hardware or configuration settings change.

To evaluate our forecasting models, we integrated this framework into both MySQL [1] and PostgreSQL [5], and measure its ability to model and optimize three real-world database applications. The results demonstrate that our framework can efficiently forecast the expected future workload with only a minimal loss in accuracy. They also show how a self-driving DBMS can use this framework to improve the system's performance.

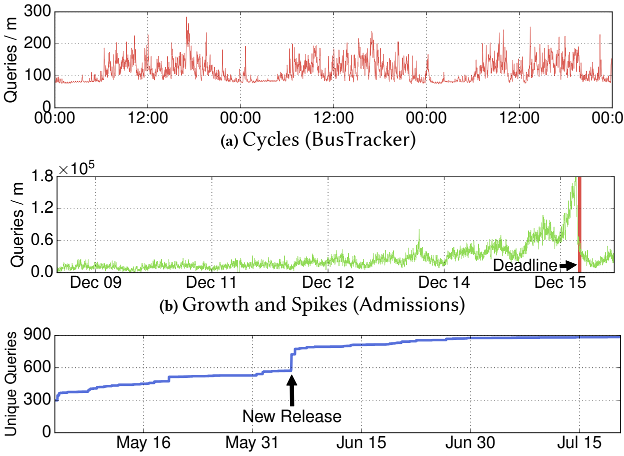
The remainder of this paper is organized as follows. Section 2 discusses common workload patterns in database applications. We next give an overview of our approach in Section 3 and then present the details of its components: Pre-Processor (Section 4), Clusterer (Section 5), and Forecaster (Section 6). We provide an analysis of our methods in Section 7 and conclude with related work in Section 8.

2 BACKGROUND

The goal of workload forecasting in an autonomous DBMS is to enable the system to predict what an application's workload will look like in the future. This is necessary because the workloads for real-world applications are never static. The system can then select the optimizations to prepare based on this prediction.

We contend that there are two facets of modern database applications that make robust, unsupervised workload forecasting challenging. The first is that an application's queries may have vastly different arrival rates. Thus, an effective forecasting model must be able to identify and characterize each of these arrival rate patterns. The second is that the composition and volume of the queries in an application's workload change over time. If the workload deviates too much from the past, then the forecasting models become inaccurate and must be recomputed.

We now investigate the common characteristics and patterns in today's database applications in more detail. We begin with an introduction of the three real-world database application traces used in our discussions and experiments, followed by an overview of the common patterns that these traces exhibit. We then discuss the challenges that effective forecasting models must overcome.



(a) Cycles (BusTracker)

(b) Growth and Spikes (Admissions)

(c) Workload Evolution (MOOC)

Figure 1: Workload Patterns – Examples of three common workload patterns in database applications.

2.1 Sample Workloads

We now give a high-level description of the three sample workload traces collected from real-world database applications. Table 1 provides a more detailed summary of their properties.

Admissions: An admissions website for a university's graduate program. Students submit their application materials to programs in different departments. Faculties review the applications after the deadline and give their decisions.

BusTracker: A mobile phone application for live-tracking of the public transit bus system. It ingests bus location information at regular intervals from the transit system, and then helps users find nearby bus stops and get route information.

MOOC: A web application that offers on-line courses to people who want to learn or teach [3]. Instructors can upload their course materials, and students can check out the course content and submit their course assignments.

2.2 Workload Patterns

We now describe the three common workload patterns prevalent in today's database applications. The first two patterns are examples of different arrival rates that the queries in database applications can have. The third pattern exhibits how the composition of the queries in the workload can change over time.

Cycles: Many applications are designed to interact with humans, and as such, their workloads follow cyclic patterns. For example, some applications execute more queries at specific ranges of a day than at others because this is when people are awake and using the service. Figure 1a shows the number of queries executed per minute by the BusTracker application over a 72-hour period. The DBMS executes more queries in the daytime, especially during the morning and afternoon rush hours since this is when people are taking buses to and from work. This cycle repeats every 24 hours.

Not all applications have such peaks at the same time each day, and the cycles may be shorter or longer than this example.

Growth and Spikes: Another common workload pattern is when the query volume increases over time. This pattern is typical in start-ups with applications that become more popular and in applications with events that have specific due dates. The Admissions application has this pattern. Figure 1b shows the number of queries per minute executed over a week-long period leading up to the application deadline. The arrival rate of queries increases as the date gets closer: It grows slowly at the start of the week but then increases rapidly for the final two days before the deadline.

Workload Evolution: Database workloads evolve over time. Sometimes this is a result of changes in the arrival rate patterns of existing queries (e.g., new users located in different time zones start using an application). The other reason this happens is that the queries can also change (e.g., new application features). Of our three applications, MOOC incurs the most changes in its workload mixture. Figure 1c shows the accumulated number of distinct queries that the MOOC application executes over time. The graph shows that there is a large shift in the workload after the organization released a new feature in their application in early May.

2.3 Discussion

There are three challenges that one must solve for a forecasting framework to work in real-world DBMS deployments. Foremost is that in order to exploit the various arrival rate patterns in the workloads for optimization planning, there is a need for good arrival rate forecasting models. Not only do different workloads have different patterns, but a single workload can also have separate patterns for sub-groups of queries. Thus, an effective forecasting model must be able to identify and characterize patterns that are occurring simultaneously within the same workload.

The second challenge is that since applications execute millions of queries per day, it is not feasible to build forecasting models for each query in the workload. This means that the framework must reduce the complexity of the workload that it analyzes without severely reducing the accuracy of its predictions.

Lastly, the framework must handle the changes in the workload's patterns as well as the query mixtures. All of this must be done without any human intervention. That is, the framework cannot require for a DBA to tune its internal parameters or provide hints about what the application's workload is and when it changes.

3 QUERYBOT 5000 OVERVIEW

The QueryBot 5000 (**QB5000**) is a workload forecasting framework that runs as either an external controller or as an embedded module. The target DBMS connects to the framework and forwards all the queries that applications execute on it. Decoupling the framework from the DBMS allows the DBA to deploy it on separate hardware resources. Forwarding the queries to QB5000 is a lightweight operation and is not on the query executor's critical path. As QB5000 receives these queries, it stores them in its internal database. It then trains models that predict which types of queries and how many of them the DBMS is expected to execute in the future. A self-driving DBMS can then use this information to deploy optimizations that will improve its target objective (e.g., latency, throughput) [40].

As shown in Figure 2, QB5000's workflow is comprised of two phases. When the DBMS sends a query to QB5000, it first enters the **Pre-Processor** and the **Clusterer** components. This is the part of the system that maps the unique query invocation to previously seen queries. This enables QB5000 to reduce both the computational and storage overhead of tracking SQL queries without sacrificing accuracy. The Pre-Processor converts raw queries into generic *templates* by extracting constant parameters out of the SQL string. It then records the arrival rate history for each template.

It is still, however, not computationally feasible to build models to capture and predict the arrival patterns for each template. To further reduce the computational resource pressure, QB5000 then maps the template to the most similar group of previous queries based on its semantics (e.g., the tables that it accesses). The Clusterer then performs further compression of the workload using an online clustering technique to group templates with similar arrival rate patterns together. It is able to handle evolving workloads where new queries appear and older ones disappear.

In the final phase, the **Forecaster** selects the largest template clusters (i.e., clusters with the highest query volumes) and then trains forecasting models based on the average arrival rate of the templates within each cluster. These models predict how many queries in each template cluster that the application will execute in the future (e.g., one hour from now, one day from now). This is important because the DBMS will decide how to optimize itself based on what it expects the application to do in the future, rather than what happened in the past. QB5000 also automatically adjust these clusters as the workload changes over time. Every time the cluster assignment changes for templates, QB5000 re-trains its models.

The Pre-Processor always ingests new queries and updates the arrival rate history for each template in the background in real time when the DBMS is running. The Clusterer and Forecaster periodically update the cluster assignments and the forecasting models. When QB5000 predicts the expected workload in the future, it uses the most recent data as the input to the models.

4 PRE-PROCESSOR

Most applications interact with a DBMS in a programmatic way. That is, the queries are constructed by software in response to some external mechanism rather than a human writing the query by hand. For OLTP workloads, the application invokes the same queries with different input parameters (e.g., prepared statements). For OLAP workloads, a user is often interacting with a dashboard or reporting tool that provides an interface to construct the query with different predicates and input parameters. Such similar queries execute with the same frequency and often have the same resource utilization in the system. Thus, QB5000 can aggregate the volume of queries with identical templates together to approximate the characteristics of the workload. This reduces the number of queries that QB5000 tracks since it only needs to maintain arrival rate information for each template rather than each individual query. Given this, we now describe how QB5000's Pre-Processor collects and combines the queries that it receives from the DBMS.

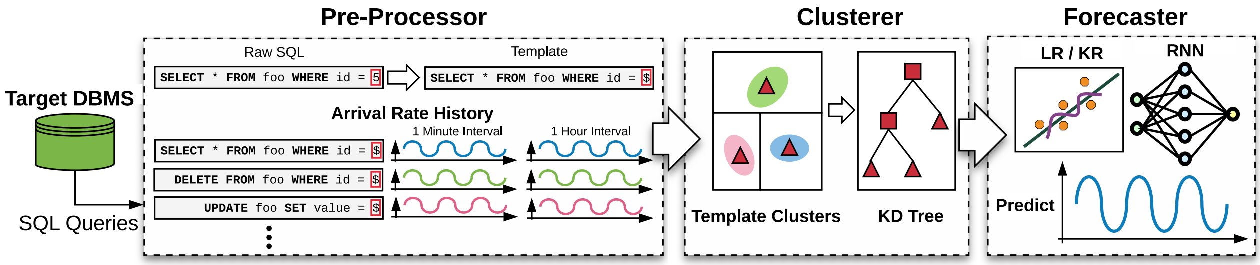


Figure 2: QB5000 Workflow – The framework receives SQL queries from the DBMS. This data is first passed into the Pre-Processor that identifies distinct templates in the workload and records their arrival rate history. Next, the Clusterer combines the templates with similar arrival rate patterns together. This information is then fed into the Forecaster where it builds models that predict the arrival rate of templates in each cluster.

The Pre-Processor processes each query in two steps. It first extracts all of the constants from the query's SQL string and replaces them with value placeholders. This converts all of the queries into prepared statements. These constants include:

- The values in WHERE clause predicates.
- The SET fields in UPDATE statements.
- The VALUES fields in INSERT statements. For batched INSERTs, QB5000 also tracks the number of tuples.

The Pre-Processor then performs additional formatting to normalize spacing, case, and bracket/parenthesis placement. We use the abstract syntax tree from the DBMS's SQL parser to identify the tokens. The outcome of this step is a generic query *template*.

QB5000 tracks the number of queries that arrive per templates over a given time interval and then stores the final count into an internal catalog table at the end of each interval. The system aggregates stale arrival rate records into larger intervals to save storage space. We explain how to choose the time interval in Section 6.

QB5000 also maintains a sample set of the queries' original parameters for each template in its internal database. We use reservoir sampling to select a fixed amount of items with low variance from a list containing a large or unknown number of items [53]. An autonomous DBMS's planning module uses these parameter samples when estimating the cost/benefit of optimizations [40].

The Pre-Processor then performs a final step to aggregate templates with equivalent semantic features to further reduce the number of unique templates that QB5000 tracks. Evaluating semantic equivalence is non-trivial, and there has been extensive research on this topic [33, 47, 52]. QB5000 uses heuristics to approximate the equivalence of templates. It considers two templates as equivalent if they access the same tables, use the same predicates, and return the same projections. One could use more formal methods to fully exploit semantic equivalence [13]. We found, however, that heuristics provide reasonable performance without reducing accuracy. We defer investigating more sophisticated methods as future work.

Table 2 shows that QB5000's Pre-Processor is able to reduce the number of queries from millions to at most thousands of templates for our sample workloads.

5 CLUSTERER

Even though the Pre-Processor reduces the number of queries that QB5000 tracks, it is still not feasible to build models for the arrival patterns of each template. Our results in Section 7.5 show that it can take over three minutes to train a single model. Thus, we need to further reduce the total number of templates that QB5000 forecasts.

	Admissions	BusTracker	MOOC
Total Number of Queries	2546M	1223M	95M
Total Num of Templates	4060	334	885
Num of Clusters	1950	107	391
Reduction Ratio	**1.3M**	**10.5M**	**0.24M**

Table 2: Workload Reduction – Breakdown of the total number of queries that QB5000 must monitor after applying the reduction techniques in the Pre-Processor and Clusterer.

The Clusterer component combines the arrival rate histories of templates with similar patterns into groups. It takes templates in a high-dimensional feature space and identifies groups of comparable templates using a similarity metric function. To support modeling an application in a dynamic environment (i.e., one where the workload, database physical design, and the DBMS's configuration can change), the clustering algorithm must generate stable mappings using features that are not dependent on the current state of the database. This is because if the mapping of templates to clusters changes, then QB5000 has to retrain all of its models.

We now examine the three design decisions in our implementation of QB5000's clustering phase. We begin with a discussion of the features that it extracts from each template. We then describe how it determines whether templates belong to the same cluster. Lastly, we present QB5000's clustering algorithm that supports incremental updates as the application's workload evolves, and how the framework quickly determines whether to rebuild its clusters.

5.1 Clustering Features

There are three types of features that the framework can derive from templates: (1) *physical*, (2) *logical*, and (3) *arrival rate history*. Physical features are the amount of resources and other runtime metrics that the DBMS used when it executed the query, such as the number of tuples read/written or query latency. Previous clustering algorithms for database applications have used physical features for query plan selection [24], performance modeling [37], and workload compression [11]. The advantage is that they provide fine-grained and accurate information about an individual query. But they are dependent on the DBMS's configuration and hardware, the database's contents, and what other queries were running at the same time. If any of these change, then the previously collected features are useless and the framework has to rebuild its models. Such instability makes it difficult for the DBMS's planning module to learn whether its decisions are helping or hurting the performance.

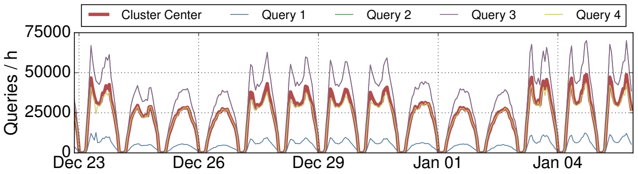


Figure 3: Arrival Rate History – The past arrival rates for the largest cluster and the top four queries within that cluster from BusTracker.

Another approach is to use the template's logical features, such as the tables/columns it accesses and the properties of the query's syntax tree. Unlike physical features, these logical features do not depend on the DBMS's configuration nor the characteristics of the workload (e.g., which queries execute more often than others). The disadvantage, however, is that they may generate clusters without a discernible workload pattern because there is limited information from the logical feature and thus the forecasting models make poor predictions. The inefficiency of logical features has also been identified in previous work on predicting query runtime metrics [23].

QB5000 uses a better approach to cluster queries based on their arrival rate history (i.e., the sequence of their past arrival rates). For example, consider the cluster shown in Figure 3 from the BusTracker application that is derived from four templates with similar arrival rate patterns. The cluster *center* represents the average arrival rate of the templates within the cluster. Although the total volume per template varies at any given time, they all follow the same cyclic patterns. This is because these queries are invoked together as part of performing a higher level functionality in the application (e.g., a transaction). Since templates within the same cluster exhibit similar arrival rate patterns, the system can build a single forecasting model for each cluster that captures the behavior of their queries.

Calculating the similarity between a pair of arrival rate history features is straightforward. QB5000 first randomly samples timestamps before the current time point. Then for each series of arrival rate history, QB5000 takes the subset of values at those timestamps to form a vector. The similarity between the two features is defined as the cosine similarity of the two vectors. If the template is new, we compare its available timestamps with the corresponding subset in the vectors of other templates. Our current implementation uses 10k time points in the last month of a template's arrival rate history as its feature vector. We found that this is enough to capture the pattern of every arrival rate history in our experiments.

Logical features and arrival rate history features express different characteristics of the queries. But as we show in Section 7.7, clustering on the arrival rate features produce better models for real-world applications because they capture how queries impact the system's performance. Though using the template's arrival rates avoids rebuilding clusters whenever the DBMS changes, it is still susceptible to workload variations, such as when the system identifies a new template or the arrival rates of existing ones change.

5.2 On-line Clustering

QB5000 uses a modified version of **DBSCAN** [21] algorithm. It is a density-based clustering scheme: given a set of points in some space, it groups together points with many nearby neighbors (called *core objects*), and marks points that lie alone in low-density regions as outliers (i.e., points whose nearest neighbors are too far away). Unlike K-means, this algorithm is not affected by the number of small clusters or the cluster densities[1].

The original DBSCAN algorithm evaluates whether an object belongs to a cluster by checking the minimum distance between the object and any core object of the cluster. But we want to assign templates to clusters based on how close they are to a cluster's center and not just any random core object. This is because QB5000 uses the center of a cluster to represent the templates that are members of that cluster, and builds forecasting models with the center. An on-line extension of the canonical DBSCAN algorithm also has high overhead when updating clusters [20].

Our on-line variant of DBSCAN uses a threshold, ρ ($0 \leq \rho \leq 1$), to decide how similar the arrival rates of the templates must be for them to belong to the same cluster. The higher ρ is, the more similar the arrival rates of the templates within a cluster are, so the modeling result will be more accurate. But the computational overhead will also be higher given the larger number of generated clusters. We conduct a sensitivity analysis on setting this value in Appendix A. As shown in Figure 4, QB5000's incremental clustering algorithm periodically performs the following three steps together:

Step #1: For each new template, QB5000 first checks whether the similarity score between its arrival rate history and the center of any cluster is greater than ρ. The template is assigned to the cluster with the highest similarity score that is greater than ρ. We use a *kd-tree* to allow QB5000 to quickly find the closest center of existing clusters to the template in a high-dimensional space [8]. Then QB5000 will update the center of that cluster, which is the arithmetic average of the arrival rate history of all templates in that cluster. If there is no existing cluster (this is the first query) or none of the clusters' centers are close enough to the template, QB5000 will create a new cluster with that template as its only member.

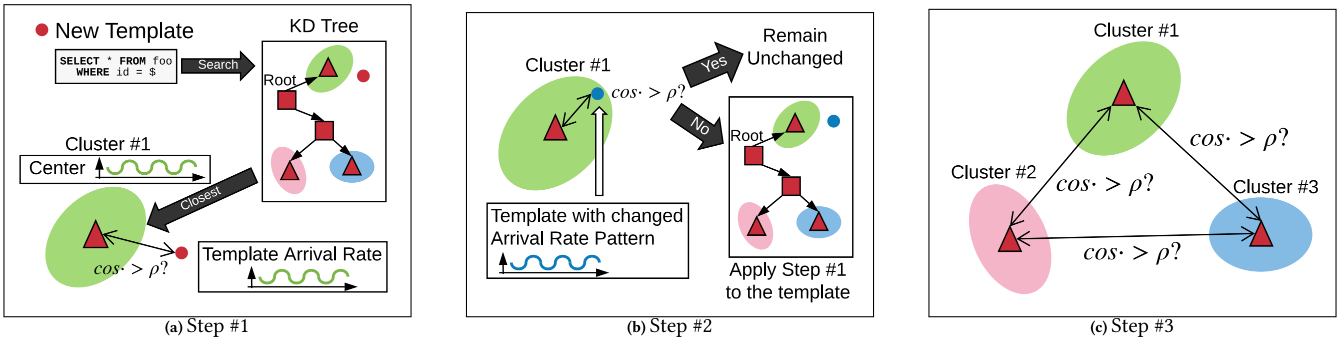
Step #2: QB5000 checks the similarity of previous templates with the centers of the clusters they belong to. If a template's similarity is no longer greater than ρ, QB5000 removes it from its current cluster and then repeat step (1) to find a new cluster placement. Sometimes moving a template from one cluster to another causes the centers of the two clusters to change, and recursively forces other templates from the two clusters to move. QB5000 defers modifying the clusters until the next update period. QB5000 removes a template if it has not received one of its queries for an extended period.

Step #3: QB5000 computes the similarity between the clusters' centers and merges two clusters with a score greater than ρ.

In addition to periodically executing these three steps, QB5000 monitors the new templates in the workload. If the percentage of previously unseen templates is above a threshold, it then triggers these steps to adapt to the workload change. Setting this threshold properly is dependent on the performance attributes of the target DBMS. We defer investigating this problem as future work.

QB5000's incremental algorithm adaptively adjusts the clusters for a dynamic workload without requiring a warm-up period or having prior knowledge of the workload. More importantly, it guarantees that the similarity between a template's arrival rate history

[1] We also evaluated K-means clustering, but it has a known problem when the workload has a large number of small clusters, or the clusters have different sizes or densities. These issues have also been observed for previous database workload modeling techniques [11].



(a) Step #1 (b) Step #2 (c) Step #3

Figure 4: On-line Clustering – Clusterer periodically performs three steps to change the templates' cluster assignments incrementally. It first checks the similarities between the arrival rate history features of new templates with the centers of existing clusters. Then it removes the templates from their clusters if the arrival rate patterns of a template and its cluster center have deviated. Finally, it merges the clusters with high similarities between their centers together.

	LR	ARMA	KR	RNN	FNN	PSRNN
Linear	✓	✓	✗	✗	✗	✗
Memory	✗	✓	✗	✓	✗	✓
Kernel	✗	✗	✓	✗	✗	✓

Table 3: Forecasting Models – The properties of the forecasting models that we investigated for QB5000.

and the center of its cluster is smaller than ρ, which improves the accuracy of QB5000's forecasting models. The complexity of these steps is bounded by $O(n \log n)$, where n is the number of templates in the workload. Since Step #2 does not apply cluster changes recursively, this approach does not guarantee the convergence of the clusters at any specific time. Thus, QB5000 might not achieve the optimal clustering given the similarity metric. We found, however, this does not affect the efficacy of its forecasting models in practice.

Table 2 shows the number of clusters determined by Clusterer and the reduction ratio from the total number of queries in the three workloads. Since QB5000 periodically updates its clustering results, we show the average number of clusters per day.

5.3 Cluster Pruning

Even after using clustering techniques to reduce the total number of queries that QB5000 needs to model, real-world applications still tend to have lots of clusters because of the long-tailed distribution of the arrival rate patterns. There are only a few large clusters that exhibit the major workload patterns, but several small clusters with noisy patterns. Those small clusters usually contain queries that only appear a few times and increase the noise in the models with little to no benefit since they are not representative of the application's main workload. QB5000 does not build models for them since their impact on the DBMS's performance is limited. Our experiments in Section 7.1 show that the five largest clusters cover up to 95% of the query volume for our three sample workloads.

6 FORECASTER

At this point in QB5000's pipeline, the framework has converted raw SQL queries into templates and grouped them into clusters. QB5000 is also recording the number of queries executed per minute for each cluster. The final phase is to build *forecasting models* to predict the arrival rate patterns of the clusters' queries. These models allow the DBMS's planning module to estimate the number of queries that the application will execute in the future and select the proper optimizations to meet SLAs [4, 40]. In this section, we describe how QB5000 constructs and uses its forecasting models. We begin with an explanation of its underlying data structures and training methods. We then discuss how QB5000 supports different prediction horizons and intervals for the same cluster over multiple models.

6.1 Forecasting Models

There are many choices for forecasting models of the query arrival rates with different prediction properties. Table 3 shows a summary of the six models that we considered in the development of QB5000. The first property is whether the model is *linear*, which means that it assumes that there is a linear relationship between the input and output data. Next, a model can retain *memory* that allows it to use both the input data and the information "remembered" from the past observations to predict the future. Lastly, a model can support *kernel* methods to provide another way to model non-linear relationships. In contrast to models that employ non-linear functions, kernel methods achieve non-linearity by using linear functions on the feature maps in the kernel space of the input.

Linear models are good at avoiding overfitting when the intrinsic relationship in the data is simple. They take less computation to build and require a smaller amount of training data. On the other hand, more powerful non-linear models are better at learning complex data patterns. They do, however, take longer to train and are prone to overfitting. Thus, they require more training data. As we will show in Section 7.2, linear models often perform better at making predictions in the near future (e.g., one hour) whereas the non-linear models are better at making predictions further out in time (e.g., over a day). But it is non-trivial to determine which type of model to use for different time horizons on different workloads. Likewise, there are also trade-offs between memory-based models. Retaining memory allows the model to exploit the dynamic temporal behavior in an arbitrary sequence of inputs, but this adds training complexity and makes the model less data-efficient.

A well-known solution that works well in other application domains is to use an *ensemble* method (**ENSEMBLE**) that combines multiple models together to make an average prediction. Ensemble methods are used in prediction tasks to combine several machine learning techniques into one predictive model in order to decrease variance or bias (e.g., *boosting*) [39]. Previous work has shown that ensemble methods work well on challenging datasets and they are often among the top winners of data science competitions [43, 57].

We now discuss the two types of forecasting models that QB5000 combines to make its ENSEMBLE model.

Linear Regression (LR): LR models are also known in the statistics and time series prediction literature as linear auto-regressive models. They are simple linear models that have closed-form solutions, which means that they do not require an additional optimization step to find a global optima. In QB5000, the framework regresses the future arrival rate of queries in a cluster based on the arrival rate of the query over a specified period of time in the past. LR has been used for DBMS operator modeling in prior work [6].

Recurrent Neural Network (RNN): Previous work has shown RNNs to be effective at predicting patterns for non-linear systems [54]. It is a class of network where its neurons have a cyclic connection. QB5000 uses a variant of the RNN called *long short-term memory* (LSTM) [27]. LSTMs contain special blocks that determine whether to retain older information and when to output it into the network. This allows the networks to automatically learn the periodicity and repeating trends of data points in a time-series beyond what is possible with regular RNNs [22]. This approach has been used to predict the host-load in data centers [50].

We apply an ensemble method by equally averaging the prediction results of the LR and RNN models. We also tried averaging the models with weights derived from the training history, but that led to overfitting and generated worse results.

Although the ensemble method achieves good average prediction accuracy, we found that it fails to predict the periodic spikes in the workload that are far apart from each occurrence. For example, the December 15th deadline shown in Figure 1b occurs each year on the same date, but both the LR and RNN models are unable to predict this annual pattern. This is a common workload pattern, and the ability to predict such spikes ahead of time is necessary for many database optimizations, such as resource provisioning. Thus, we now describe the third forecasting model that we employ in QB5000 that is able to correctly handle this scenario.

Kernel Regression (KR): This is a non-linear variant of LR models that uses the Nadaraya-Watson estimator to achieve its non-linearity without iterative training [9]. The prediction for a given input is a weighted average of training outputs where the weights decrease with distance between the given input and corresponding training inputs.

KR is a non-parametric method, which means that it assumes no particular functional form. It instead only assumes that the function is smooth. This provides it with the necessary flexibility to model different non-linear functions between the inputs and the outputs. Thus, it is able to predict when a spike will repeat in the future even if the spike has only occurred a few times in the past. KR, however, does not extrapolate well with data it has not seen before. As we show in Section 7.2, it performs worse than ENSEMBLE in terms of the average prediction accuracy. But it is the only investigated model that is able to handle the yearly spike in Admissions.

Given this, QB5000 uses a hybrid forecast model (**HYBRID**) that we developed to automatically determine when to use predictions from ENSEMBLE versus ones generated from KR. Since KR is good at predicting spikes with a small number observations, if its predicted workload volume is above that of ENSEMBLE by more than a specified threshold, γ ($\gamma \geq 0$), then QB5000 uses the result from KR as its prediction. Otherwise, it uses the result generated from the ENSEMBLE model. In QB5000, we set γ to 150% as this provided the most accurate forecasts for all of the application workloads that we tested. We provide a sensitivity analysis of γ in Appendix C.

6.2 Prediction Horizons & Intervals

The scope of a forecasting model is defined in terms of its horizon and interval. How far into the future a model can predict is known as its *prediction horizon*. In general, the longer the horizon is, the less accurate its predictions are expected to be. This is important for self-driving DBMSs since it improves their ability to prepare for immediate workload and resource demand changes. The time granularity at which the model can predict is called its *prediction interval*. For example, a model can predict the number of queries that will execute in one-minute or one-hour intervals. Like the horizon, using a shorter interval often improves the accuracy of its models, but increases the storage and computational overhead.

QB5000 sets the interval at which it records the query arrival rates to be one minute, which is the finest level of prediction that QB5000 is able to provide to the DBMS. When the planning module of a self-driving DBMS evaluates potential optimizations, it can decide how to aggregate the per-minute history into longer intervals for training the forecasting models. We evaluate QB5000's performance when aggregating over different time intervals in Section 7.4. To predict the spikes, QB5000 trains the KR model used by HYBRID using the entire history of an application aggregated into one-hour intervals to reduce the computational and storage overhead.

The planning module of a self-driving DBMS also decides how far ahead of time its models need to make predictions. QB5000 builds a forecasting model for each required prediction horizon.

7 EXPERIMENTAL ANALYSIS

We now present an evaluation of QB5000's ability to model a database application's workload and predict what it will look like in the future. We implemented QB5000's algorithms using `scikit-learn`, `Tensorflow`, and `PyTorch`. We use the three real-world workload traces described in Section 2.1. We performed our experiments on a single server with an Intel Xeon E5-2420v2 CPU and 32 GB RAM. Unless otherwise noted, we use a GeForce GTX 1080 GPU with 8 GB RAM for training QB5000's forecasting models.

We first analyze the effectiveness of the Clusterer's compression techniques. We then evaluate the accuracy of the Forecaster's models. We also examine the computation time and storage footprint for QB5000's components. Finally, we demonstrate the benefits of QB5000 for self-driving DBMSs for an example application.

7.1 Number of Clusters

The goal of this first experiment is to show that QB5000 can model the majority of a database workload using a small number of the highest-volume clusters. Although modeling more clusters may allow QB5000 to capture more information about the workload, it also increases the computational overhead, data requirements, and memory footprint. We use the method described in Section 5 to perform on-line query clustering for all the three workloads. We set QB5000's threshold as ρ=0.8 and the frequency at which it performs

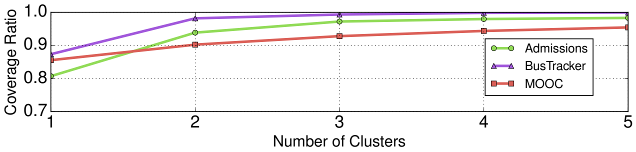


Figure 5: Cluster Coverage – The average ratio between the volume of the largest clusters and the total workload volume.

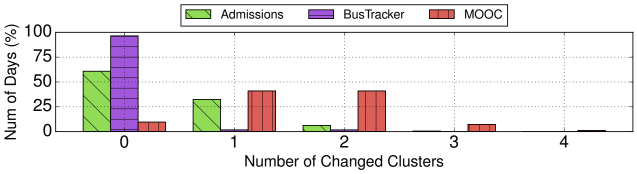


Figure 6: Cluster Change – The number of clusters that changed among the five largest clusters between two consecutive days.

incremental clustering algorithm to be once per day. We provide a sensitivity analysis for selecting ρ in Appendix A.

We first calculate the average ratio between the volume of the largest clusters and the total workload volume for each day throughout the entire workload execution. It is calculated by dividing the volume of a given cluster by the total volume of all the clusters for that day. The results in Figure 5 show that the highest-volume clusters cover the majority of the queries in the workload. For example, in BusTracker most of the queries come from people checking bus schedules during the morning and evening rush hours. In particular, the five largest clusters comprise over 95% of the queries for all three workloads. This shows that even though a real-world application may consist of several arrival rate patterns among sub-groups of queries, we can still obtain a good estimation of the workload by modeling only a few of its major patterns. Recall that QB5000 tracks the ratio between the volume of the templates within a cluster.

We then calculate how frequently the five highest-volume clusters change from one day to the next. Figure 6 shows the number of days where zero or more changes occurred in the five largest clusters of the three workloads. For Admissions and BusTracker, there is at most one change in the five largest clusters for over 90% of the days. This shows that QB5000's on-line clustering method is not only efficient regarding the characterization of the workload, but is also stable under the usual fluctuations in the workload. The MOOC workload has more cluster changes than the other two because new queries appear as instructors create and launch new classes. This shows that QB5000's incremental clustering algorithm can capture shifts in the application's workload as it changes over time.

7.2 Prediction Accuracy Evaluation

We now evaluate the prediction accuracy of QB5000's forecasting models for different prediction horizons. Recall from Section 6.2 that the prediction horizon defines how far into the future a model predicts. In this experiment, QB5000 models the highest-volume clusters that cover more than 95% of the total queries in the workload. Our previous results in Figure 5 show that we can achieve this by modeling the three largest clusters for Admissions and BusTracker, and the five largest clusters for MOOC. QB5000 trains a single forecasting model that jointly predicts the query arrival rates for all of the clusters on each prediction horizon. This allows for sharing information across clusters, which improves the prediction accuracy. It uses up to three weeks of the latest query arrival rate data for the training. We use the log of the mean squared error (MSE) as the metric for measuring the accuracy of QB5000's forecasting models. The smaller the MSE, the better the prediction accuracy. We use one-hour prediction horizon in this experiment.

For a self-driving DBMS, a desirable property of its forecasting models is that they are not overly sensitive to their hyperparameters. This is because fine-tuning a model's hyperparameters is by itself a hard optimization task. To evaluate this, we fix the hyperparameters for all models to be the same across the different prediction horizons and workloads. We obtained these settings using cross-validation. Except for the KR used by HYBRID, we use the last day's arrival rate as the input for the LR and KR models. For RNN, we use a linear embedding layer of size 25 followed by two LSTM layers each with 20 cells. We take the log of the input before training the models, and convert them back by taking the exponentials of the output.

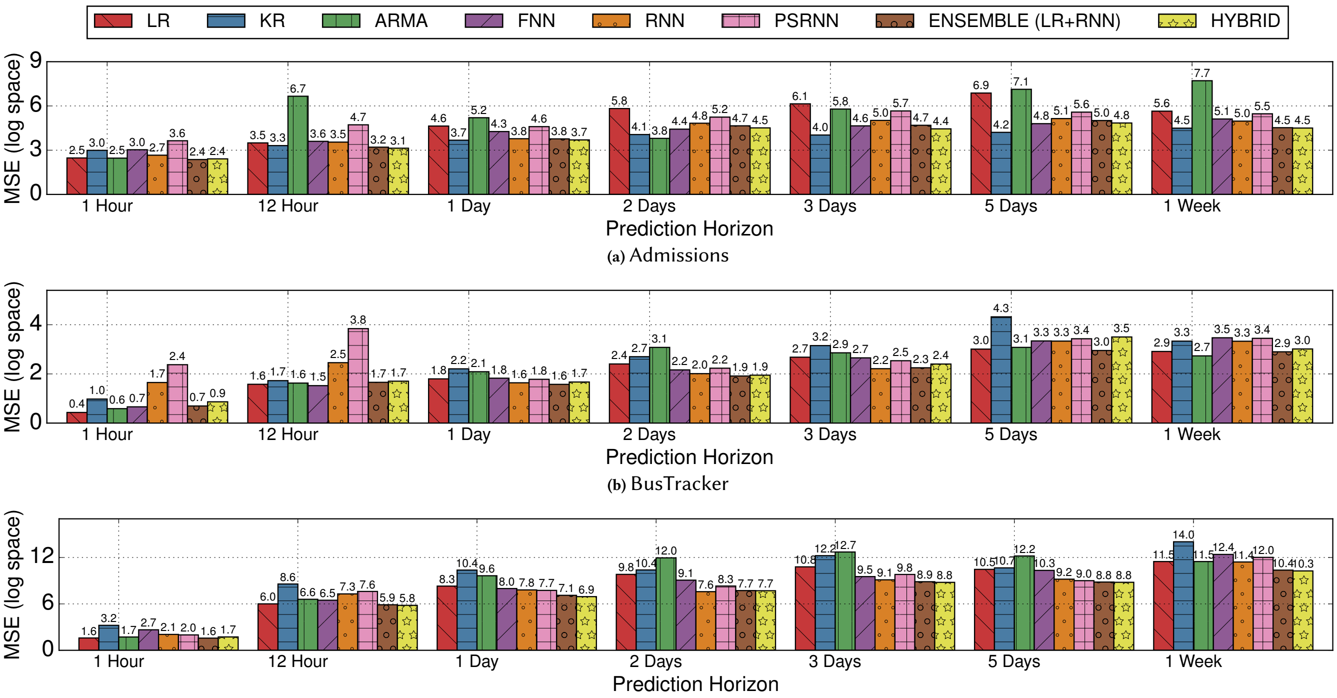
We compare the accuracy of QB5000's forecasting models discussed in Section 6.1 against other models used for forecasting arrival rate patterns. They are used in existing approaches for modeling system workloads and resource usages, as introduced below.

Autoregressive Moving Average (ARMA): ARMA is a generalization of LR models that consists of an autoregressive part and a moving average part acting on residuals. When making predictions, ARMA makes use of all previous observations, either directly or through its residuals. This approach was used for predicting workloads in cloud service environments [7, 46].

Feed-forward Neural Network (FNN): This is a non-linear version of the LR models in which the linear function that approximates the output is replaced by a feed-forward neural network that separates a sequence of linear transformations to the input vector by non-linear activations [51]. FNNs differ from RNNs in that their neurons do not form a cycle that feeds information from all previous observations back into the model. This approach was used to predict the resource usage for transactions in an OLTP DBMS [32].

Predictive State Recurrent Neural Network (PSRNN): This a newer RNN variant that outperforms LSTMs in a variety of prediction tasks [17]. The key advantage of RNNs is that they have an initialization algorithm based on a method of moments that aims to start the optimization process in a better position towards the global optima, as opposed to typical RNN/LSTM initializations.

Figure 7 shows the average prediction accuracy of the forecasting models over horizons ranging from one hour to one week for the three workloads. These results exhibit similar trends for how the horizon impacts the prediction accuracy of the models. First, we observe that for shorter horizons, the LR models perform as well as or better than the more complex RNN models. This is because when the horizon is short, the relationship between the arrival rate observed in the recent past and the arrival rate in the near future is more linear than for longer horizons. Thus, a simple model like LR is sufficient for making predictions. In contrast, complex models often overfit the noise in the training data and generate less accurate results when the horizon is short. But as the horizon increases, the relationship between the past and the future also grows in complexity. In this case, more powerful models like RNNs



(a) Admissions

(b) BusTracker

(c) MOOC

Figure 7: Forecasting Model Evaluation – The average prediction accuracy of the different forecasting models over prediction horizons ranging from one hour to one week for the Admissions, BusTracker, and MOOC workloads.

that are adept at learning complex relationships achieve better accuracy. This is consistent with our results; RNN outperforms LR when the horizon is greater than or equal to one day. This effect is also observed in sequence prediction in other domains [10].

These results also show that the accuracy of ARMA is not stable across the different horizons. For all of the trials in Figure 7, it achieves the best performance for only 10% of them, but it has the worst performance 38% of the time. This is because the model is sensitive to its hyperparameters. The optimal hyperparameter settings for ARMA are highly dependent on the statistical properties of the data, such as stationarity and the autocorrelation structure.

The FNN models generally have worse prediction accuracy compared to the RNN models. FNN achieves the best and the worst accuracy in both 5% of the trials. The FNN models cannot remember the state of the workload like RNNs. They also lack the simplicity of LR that protects against overfitting.

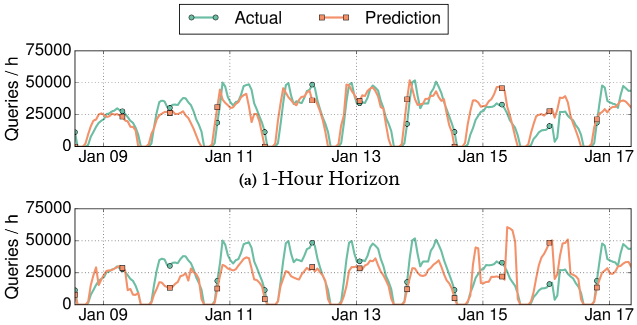
KR has the best performance in 19% of the experiments, but the worst in 24% of the experiments. This model is able to model non-linear functions, but it is prone to error when it has not seen inputs in training that are close to the input to make the prediction with.

PSRNN also performs worse than RNN. It is supposed to have a better initialization than RNN, but this does not always guarantee a better performance because (1) it uses approximation algorithms to find the initialization and (2) its benefit from smarter initialization is restricted when the size of the training data is limited [17]. Since RNN takes less training time than PSRNN and is more available in ML frameworks, we did not use PSRNN in ENSEMBLE.

As shown in Figure 7, ENSEMBLE provides the best overall prediction accuracy. It performs better than all the stand-alone models in 61% of the experiments and never has the worst performance. Ensemble methods often have lower variance than their underlying models and produce better results when their models have complementing characteristics [34]. LR and RNN have distinct properties: LR only uses a limited number of observations from the past when making linear predictions, whereas RNN is non-linear and maintains state to memorize the information from all previous observations. Since LR has comparable performance to ENSEMBLE for horizons shorter than a day, using LR on these short horizons may also be desirable for a DBMS that is short of computational resource.

Even though ENSEMBLE achieves the best accuracy of all the models, recall from Section 6.1 that it cannot predict the spikes that repeat infrequently in the workload. HYBRID solves this issue by correcting the result of ENSEMBLE with the help of the prediction from KR. Figure 7 shows that HYBRID has little impact on the average prediction accuracy compared to ENSEMBLE.

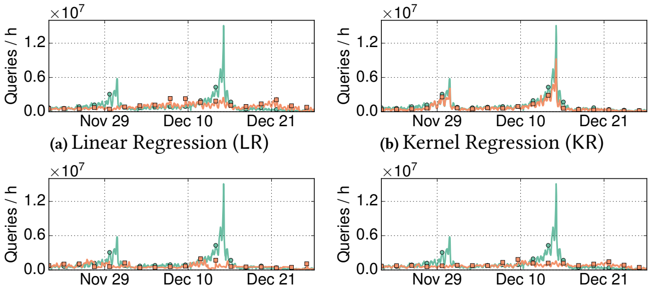
We now demonstrate how QB5000 uses its HYBRID models to predict queries' arrival rates for each cluster. We compare the predicted arrival rates of the queries belonging to the highest-volume cluster in the BusTracker application with their actual arrival rates for one-hour and one-week horizons. Figure 8 shows that the one-hour horizon prediction is more accurate than the one-week horizon. This is consistent with Figure 7 where the prediction accuracy decreases for longer horizons. The results also show that QB5000's models provide good predictions for both horizons since the predicted patterns of the arrival rates closely mimic the actual patterns.



(a) 1-Hour Horizon

(b) 1-Week Horizon

Figure 8: Prediction Results – Actual vs. predicted query arrival rates for the highest-volume cluster in the BusTracker workload with prediction horizons of one hour and one week.



(a) Linear Regression (LR)

(b) Kernel Regression (KR)

(c) Recurrent Neural Network (RNN)

(d) Ensemble (LR + RNN)

Figure 9: Prediction Results – Actual vs. predicted query arrival rates for the combined clusters in the Admissions workload with spike patterns.

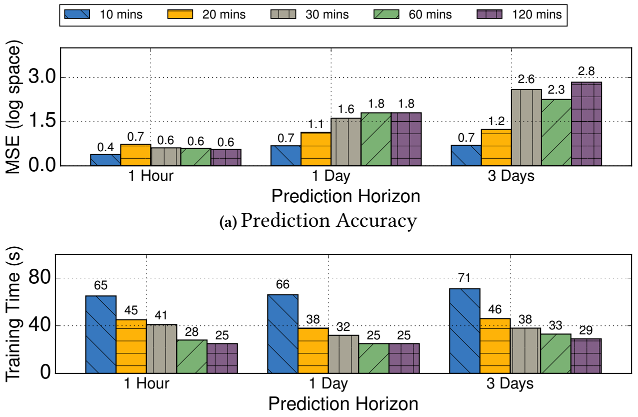
7.3 Spike Prediction Evaluation

Next, we evaluate QB5000's ability to forecast the growth and spike workload pattern. As mentioned in Section 6.2, QB5000 uses the full workload history aggregated into one-hour intervals to predict spikes. It tries to identify workload spikes one week before they will occur. The other settings are the same as in Section 7.2. We again evaluated all of the models and present the forecasting results from Nov 15 to Dec 31 (2017), which include two spikes from the admission deadlines on Dec 1 and Dec 15. The Admissions trace contains the similar spikes from the previous year (2016).

The results in Figure 9 show that ENSEMBLE and its two base models are unable to predict the spikes in the workload. Although they are not shown here, the other models also perform poorly in this scenario. The linear models' failure is likely due to the scarcity of the spike data and the limited capacity of the models. In contrast, models with higher capacity, like RNNs, may get caught in local optima and thus are also not able to produce good results. KR is the only model that successfully predicts the spikes. This is because its prediction is based on the distance between the test points and training data, where the influence of each training data point decreases exponentially with its distance from the test point. We further demonstrate why KR can separate data points with high query volumes from the other data points in Appendix B.

7.4 Prediction Interval Evaluation

We next evaluate the prediction accuracy and the training time of our ENSEMBLE forecasting models with varying prediction intervals. Since KR from HYBRID always uses one-hour intervals, it is unnecessary to evaluate it for other intervals. To provide a fair comparison, we compute the total prediction for each hour in the horizon by summing the predictions across the intervals within that hour. We then use these per-hour predictions to compute the MSE. For two-hour intervals, we calculate the prediction for each hour by dividing the interval that contains that hour into two.



(a) Prediction Accuracy

(b) Training Time (GPU)

Figure 10: Prediction Interval Evaluation – The average prediction accuracy and training time with different intervals for BusTracker.

The results in Figure 10a show that the accuracy of the models increases as the intervals become shorter. This is because shorter intervals provide more training samples and better information for learning the patterns in the data. Shorter intervals are especially beneficial for longer horizons since the relationship between the future and the past arrival rate is more complex. But shorter intervals increase the noise in the data and require more intervals to include the same extent of time. Thus, models trained on shorter intervals are larger (e.g., higher input dimension) and more complex.

Figure 10b shows the training time for each model at different intervals. It takes less time to train the models on longer intervals, which is expected since these models are smaller and less complex. Increasing the interval from 10 to 120 minutes reduces the training time by roughly 2.5× across all horizons, but decreasing the horizon provides only a minor training time reduction across all intervals.

One must consider these trade-offs when setting the interval, along with the planning capabilities of the target self-driving DBMS. We found that a one-hour interval works well for our operating environment in Section 7.6. As such, we use this interval for the remainder of the evaluation. Automatically determining the interval is beyond the scope of this paper and we leave it as future work.

7.5 Computation & Storage Overhead

To better understand the computational and storage overhead of QB5000, we instrumented its code to record the amount of time and space it spends in its four components:

Pre-Processor: The time to templatize a query and update its arrival rate history, and the amount of history data generated daily.

Clusterer: The time to update the clustering results once per day according to the latest history, and the size of the result data.

		Pre-Processor	**Clusterer**	**LR**	**RNN**	**KR**
COMPUTATION	Admissions	0.043ms/query	15s/day	GPU:0.3s CPU:0.3s	GPU:9s CPU:58s	GPU:0.16 CPU:0.18s
	BusTracker	0.05ms/query	3s/day	CPU:0.12s GPU:0.13s	GPU:33s CPU:221s	GPU:0.02s CPU:0.02s
	MOOC	0.048ms/query	12s/day	GPU:0.54s CPU:0.51s	GPU:5s CPU:18s	GPU:0.04s CPU:0.04s
STORAGE	Admissions	1.6MB/day	6.7KB	100B	28KB	11MB
	BusTracker	0.25MB/day	2.2KB	100B	28KB	1.9MB
	MOOC	1.4MB/day	0.8KB	100B	28KB	0.4MB

Table 4: Computation & Storage Overhead – The measurements for QB5000's different components.

LR Model: The time to train one LR model, and the size of the learned weights.

KR Model: The time to predict one test point with the KR model, and the size of the historical data maintained for the model.

RNN Model: The time to train one RNN model, and the size of the serialized model object from PyTorch, which contains both the model parameters and network structure.

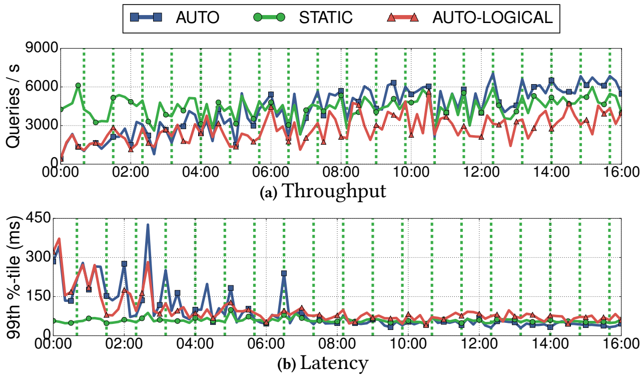
For LR, RNN, and KR, we use a one-hour interval and measure the model's time and space requirements over seven horizons. We report the average overhead of these horizons in our results.

Table 4 shows that all of QB5000's components have reasonable storage overhead. The results also show that training the RNN models is the most computationally expensive task. We stop training the RNN models when the validation accuracy stops improving. This means that the number of iterations performed and the training time differ per workload. Using a GPU improves the training time of RNN models by 3.6–7×. The overhead of training LR models is low compared to RNN models, and that the CPU/GPU performances are similar for the LR models since they are so simple. KR requires no training time and little testing time. But its testing time and training-data size increases linearly with the length of the workload history. QB5000 reduces this overhead by always aggregating the training for KR to one-hour intervals (see Section 6.2).

7.6 Automatic Index Selection

We now demonstrate how self-driving DBMSs can use QB5000's workload forecasts to make proactive optimizations that improve the system's performance. We integrate QB5000 with MySQL [1] and PostgreSQL [5] to process, cluster, and predict SQL workloads to automatically builds indexes for the predicted workload. We select a representative workload for PostgreSQL (BusTracker) and for MySQL (Admissions) for the evaluation. We use a 10 GB database for the Admissions and a 5 GB database for the BusTracker. We set each DBMS's buffer pool size to be 1/5 of the database size.

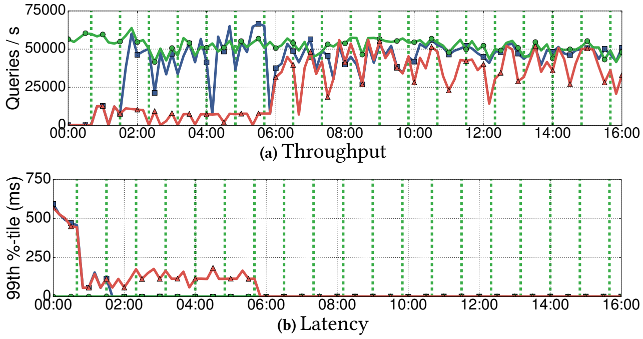
We use an index selection technique based on the one introduced by AutoAdmin to generate the set of indexes to build [12]. AutoAdmin first selects the best index for each query in a sample workload to form a candidate set of indexes. It then uses a heuristic search algorithm to find the best-bounded subset of indexes within the candidates. Instead of using a sample workload to generate the candidate indexes, we use the predicted workload of the three largest clusters generated by Clusterer. We note that the purpose of this evaluation is to demonstrate QB5000's ability to dynamically model and predict workloads, and not the efficacy of the index selection algorithm. We compare the performance of the automatic index selection (**AUTO**) against a static index selection method, which uses the same index selection algorithm but applies it to a fixed workload sample over the entire query history that is prepared manually before the start of the experiment (**STATIC**).



(a) Throughput

(b) Latency

Figure 11: Index Selection (MySQL) – Performance measurements for the Admissions workload using different index selection techniques.



(a) Throughput

(b) Latency

Figure 12: Index Selection (PostgreSQL) – Performance measurements for the BusTracker workload using different index selection techniques.

In both of the workloads, we initialize the DBMSs with the primary key and foreign key indexes defined in the applications' original schemas but remove all secondary indexes. We choose a random date to start the index selection process and train QB5000's forecasting models from the previous three weeks history. The system builds a new index at hourly intervals based on QB5000's real-time workload predictions for one-hour and twelve-hour prediction horizons. We weight the predictions from the one-hour horizon higher since models for shorter horizons are more accurate. We run the index selection technique for a 16 hour period. To control the total experiment time, we replay the workload 600× faster than the actual workload execution speed. That is, one second in our experiment represents 10 minutes of workload execution in the traces. During the experiment AUTO builds 20 indexes in total, and thus we set STATIC to also build 20 indexes before the experiment begins.

Figures 11 and 12 show the performance of MySQL and PostgreSQL using the AUTO and STATIC index selection techniques. The vertical green dotted lines indicate when the DBMS builds a new index. The results in Figure 11 show that the throughput and latency of MySQL executing the Admissions workload improve by 5× and 78% over the 16 hour period, respectively. AUTO initially performs worse than STATIC since it has not yet created any secondary indexes, but achieves 28% better throughput and 23% better latency by the end of the experiment. This is because AUTO selects four indexes that were not chosen by STATIC since it is able to leverage QB5000's forecasts. Figure 12 shows that PostgreSQL achieves 180× better throughput and 99% better latency for the BusTracker workload over this period. AUTO selects only one different index than STATIC, and thus their final performances are similar.

Automatic index selection is just one example of how QB5000's workload forecasting could be applied in a self-driving DBMS. There are other application scenarios that would serve as more powerful examples, such as resource provisioning. Such scenarios require a planning module for a self-driving DBMS, which requires a forecasting framework like what we present in this paper.

7.7 Logical vs. Arrival Rate History Feature

In this final experiment, we compare the effectiveness of the arrival rate history feature that QB5000's Clusterer uses against the logical feature (**AUTO-LOGICAL**). We repeat the same experiments from Section 7.6, except that we group templates based on the similarity of the logical structures of SQL strings. More specifically, the logical feature vector of a templates consist of the query type (e.g., INSERT, SELECT, UPDATE, or DELETE), tables that it accesses, the columns that it references, number of clauses (e.g., JOIN, HAVING, or GROUP BY), and number of aggregations (e.g., SUM, or AVG). We use the L2 distance to measure the similarity between two templates. We adjust the threshold ρ so that the volumes of the largest clusters are similar to those generated with the arrival rate history features.

The results in Figures 11 and 12 show that the DBMSs' throughput is ∼20% slower for AUTO-LOGICAL than for AUTO for both workloads. Figure 11 shows that the latencies are similar for AUTO-LOGICAL and AUTO for MySQL running the Admissions workload. But the results in Figure 12 show BusTracker in PostgreSQL has 38% higher latency with AUTO-LOGICAL. There are two reasons why logical features lead to worse index selection. The first is that the SQL queries are insufficient for determining whether two templates will have similar impacts on the system. The second reason is that templates within the same logical feature cluster may have multiple arrival rate patterns (including anomalies like one-time queries); this makes it more difficult for the Forecaster to identify these patterns and predict the trends according to the cluster centers.

8 RELATED WORK

We classify the previous work on workload modeling in systems into several categories: resource estimation and auto-scaling, performance diagnosis and modeling, shift detection, workload characterization for system design, and metrics prediction for queries.

There are works on automatically identifying the trends of the workload and scaling resources for provisioning. Das *et al.* proposed an auto-scaling solution for database services using a manually constructed hierarchy of rules [14]. The resource demand estimator derives signals from the DBMS's internal latency, resource utilization, and wait statistics to determine whether there is a high or low demand for resource. Their work focuses on short-term trends and estimates the demand for each resource in isolation. Other works investigated scaling resources proactively in cloud platforms [25, 46]. All of these methods estimate whether there will be a demand for each type of resource in the near future. In contrast, we model the query arrival rates for both short- and long-term horizons to support complex optimization planning decisions.

There is previous research on the modeling and diagnosis of DBMS performance. DBSeer predicts the answer of "what-if" questions given workload changes, such as estimating the disk I/O for future workload fluctuations [37]. The model clusters the workload based on transaction types and predicts the resource utilization of the system based on transaction mixtures. It is an off-line model that assumes fixed types of transactions. Our work not only looks at the current workload mixtures but also predicts the future workload. DBSherlock is a diagnostic tool for transactional databases that uses causal models to identify the potential causes of abnormal performance behavior and provides a visualization [55]. The modeling in this line of work aims to help the DBAs understand their system and identify bottlenecks. The authors in [38] use analytical models to continuously monitor the relationship between system throughput, response time, and buffer pool size.

Others have used Markov models to predict the next SQL statement that a user will execute based on what statements the DBMS is currently executing [31, 41]. The authors in [18] combine Markov models with a Petri-net to predict the next transactions. The technique proposed in [31] extends this model to detect workload shifts. Previous work combines these techniques with a workload classification method to model periodic and recurring patterns of the workload [28, 29]. These methods capture certain patterns in a workload, but none of them are able to predict the volume, duration, and changes of workloads in the future.

The workload compression technique in [11] shares a similar goal with our work. A set of SQL DML statements are compressed by searching which queries to remove from the workload with an application-specific distance function $D(q_i, q_j)$ for any pair of SQL queries q_i and q_j. This technique does not model the temporal pattern of the queries. Previous works also use set representation to model a database workload [48, 49]. They assume that all transactions arriving at the system belong to a fixed set of pre-defined transaction types. Various collected/aggregated run-time statistics are also used to analyze the structure and complexity of SQL statements and the run-time behaviors of the workload [56].

Other works have looked at how to predict the run-time metrics of specific queries. PQR uses a variant of the decision tree to determine which bucket the latency of a query belongs to using the query plan and system load metrics as input features [26]. Ganapathi *et al.* applied Kernel Canonical Correlation Analysis to project query plan vectors and performance metric vectors into the same subspace to estimate the metrics for a new query [23]. Such a prediction is invalidated when the hardware/database design changes.

9 CONCLUSION

We presented a forecasting framework that allows a DBMS to predict the expected arrival rate of queries in the future based on historical data. Our approach uses the logical composition of queries to reduce the number of queries that it needs to monitor, an on-line clustering method to group query templates with similar patterns together, and a hybrid learning method to forecast the query arrival rate. Our results show that QB5000 is effective in helping the DBMS select the optimal indexes for the target workload in real time.

REFERENCES

[1] MySQL. https://www.mysql.com/.

[2] OLTPBenchmark.com. http://oltpbenchmark.com.

[3] Open Learning Initiative. http://oli.cmu.edu.

[4] Oracle Self-Driving Database. https://www.oracle.com/database/autonomous-database/index.html.

[5] PostgreSQL. https://www.postgresql.org/.

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

[7] F. J. Baldan Lozano, S. Ramirez-Gallego, C. Bergmeir, J. Benitez, and F. Herrera. A forecasting methodology for workload forecasting in cloud systems. PP:1–1, 06 2016.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[9] H. J. Bierens. The nadaraya-watson kernel regression function estimator. In *Topics in Advanced Econometrics: Estimation, Testing, and Specification of Cross-Section and Time Series Models*, pages 212–247. Cambridge University Press, 1994.

[10] B. Boots, G. J. Gordon, and A. Gretton. Hilbert space embeddings of predictive state representations. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*, 2013.

[11] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *Proceedings of the 2002 International Conference on Management of Data*, pages 488–499. ACM, 2002.

[12] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 146–155, 1997.

[13] S. Chu, D. Li, C. Wang, A. Cheung, and D. Suciu. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 International Conference on Management of Data*, pages 1591–1594. ACM, 2017.

[14] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1923–1934. ACM, 2016.

[15] B. Debnath, D. Lilja, and M. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *ICDEW*, pages 11–18, 2008.

[16] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[17] C. Downey, A. Hefny, B. Boots, G. J. Gordon, and B. Li. Predictive state recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 6055–6066, 2017.

[18] N. Du, X. Ye, and J. Wang. Towards workflow-driven database system workload modeling. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 10. ACM, 2009.

[19] S. S. Elnaffar and P. Martin. An intelligent framework for predicting shifts in the workloads of autonomic database management systems.

[20] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, volume 98, pages 323–333, 1998.

[21] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231. AAAI Press, 1996.

[22] F. F/jr Informatik, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. 03 2003.

[23] 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 *International Conference on Data Engineering*, pages 592–603. IEEE, 2009.

[24] A. Ghosh, J. Parikh, V. S. Sengar, and J. R. Haritsa. Plan selection based on query clustering. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 179–190. VLDB Endowment, 2002.

[25] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management (CNSM)*, pages 9–16. Ieee, 2010.

[26] C. Gupta, A. Mehta, and U. Dayal. Pqr: Predicting query execution times for autonomous workload management. In *International Conference on Autonomic Computing*, pages 13–22. IEEE, 2008.

[27] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] M. Holze, C. Gaidies, and N. Ritter. Consistent on-line classification of dbs workload events. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1641–1644. ACM, 2009.

[29] M. Holze, A. Haschimi, and N. Ritter. Towards workload-aware self-management: Predicting significant workload shifts. In *26th International Conference on Data Engineering Workshops (ICDEW)*, pages 111–116. IEEE, 2010.

[30] M. Holze and N. Ritter. Towards workload shift detection and prediction for autonomic databases. In *Proceedings of the ACM first Ph. D. workshop in CIKM*, pages 109–116. ACM, 2007.

[31] M. Holze and N. Ritter. Autonomic databases: Detection of workload shifts with n-gram-models. In *East European Conference on Advances in Databases and Information Systems*, pages 127–142. Springer, 2008.

[32] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[33] T. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for real conjunctive queries with inequalities. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 80–89. ACM, 2006.

[34] L. I. Kuncheva and C. J. Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine learning*, 51(2):181–207, 2003.

[35] G. Lanfranchi, P. Della Peruta, A. Perrone, and D. Calvanese. Toward a new landscape of systems management in an autonomic computing environment. *IBM Systems journal*, 42(1):119–128, 2003.

[36] P. Martin, S. Elnaffar, and T. Wasserman. Workload models for autonomic database management systems. In *International Conference on Autonomic and Autonomous Systems*, pages 10–10. IEEE, 2006.

[37] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 International Conference on Management of data*, pages 301–312. ACM, 2013.

[38] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting dbms. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 239–248. IEEE, 2005.

[39] D. W. Opitz and R. Maclin. Popular ensemble methods: An empirical study. 1999.

[40] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.

[41] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *Proc. VLDB Endow.*, 5:85–96, October 2011.

[42] K. Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[43] R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 6(3):21–45.

[44] J. Rogers, O. Papaemmanouil, and U. Cetintemel. A generic auto-provisioning framework for cloud databases. In *International Conference on Data Engineering Workshops (ICDEW)*, pages 63–68. IEEE, 2010.

[45] A. Rosenberg. Improving query performance in data warehouses. *Business Intelligence Journal*, 11, Jan. 2006.

[46] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.

[47] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4), 1980.

[48] S. Salza and M. Terranova. Workload modeling for relational database systems. In *Database Machines*, pages 233–255. Springer, 1985.

[49] S. Salza and R. Tomasso. A modelling tool for the performance analysis of relational database applications. In *Proc. 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 323–337, 1992.

[50] B. Song, Y. Yu, Y. Zhou, Z. Wang, and S. Du. Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing*, pages 1–15, 2017.

[51] Z. Tang and P. A. Fishwick. Feedforward neural nets as models for time series forecasting. *ORSA journal on computing*, 5(4):374–385, 1993.

[52] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950.

[53] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[54] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[55] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1599–1614. ACM, 2016.

[56] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE Transactions on Software Engineering*, 18(4):347–355, 1992.

[57] Z.-H. Zhou. *Ensemble methods: foundations and algorithms.* CRC press, 2012.

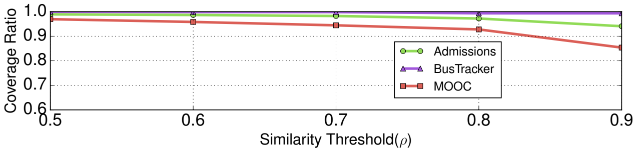


Figure 13: Cluster Coverage with ρ – The ratio between the volume of the three largest clusters and the total workload volume with different similarity threshold ρ.

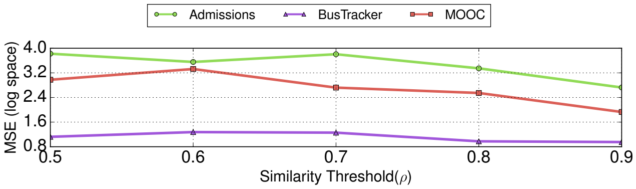


Figure 14: Prediction Accuracy with ρ – Normalized prediction accuracy for one-hour horizon with different similarity threshold ρ.

APPENDIX

A SENSITIVITY ANALYSIS OF ρ

We now the analyze QB5000's cluster coverage ratio and prediction accuracy for different ρ values. As discussed in Section 5.2, ρ is the sensitivity threshold used by the on-line clustering component to determine whether a template should belong to a cluster. In these experiments, we perform on-line clustering and arrival rate forecasting for values of ρ ranging from 0.5 to 0.9 for each of the three workloads. Setting ρ=1.0 results in every template having its own cluster, whereas setting it to 0.0 would group all the templates into a single cluster. We set both the prediction interval and the horizon to be one hour for the forecasting. We then measure the cluster coverage ratio and the prediction accuracy for the three highest-volume clusters.

Figure 13 shows the average volume ratio between the three largest clusters and the total workload volume for increasing values of ρ. The higher the ρ is, the more similar the templates within the same cluster are. But the number of templates contained in each cluster is also smaller, which means that the three largest clusters cover less of the workload when ρ is high. The results show that the coverage of the largest clusters is stable when ρ increases from 0.5 to 0.8 for all three workloads, but it drops when ρ reaches 0.9.

Figure 14 shows the sensitivity of the prediction accuracy to different values of ρ for the three largest clusters. The results show that the prediction accuracy improves as ρ increases for all three workloads. Since the similarity of the templates within the same cluster increases with ρ, the prediction results also improve since the centers of the clusters provide a more accurate representation of the arrival rate patterns captured by each cluster.

Given this analysis, we set ρ=0.8 in our evaluation since we find that it provides the best balance between the prediction accuracy and the coverage ratio for all three of the workloads.

B INPUT SPACE FOR SPIKE PREDICTION

We now demonstrate why KR is able to better predict spikes in a workload compared to the other forecasting models we considered. Recall from Section 6.1 that the prediction of KR for a given input is a weighted average of all of the training inputs, where the weights decrease with the distance between the given input and the corresponding training inputs in the kernel space. QB5000 uses the arrival rate history from the previous three weeks (aggregated into one-hour intervals) as the input for KR.

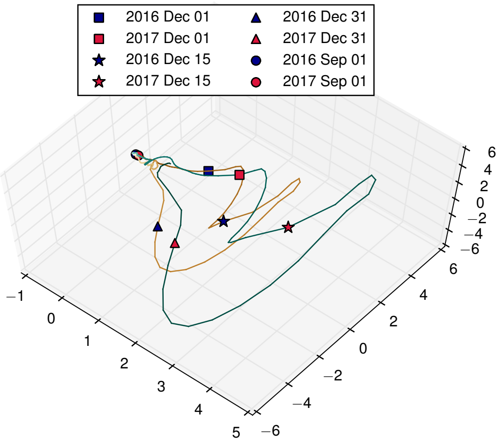


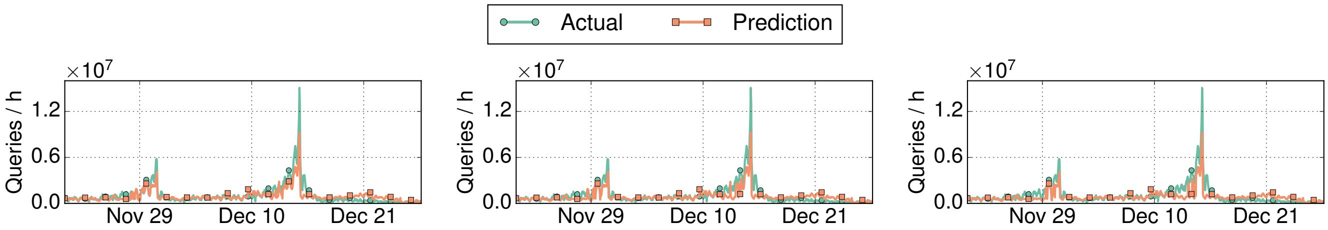
Figure 15: Input Space Time-Progress – The 3D-projected input space for the Admissions workload with spikes. The trajectory color follows dates.

Figure 15 shows the inputs for KR at each hour-interval in the Admissions workload trace. The input is projected into 3D space using a common dimensionality reduction technique, PCA [42]. Inputs on nearby dates have similar colors in the trajectory. From this figure, we see the points that correspond to the spike activity are clearly separated from the "normal" activity points. The trajectories from December 1st to December 31st in both 2016 and 2017 travel a much longer distance in the 3D space than in other months. The results also show that inputs that correspond to the same spikes in each of the two years have closer positions in the space. This demonstrates how kernel methods can easily recognize points with high query volumes and predict the future spikes based on the ones observed in the previous year. KR enables QB5000 to predict which dates spikes might occur on throughout the year without explicit domain knowledge.

C SENSITIVITY ANALYSIS OF γ

We next analyze the impact of the threshold γ on the prediction performance of QB5000's HYBRID models for workloads with growth and spike patterns. We use the same experimental settings as in Section 7.3.

Figure 16 shows the combined query arrival rate predictions from all clusters when the threshold γ is set to be 100%, 150%, and 200%. The model is able to predict the major spike patterns for all three settings of γ. When γ is lower, HYBRID uses the result from KR more often than ENSEMBLE, thus improving its ability to predict spikes. However, lower values of γ have increasingly negative impacts on the MSE of the predictions when no spike patterns are present in the workload. In our experiments, we found out that HYBRID has a negligible effect on the MSE of ENSEMBLE when $\gamma > 100\%$. Given this analysis, we set γ to 150% in QB5000.



(a) Threshold $\gamma = 100\%$

(b) Threshold $\gamma = 150\%$

(c) Threshold $\gamma = 200\%$

Figure 16: Prediction Results – Actual vs. predicted query arrival rates for the combined clusters in the Admissions workload with HYBRID (ENSEMBLE model corrected by KR).

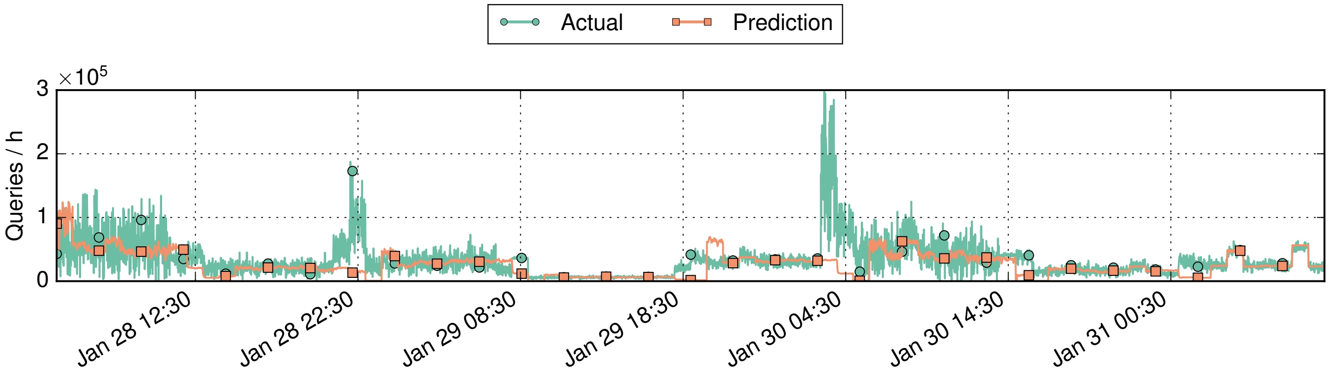


Figure 17: Prediction Results – Actual vs. predicted query arrival rates for a synthetic noisy workload.

D PREDICTING NOISY WORKLOADS

In Section 2.2, we presented three workload patterns that are prevalent in today's applications. But some workloads may not exhibit strong temporal patterns, and thus their arrival rates from the recent past may differ substantially from the present. This section extends our evaluation of QB5000 by demonstrating its ability to predict the arrival rates of a noisy workload without any temporal patterns to exploit. We use the HYBRID forecast models in this experiment. Such a workload represents a worst-case forecasting scenario for QB5000 since the arrival rates are unpredictable.

We constructed a synthetic workload trace that consists of benchmarks from the OLTP-Bench testbed that differ in complexity and system demands [2, 16]. We execute the following eight benchmarks consecutively with varying average arrival rates: Wikipedia, TATP, YCSB, Smallbank, TPCC, Twitter, Epinions, and Voter. Each benchmark is executed for 10 hours. We add white noise to the arrival rate that has a variance set to be 50% of its mean. We also inject random anomalies (i.e., spikes) into the arrival rate of the queries. As we described in Section 5.2, QB5000 monitors the ratio of previously unseen templates in the workload, and re-clusters the templates once it detects that the workload has switched from one to another. We set the prediction horizon to be one hour and the prediction interval to be one minute. Since each workload is executed for only 10 hours, QB5000 does not have enough training data to predict long horizons.

The results in Figure 17 show the combined predicted arrival rates of all clusters. Each time tick represents when the benchmark shifts from one to another. The results show that QB5000 is able to automatically identify these shifts in the workload and adapt to the new queries. This demonstrates that even when the workload is noisy and the arrival rate is unpredictable, QB5000 is still able to predict the average query volume most of the time.

E ACKNOWLEDGEMENTS

This work was supported (in part) by the National Science Foundation's Graduate Research Fellowship Program (DGE-1252522), the Intel Science and Technology Center for Visual Cloud Systems, Google Research Grants, and AWS Cloud Credits for Research. The authors thank the forever caustic Alex Smola for getting us started in the right direction on this project. We also thank Anthony Tomasic and Gene Hastings for providing us with the workload traces used in our analysis.

This paper is dedicated to the memory of Leon Wrinkles.
May his tortured soul rest in peace.