Sibyl: Forecasting Time-Evolving Query Workloads

Hanxian Huang
University of California San Diego
USA
hah008@ucsd.edu

Tarique Siddiqui
Microsoft Research
USA
Tarique.Siddiqui@microsoft.com

Rana Alotaibi
Microsoft Gray Systems Lab
USA
ranaalotaibi@microsoft.com

Carlo Curino
Microsoft Gray Systems Lab
USA
Carlo.Curino@microsoft.com

Jyoti Leeka
Microsoft
USA
Jyoti.Leeka@microsoft.com

Alekh Jindal
SmartApps
USA
alekh@smart-apps.ai

Jishen Zhao
University of California San Diego
USA
jzhao@ucsd.edu

Jesús Camacho-Rodríguez
Microsoft Gray Systems Lab
USA
jesusca@microsoft.com

Yuanyuan Tian
Microsoft Gray Systems Lab
USA
yuanyuantian@microsoft.com

ABSTRACT

Database systems often rely on historical query traces to perform workload-based performance tuning. However, real production workloads are time-evolving, making historical queries ineffective for optimizing future workloads. To address this challenge, we propose Sibyl, an end-to-end machine learning-based framework that accurately forecasts a sequence of future queries, with the entire query statements, in various prediction windows. Drawing insights from real-workloads, we propose template-based featurization techniques and develop a stacked-LSTM with an encoder-decoder architecture for accurate forecasting of query workloads. We also develop techniques to improve forecasting accuracy over large prediction windows and achieve high scalability over large workloads with high variability in arrival rates of queries. Finally, we propose techniques to handle workload drifts. Our evaluation on four real workloads demonstrates that Sibyl can forecast workloads with an 87.3% median F1 score, and can result in 1.7× and 1.3× performance improvement when applied to materialized view selection and index selection applications, respectively.

1 INTRODUCTION

Workload-based optimization is a critical aspect of database management, which tunes a database management system (DBMS) to maximize its performance for a specific workload. Consequently, a large number of performance tuning tools have been developed for workload-based optimization. While execution statistics may be sufficient for optimizing certain aspects of the system (e.g., buffer pool size), many optimizations require an understanding of semantics of the queries, necessitating a workload query trace as input. For instance, many commercial database products [2–5] support physical design tools such as Microsoft's AutoAdminn [11] and IBM's DB2 design advisor [56]. These tools automatically recommend physical design features, such as indexes, materialized views, partitioning schemes of tables, and multidimensional clustering (MDC) [41] of tables for a given workload of queries. In fact, many of these form the foundational pieces of the self-tuning [12], self-managing [40], and self-driving [16, 42] databases.

For workload-based optimization, the input workload plays a crucial role and needs to be a good representation of the expected workload. Traditionally, historical query traces have been used as input workloads with the assumption that workloads are mostly *static*. However, as we discuss in §2, many real workloads exhibit highly recurring query structures with changing patterns in both their arrival intervals and data accesses. For instance, query templates are often shared across users, teams, and applications, but may be customized with different parameter values to access varying data at different points in time. Consider a log analysis query that reports errors for different devices and error types: `"SELECT * FROM T WHERE deviceType = ? AND errorType = ? AND eventDate BETWEEN ? AND ?"`. Although the query template is recurring, the parameter values may be customized depending on the reporting needs, e.g., different types of devices may be analyzed on different days of the week, and the granularity of the time interval may switch from daily to weekly over weekends. Thus, optimization recommendations (e.g., recommended views) based solely on historical queries may not be effective for such time-evolving workload patterns. To adapt to evolving workloads, existing workload-based optimization tools can be modified and enhanced to accommodate workload changes, or alternatively, applied *without modification* by substituting the history workload with a workload that more accurately represents the anticipated query trace in the future. This motivates the need for forecasting future workload.

As depicted in Table 1, prior research works [8, 33, 38] have made efforts to forecast different aspects of future workload, but none have addressed the challenge of predicting future query traces with precise *query statements* in a future *time window* for *time-evolving* workloads. The Q-Learning approach proposed by Meduri et al. [38] takes the current query as input and predicts the next *one* query by forecasting query fragments separately and then assembling them into a complete query statement. For predicting the literals (or parameters) used in the query statement, Q-Learning only forecasts a bin of possible values instead of the exact values. QueryBot 5000 [33] and Tiresias [8] focus on forecasting the *arrival rate* of future query workload by training on patterns from historical query arrival rates. Hence, these techniques can only predict the

types of queries and the number of them expected in the future in a coarse granularity. Yet they do not generate the specific query statements for those workload-based optimization tools that require understanding the query semantics, especially in the context of time-evolving workloads.

Table 1: Sibyl vs. three closely related works.

	QueryBot 5000 [33] Tiresias [8]	Q-Learning [38]	Sibyl
What to predict	Query arrival rate	Next query	Future queries & arrival time
Methods	Hybrid-ensemble Learning	RNNs & Q-learning	Sibyl-LSTMs
Applications	Index selection	Query recommendation	Physical design tools
Time-evolving forecasting	✓	×	✓
Variable prediction windows	✓	×	✓
Forecast future query statement	×	✓†	✓
Forecast *precise* query parameter values	×	×†	✓

† Meduri et al. [38] forecast future query statement with coarse-grained parameter value ranges.

To address these limitations, we propose Sibyl, an end-to-end machine learning (ML)-based workload forecasting framework, which can accurately predict the query statements in a future time window for time-evolving workload. Different from a specific workload-based optimization technique, by addressing this broader but also more challenging workload forecasting problem, our aim is to enable a wide range of existing workload-based optimization tools that were originally designed for static workloads to be directly applied *without modification* for time-evolving workloads. We next highlight the major contributions of Sibyl.

1.1 Sibyl Contributions

We motivate the Sibyl design by qualitative and quantitative studies on four different real workloads (§2). We identify three insights from our observations that are shared by many existing studies [8, 27, 33, 48, 49, 55]: 1) real workloads are highly recurrent (sharing the same *query templates* but with different *query parameters*) as discussed earlier; 2) these recurrent queries often exhibit time-evolving behavior; and 3) they are highly predictable. Based on these insights, we first formalize a next-k workload forecasting problem that predicts the next k queries in the future. We develop an ML-based technique that integrates both query arrival time and query parameters into a common framework, and forecasts the *entire* statements of future recurrent queries and their arrival time. To this end, we develop a template-based featurization method that captures a large number of parametric expressions (§5.2), and combine stacked LSTM [20] with an encoder-decoder architecture [13], resulting in a model that we refer to as Sibyl-LSTMs (§5.3), to better capture both the temporal dependencies and the possible inter-dependencies among query parameters, for *per-template*, multi-variate, multi-step, time-series prediction.

We further identify a set of practical challenges in order to make the predictions usable for performance tuning tools. First, it is difficult to pre-determine the right number of queries (k) to forecast, to produce a useful workload for database optimization tools. Rather, a more practical version of the problem is to forecast the future queries for the next time interval Δt. But the number of queries expected in next Δt varies substantially across templates. Second, with one model per template, we need to limit the number of models to ensure the scalability of our design. To address these two issues, we extend the solution for the next-k problem to the next-Δt forecasting problem (§6). For templates with large expected numbers of queries, we 'cut' them into sub-templates in order to employ the predicted next-k queries to produce results for the next-Δt problem. To reduce the number of models, we 'pack' templates with a small expected number of queries into bins. We then build per-bin models (§6.4). Our approach not only yields accurate results for the next-Δt forecasting problem, but also reduces the number of Sibyl-LSTMs models by up to 23×, resulting in a significant reduction in both time and storage overhead by up to 13.6× and 6×, respectively.

A third challenge is that real-world workloads change dynamically, i.e., new templates may emerge, while old templates may become less relevant. The evolving patterns of literals may also change. To capture such changes, Sibyl adopts a feedback loop to handle workload shifts (§7). The feedback loop uses incremental learning to adapt the pre-trained Sibyl-LSTMs to the shifted workloads, achieving comparable accuracy to full training, with a negligible fine-tuning overhead.

We integrate the forecasting model with the feedback loop to develop the end-to-end forecasting solution of Sibyl. The paper also presents a common effectiveness measurement that can be utilized for many performance tuning tools (§8). We evaluate Sibyl on four real workloads (§9) and demonstrate its accurate forecasting ability. Furthermore, we apply Sibyl to two database applications on real workloads: materialized view and index selection, and our results show 1.7× and 1.3× improvement, respectively, using forecasted workloads compared to historical workloads.

The primary contribution of Sibyl is not proposing new ML algorithms, rather leveraging them for the problem of forecasting *entire* query statements, adapting feature selection and combining encoder-decoder architecture with LSTM for better accuracy, and improving scalability through template cutting and packing. To the best of our knowledge, Sibyl is the first framework that learns and forecasts the *entire* statements of future recurrent queries in various time spans.

2 OBSERVATIONS AND MOTIVATION

In this section, we present our observations from multiple real-world workloads, which serve as the motivation for the development of Sibyl. We provide a brief description of these workloads below along with detailed statistics (see Table 2).

Telemetry: This workload contains 14 days of point lookup queries from a decision support system used for querying telemetry data of Microsoft's products and services.

SCOPE: The workload contains 2 weeks of production jobs executed in Microsoft's SCOPE query engine. The jobs are written in the SCOPE query language, some of which contain UDFs (user defined functions) and UDOs (user defined objects).

BusTracker: The workload contains 57 days of queries from a mobile phone application for live-tracking of the public transit bus system, open-sourced by [33].

Sales: The workload contains 32 days of analytical query traces from Microsoft's internal revenue reporting platform. It consists of queries on purchase, sales, budget, and forecast data.

Table 2: The basic statistics of the workloads

	Telemetry	SCOPE	BusTracker	Sales
trace length (days)	14	14	57	32
# queries	2.6M	6M	25M	13.3K

```
Parameterized Query:
SELECT A.x,A.y, SUM(A.e) / SUM(A.z) AS val
FROM A
WHERE A.val1 = $1 AND A.val2 = $2 AND A.val3 IN $3
GROUP BY A.x, A.y
Parameters: $1='v1', $2='v2', $3=(1,2)
```

Figure 1: An example of parameterized query.

Observation 1: *Queries in real workloads are highly recurrent.*

Observing real production workloads, we found that most queries come from applications that use programmatic parameterized queries, with an example shown in Figure 1. Many queries in the workload share the same query template, while the parameter values vary. Building on definitions from prior work [8, 27, 33, 48, 49, 55], we call a query that shares the same template with at least another query as a *recurrent query*. As depicted in Table 3, we observe over 94.5% of queries in the four workloads are recurrent. The templates of the recurrent queries are *recurrent templates*. Table 3 also provides the number of recurrent templates in the four workloads. The BusTracker workload has a small number of recurrent query templates. In contrast, SCOPE exhibits a large number of recurrent templates [26]. We also observe the dominance of frequent recurrent queries, with more than 94% of total queries in the workloads having more than 20 recurrences.

Observation 2: *Recurrent queries often evolve over time.*

In real workloads, the parameter values in recurrent templates can change dynamically over time. A recurrent template with at least one changing parameter value is called an *evolving template*. Query instances belonging to evolving templates are referred to as *evolving queries*. Table 4 shows the percentage of evolving templates among the recurrent templates, and the proportion of evolving queries among the recurrent queries in the four workloads. In Telemetry, SCOPE, and BusTracker, the majority of recurrent queries are evolving. In Sales, evolving queries represent 26.6% of recurrent queries but account for 57.4% of the total workload execution time, indicating their higher cost.

We further analyze parameter value changes with query arrival time and identify common patterns: (a) trending pattern: increasing, decreasing, or level trends; (b) periodic pattern: regular pattern with fixed interval (e.g., hourly, daily, weekly); (c) combination of trending and periodic patterns; (d) random pattern (no regular or predictable pattern). Examples of these patterns are visualized in Figure 2 using the Telemetry workload.

Observation 3: *Recurrent queries are highly predictable.*

We further study the predictability of parameters in the recurrent queries, by employing the widely-used approximate entropy

Table 3: Recurrent queries in the workloads

	Telemetry	SCOPE	BusTracker	Sales
% recurrent queries	99.9%	94.5%	99.9%	96.9%
# recurrent templates	2157	168197	258	1143

Table 4: Time-evolving queries in the workloads

	Telemetry	SCOPE	BusTracker	Sales
% evolving templates	96.6%	97.3%	99.8%	0.2%
% evolving queries	99.9%	99.4%	99.9%	26.6%

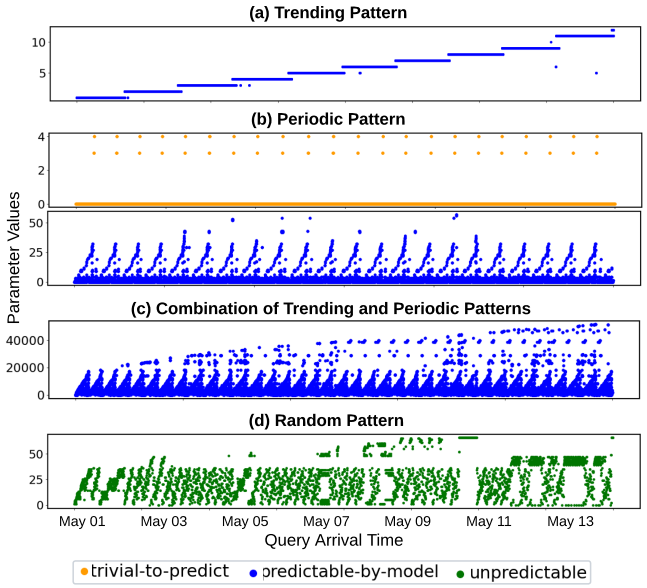

Figure 2: Characterizing time-evolving patterns and their predictability using queries from the Telemetry workload.

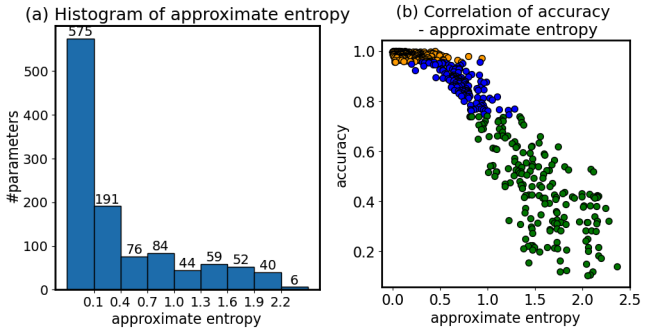

Figure 3: (a) The histogram of ApEn on parameters of Telemetry workload. (b) The negative correlation between parameter forecasting accuracy (using vanilla LSTM) and ApEn.

(ApEn) [44] metric to quantify the unpredictability of time-series parameter data. A lower ApEn indicates higher predictability. Figure 3(a) shows the histogram of ApEn of all parameters of the Telemetry workload. We observe some parameters have low ApEns, even smaller than $5e^{-4}$ in our study, while some have relatively higher ApEns, greater than 1.0.

Table 5: The predictability of workloads

	Telemetry	SCOPE	BusTracker	Sales
trivial-to-predict	21.3%	23.1%	13.7%	9.1%
predictable-by-model	68.5%	62.0%	78.7%	81.8%
unpredictable	10.3%	14.9%	7.6%	9.1%
total predictable†	89.7%	85.1%	92.4%	90.9%

† Sum of the trivial-to-predict and the predictable-by-model categories

Unfortunately, it is well known that there is no universally defined threshold value for classifying a variable as unpredictable using ApEn [14, 31]. It all depends on the context and the threshold is determined empirically for a particular scenario. However, not surprisingly, we found that ApEn is negatively correlated with parameter prediction accuracy by an ML model. Figure 3(b) illustrates this relationship for a widely-used vanilla LSTM model [20]. Empirically, we tried various ML models[1] for predicting each parameter, and get the best accuracy value Acc_{max} for each parameter. To quantitatively identify unpredictable parameters, we introduce an accuracy threshold τ, and treat parameters with Acc_{max} lower than τ as unpredictable parameters. We empirically set $\tau = 75\%$ based on our expectation on parameter accuracy. Among the predictable parameters, we further identify the trivial-to-predict ones, which have simple patterns, e.g., repeating only a very small number of possible values. These can be easily predicted by simple heuristic based methods. The remaining predictable parameters are evolving over time with more complex patterns and require sophisticated models to achieve good forecasting accuracy. We use orange, blue, and green colors to mark the three categories: **trivial-to-predict**, **predictable-by-model**, and **unpredictable** in Figure 2. Table 5 shows the percentages of these categories for the four workloads. Overall, our analysis indicates the high predictability of parameters in recurrent templates, with a significant portion of them non-trivial to predict, thus necessitating ML models for assistance.

Note that the four studied workloads represent a diverse set of workloads from different database systems with distinct query volumes (ranging from 13K to 25M) and varying numbers of query templates (ranging from 258 to 168K), as well as a mixture of operational queries (Telemetry and BusTracker) and analytical queries (SCOPE and Sales). Yet, the aforementioned observations remain true for all four workloads. In addition, these observations align with existing works [8, 27, 33, 48, 49, 55] that emphasize the recurrent and predictable nature of real workloads. In fact, Sibyl's target applications, the various existing workload optimization tools, already implicitly assume workload predictability, as it simply doesn't make sense to tune performance on random workloads. Therefore, Sibyl is designed on the same premise of this common scenario.

3 PROBLEM STATEMENT

We begin by introducing key concepts to formally define the workload forecasting problem. A *query*, denoted by q, refers to a statement expressed in SQL or a similar declarative language. A *workload*, denoted by W, is a bag (i.e. multi-set) of queries. Most workload-based optimizations tools [11, 56] take W as input, along with other constraints (e.g. storage constraint), and produce a recommendation of target features (e.g. indexes, materialized views, partitioning schemes, or MDC strategies) to optimize the total cost of W. These tools are usually executed at regular intervals, such as hourly, daily, or weekly. Hence, forecasting a representative query workload for the upcoming time interval necessitates considering the arrival time of queries. As a result, we define a *timed-workload*, denoted as TW, as a time series of queries, where each query q_i in the workload has an associated arrival time t_{q_i} indicating when the query is issued. Queries in the timed-workload are ordered based on their arrival time, i.e., $t_{q_1} \leq t_{q_2} \leq ... \leq t_{q_n}$. Hence, the timed-workload can be represented as $TW = [(q_1, t_{q_1}), (q_2, t_{q_2}), ..., (q_n, t_{q_n})]$. Given a timed-workload TW, its corresponding workload W can be obtained by removing the arrival times and converting the sequence into a bag. When there is no ambiguity, we also refer to a timed-workload as a workload for simplicity.

We now present the formal definitions of two workload forecasting problems: next-k forecasting and next-Δt forecasting.

Definition 3.1. *(Next-k Forecasting)*
Given a timed-workload $TW = [(q_1, t_{q_1}), (q_2, t_{q_2}), ..., (q_n, t_{q_n})]$, *the next-k forecasting problem predicts the next k future queries as:*

$$F_k(TW) = [(q_{n+1}, t_{q_{n+1}}), (q_{n+2}, t_{q_{n+2}}), ..., (q_{n+k}, t_{q_{n+k}})]$$

Definition 3.2. *(Next-Δt Forecasting)*
Given a timed-workload $TW = [(q_1, t_{q_1}), (q_2, t_{q_2}), ..., (q_n, t_{q_n})]$, *the next-Δt forecasting problem predicts the queries in the next time interval of size* Δt *as:*

$$F_{\Delta t}(TW) = [(q_{n+1}, t_{q_{n+1}}), (q_{n+2}, t_{q_{n+2}}), ..., (q_{n+\sigma}, t_{q_{n+\sigma}})]$$

where $t_{q_{n+\sigma}} < t_{q_n} + \Delta t \leq t_{q_{n+\sigma+1}}$.

In contrast to the fixed number of queries to predict in next-k forecasting, the next-Δt forecasting deals with the prediction of queries within a fixed time interval. In real-world applications, determining the target time interval of the expected future queries is often more feasible and useful than the target number of future queries. Database tuning tasks are typically performed at regular intervals, such as hourly, daily, or weekly. Thus, having knowledge of the expected workload in the target time interval is crucial for optimizing database performance.

The next-Δt forecasting problem is more challenging than the next-k forecasting problem as the number of queries to predict, σ, is not known in advance and must be determined based on the arrival time of the predicted queries. Operationally, the forecasting model needs to keep predicting queries until it sees a query with an arrival time exceeding the next Δt time interval.

It is important to note that even though many workload optimization tools do not require time information, both forecasting problems defined above generate timed workloads. The predicted arrival time of each query plays a crucial role in forecasting, as it provides constraints for the next-k (selecting the next k queries ordered by the arrival time) and next-Δt (producing queries with bounded arrival time) prediction challenges. After the forecasting process, the timed workloads can be easily converted into regular workloads before being utilized by workload optimization tools.

We want to emphasize that this work primarily focuses on analytic workloads targeting workload-based optimization applications. While our techniques are general enough, we leave forecasting non-query statements (DML and DDL) as future work.

[1] We tried Random Forest, vanilla LSTM, and our own Sibyl-LSTMs.

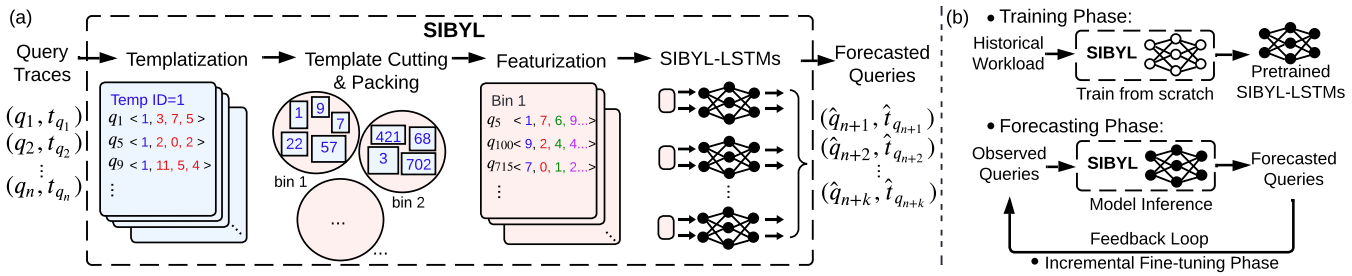

Figure 4: SIBYL Overview. (a) shows the four components of SIBYL for next-Δt forecasting. (b) shows the three phases of SIBYL.

4 SIBYL OVERVIEW

SIBYL is an ML-based workload forecasting framework that takes the past queries as input and outputs the future queries, performing time series prediction. Forecasted queries can be seamlessly integrated into existing database tools for tuning tasks.

We first tackle the simpler next-k forecasting problem, by developing a template-based featurization method and combining stacked LSTM with an encoder-decoder architecture to create SIBYL-LSTMs. To address practical challenges such as handling a large number of queries in the required prediction interval and the scalability of the design, we extend the solution for the next-k problem to the next-Δt forecasting problem, and implement template cutting and packing algorithms to reorganize templates into bins. This allows us to build per-bin models, resulting in a more practical and scalable design. Figure 4(a) depicts the overall architecture of SIBYL for solving the next-Δt problem, and the detailed design of each component will be introduced in §5 and §6.

As shown in Figure 4(b), SIBYL has the following three phases:

Training: it featurizes the past queries and their arrival time, and trains ML models from scratch. The training is only performed once. We assume that the historical workload provides sufficient training data for accurate predictions using ML models.

Forecasting: it continuously receives recent queries from the workload traces and employs the pre-trained ML models to predict the next-k or next-Δt queries with their expected arrival time in the future workload. The forecasted workload can be passed to database tuning tools as input to perform optimization.

Incremental fine-tuning: it monitors model accuracy and detects workload shifts (e.g., new types of queries emerging in the workload) via a feedback loop. It adjusts its models efficiently by fine-tuning incrementally on the shifted workload, without retraining from scratch.

Note that the three phases are not on the critical path of workload-based optimization applications. They are offline steps to prepare the inputs for these applications to tune database performance.

5 NEXT-k FORECASTING MODELS

We first solve the next-k forecasting problem. A simplistic approach would be constructing a single global model to learn on all the queries in a workload. However, this method has various limitations. Firstly, as demonstrated in Table 3, actual workloads frequently consist of different types of queries (different query templates), making it challenging to featurize the extensive range of query logic. Secondly, it results in a large number of features and a complex mixture of patterns from all templates that the model needs to learn. Thirdly, such a global model necessitates a substantial amount of training data and can be extremely expensive to train.

Based on the observations outlined in §2, real-workload queries are highly templatized, and the query parameters in each template frequently exhibit time-evolving patterns and are very predictable. Rather than creating a global model for the entire workload, it is more reasonable to build a model for each template. Specifically, we perform query templatization (§5.1) and group the queries in a workload based on their templates. For each template, we collect queries of the template, featurize the queries (§5.2), and train a model (§5.3) to forecast future k queries for this template and their arrival time. This high-level idea of constructing a model per template is also utilized in existing works such as [33, 47, 54]. Note that k is the forecasting window size, indicating how many queries are expected to forecast. We defer the discussion on practical considerations in selecting k to §6. To get the forecasting results for the entire workload, we collect the forecasted queries for all templates together and sort them by the forecasted arrival time to produce the final k queries. We next elaborate the design details.

5.1 Query Templatization

In this pre-processing step, we group the queries in the given workload based on their query templates. To obtain the template from a given query, SIBYL parses the query to create an abstract syntax tree (AST) and transforms the literals in the AST into parameter markers [6][2]. Our templatization technique can extract literals in any part of the query, including UDFs, UDOs, and Common Table Expressions (CTEs). To ascertain the equivalence of two query templates, we verify whether their ASTs are identical via strict matching. Then, SIBYL associates all parameter bindings in a query to its corresponding template. We rely on the AST representation of queries for templatization, as it simplifies the manipulation of the query structures. But our approach for evaluating template equivalence could be easily extended, e.g., using canonical representations of filter expressions. We leave such extensions for future work.

5.2 Feature Engineering

Featurization of queries with arbitrary complexity is a hard problem. Existing plan-based [34, 35] or token-based [25] featurization methods have made significant progress on this front, but can still only support limited query constructs. However, in our setting,

[2]We note that the extraction of templates does not take into account any query rewriting or optimizations.

by leveraging query templates, we can *circumvent* this challenge. Queries with the same template share identical query structure but differ only in their parameter values. Therefore, we can use the template identification (template id) to capture the query structure, while hiding the query complexity within the template.

Each query can be featurized using template id and parameter values. This also simplifies the query reconstruction process by merely filling in the parameter values into the corresponding template to reconstruct a predicted query. In addition, there are two requirements for featurization that we need to satisfy in our setting.

- **R1**: To better learn the time-evolving patterns, time-related features, either the query arrival time or any query parameter value of Date-Time type, need to be encoded in a way to capture the periodicity (such as year, month, week, day, etc.) and seasonality (including weekdays, weekends, holidays, etc.) of time.
- **R2**: Since future observations in a time series problem are often dependent on past observations, featurization should capture the relationship between consecutive queries.

Below, we describe how we featurize a query, including its arrival time to meet these requirements.

5.2.1 ***Query Feature Vector.*** In a per-template model, only the parameter values require encoding in the query feature vector. Furthermore, to satisfy R2, we also encode the difference between the parameter values of the current query and its predecessor. Here the major task is on dealing with a rich set of data types for query parameters. We utilize the table schema and parameter values to deduce the data types of the parameters. Subsequently, we encode each parameter according to its corresponding data type.

- **Numerical types**. The parameters of numerical-type, e.g., *Int*, *Long*, *Double* and *Float*, are encoded by their numerical values.
- **Categorical types**. The parameters with *String*, *Char*, *Boolean* types are encoded as categorical values. For each parameter, we collect all possible values from the training data, and assign an identical integer value to each category. To deal with high cardinality categorical features, we then apply the feature hashing technique [53] to encode the categorical values.
- **Date-Time type**. Special treatment is given to date-time parameters to ensure R1. The value of date-time is dissected into individual components such as year, month, day, hour, minute, and second. Furthermore, we incorporate additional derived features such as identification of weekends, public holidays, the season of the year, and so on.
- **Set type**. Set parameters often appear in the IN or VALUES clauses. In our study, we observe a fixed set of values recurring which are extracted as categorical values. We plan to explore alternative encoding methods in the future.

5.2.2 ***Arrival Time Feature Vector.*** Analogous to processing date-time parameters, we decompose the query arrival time into its constituent parts, including year, month, day, hour, minute, and second, thus satisfying R1. This approach allows the ML models to forecast all the features related to time and reconstruct future arrival timestamps. In addition, to ensure R2, we also featurize the difference between the arrival time of successive queries.

5.2.3 ***Input Feature Map for ML Models.*** As we formulate the workload forecasting problem as a time series prediction problem, the input to each ML model is a feature map with a sequence of feature vectors until the current timestamp $(fv_1, fv_2, ..., fv_n)$ ordered by query arrival time. Each feature vector fv_i comprises the query feature vector concatenated with the corresponding query arrival time feature vector. The output of each ML model is a sequence of the next k feature vectors $(fv_{n+1}, fv_{n+2}, ..., fv_{n+k})$, which are used to reconstruct the next k queries with their respective arrival times. The input feature map enables the ML models to capture the interrelationship among features within and between queries and learn from these connections, resulting in accurate predictions.

5.3 Forecasting Models

We now outline the ML models for solving the next-k forecasting problem. With various options available for time series forecasting, we have considered and evaluated the following two models:

Random Forest (RF). RF is an ensemble learning method [46] widely used for classification and regression problems. With its simplicity and popularity, it is tempting to apply RF to our forecasting problems. The conventional use of RF enables the prediction of only the next single query. To adapt RF for predicting next-k queries $q_{n+1}, q_{n+2}, ...q_{n+k}$, we use the past k queries $q_{n-k+1}, q_{n-k+2}, ...q_n$. More specifically, RF predicts q_{n+1} using q_{n-k+1}, q_{n+2} using q_{n-k+2}, and so on. However, as we will show in §9, even with this adaption, RF is still not a good solution for our problem.

Long Short-Term Memory Networks (LSTM). LSTM is a variant of the recurrent neural network (RNN) [37], designed to learn a sequence of data with both short-term and long-term dependencies. Its ability to capture temporal patterns makes it ideal for solving time-series prediction problems. However, vanilla LSTM model, i.e., a single-layer LSTM as shown in Figure 5(a), has limited model capacity to capture complex relationships among features. Our task involves taking a sequence of historical queries and forecasting a sequence of future queries, with each query consisting of multiple parameters, making it a *multi-variate, multi-step, time-series sequence-to-sequence* learning problem. While LSTM is good at capturing temporal dependencies, it processes each variable independently and does not directly capture the inter-dependencies between them. Thus, it is not a good solution for multi-variable problem that involves complex relationships among variables. To address this limitation, we combine LSTM with the advanced encoder-decoder architecture [13], resulting in a model that we refer to as Sibyl-LSTMs, to better capture both the temporal dependencies and the interrelationships among variables, as shown in Figure 5(b).

The encoder-decoder architecture is widely used in NLP domain [13, 32] due to its ability to capture the context more effectively. As shown in Figure 5(b), the encoder processes the input sequence and maps features into encoder states. The encoder states are the latent representations that summarize the entire input sequence encoding important information and dependencies from the input. The encoder states are then used to guide the generation of the output sequence by the decoder. This architecture allows the model to capture complex relationships between the input and output sequences and performs better on complex evolving patterns.

As will be shown in §9, Sibyl-LSTMs produces more accurate and stable results on different problem settings (i.e., various k and

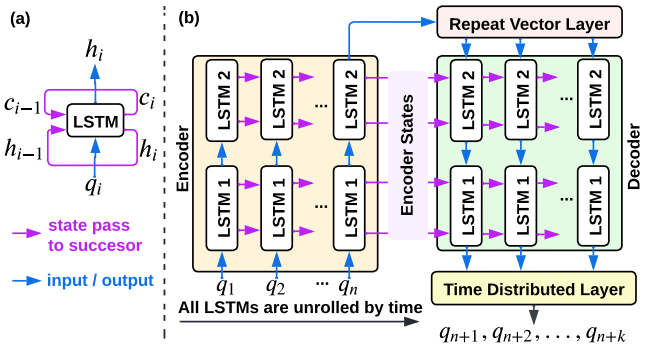

Figure 5: (a) One LSTM layer. (b) SIBYL-LSTMs.

Δt settings) than RF and vanilla LSTM. Therefore, we select SIBYL-LSTMs as our preferred model. We note that the existing forecasting models in [33, 38] are not directly suitable for our needs: QueryBot 5000 [33] performs single-variable, multi-step forecasting, while [38] conducts multi-variable, single-step forecasting. Finally, large language models [15, 45, 51] could also be considered. These models excel in complex sequence-to-sequence learning but have more parameters and longer training time. Based on our evaluation in §9, SIBYL-LSTMs provides sufficiently accurate results, so we chose to focus exclusively on SIBYL-LSTMs.

5.3.1 ***SIBYL-LSTMs.*** Figure 5 shows how we combine LSTM with an encoder-decoder architecture to build SIBYL-LSTMs.

We first briefly introduce how one LSTM layer works, as shown in Figure 5(a). At each time step i, the LSTM takes an input q_i and produces a hidden state h_i. Additionally, it passes two types of states to the next time step: the hidden state h_i, which governs short-term memory, and the cell state c_i, which governs long-term memory. The LSTM works in a recurrent manner, which can be viewed as multiple copies of the same network at different time steps, with each passing the information (the states) to a successor. Figure 5(b) depicts the unrolling of the LSTM over multiple time steps, allowing the model to retain long-term and short-term information and enabling it to reason about prior data to inform subsequent ones.

Integrating LSTM with the encoder-decoder architecture, we design SIBYL-LSTMs in Figure 5(b). We first deploy a stacked-LSTM with two LSTM layers as the encoder. The stacked LSTM is deeper and provides higher model capacity (more trainable parameters) to capture more information at different levels and model more complex representation in data, compared to the vanilla LSTM. The first layer of the encoder (*LSTM 1*) takes featurized query q_i at time step i, together with the previous cell state c^1_{i-1} and hidden state h^1_{i-1}. The second layer (*LSTM 2*) takes the output from *LSTM 1* as input, along with its previous cell state c^2_{i-1} and hidden state h^2_{i-1}. The encoder recursively learns on the entire input sequence and generates the output. The final encoder state, which summarizes the input sequence, consists of h^1_n, c^1_n, h^2_n, and c^2_n. We then employ a repeat vector layer to replicate the encoder output into k copies. The decoder module comprises two LSTM layers that are initialized with the final encoder state. The decoder takes the encoder output together with the previous states as the input, to generate the hidden state output for each of the k future steps. Finally, we apply the time distributed layer, which is a dense layer, to separate the results into each future time step. The output of SIBYL-LSTMs is a sequence of feature vectors with length k which can be decoded into k future query statements and their arrival time.

6 NEXT-Δt FORECASTING MODELS

We now address the practical challenges in adapting next-k forecasting to next-Δt forecasting, for a fixed target Δt value.

6.1 Challenges

Challenge 1: A required forecasting size exceeds a feasible k. For a target time interval Δt, a naive approach would be to use the next-k forecasting method (§5) and set a k sufficiently large so that $k \geq \sigma$, where σ is the number of queries arriving in next Δt as defined in DEFINITION 3.2. Specifically, we refer to the number of queries for template $temp_i$ in Δt as its *template size* in Δt, denoted as $\sigma_i(\Delta t)$. However, as we will discuss below, it is not always practically feasible to fulfill $k \geq \sigma_i(\Delta t), \forall temp_i$.

First, the output window size k of a sequence-to-sequence learning model is usually decided empirically rather than arbitrarily chosen, considering several factors: (1) A larger k brings more computation complexity and memory overhead. Given a specific machine to deploy SIBYL-LSTMs and a certain-sized workload, the maximum feasible k is decided and bounded by the machine resources. (2) The model accuracy degrades with arbitrarily increasing k, as it is harder for a model to capture the patterns in an extremely long sequence. The state-of-the-art sequence-to-sequence learning models [15, 45] typically adopt a feasible $k \in [128, 1024]$. (3) A larger k requires more training data, thus may lead to the problem of inadequate data available for training. Therefore, a maximum feasible k in SIBYL is decided by an exhaustive search of the available resources, workload sizes, and model accuracy expectation once for a given experimental set-up. Once k is decided, it is fixed during the whole training and inference process. A change of k leads to a new model and requires model retraining from scratch.

Secondly, given k is limited and fixed, the required number of queries to forecast for next-Δt can be larger than k. For example, a physical design tool may require next one-day's queries to perform optimization, i.e., $\Delta t = 1day$. As shown in Figure 6, the required forecasting window size for a large sized template in the Telemetry workload can be up to 10,500 to cover next day's queries. In this case, directly applying the next-k solution is impossible with the above discussed feasible setting of $k \in [128, 1024]$. This leads to the *first challenge*: addressing *large templates*, i.e., templates with a substantially larger number of expected queries than k in next Δt.

Challenge 2: Per-template model solution is not efficient for small templates and not scalable. Conversely, the *second challenge* involves the long tail of *small templates*, i.e., templates with a very limited number of expected queries in next Δt. For instance, Figure 6 shows there are 1,893 templates in the Telemetry workload, which is 87.8% of all the templates, with fewer than 50 queries per day, much smaller than a common setting of k, i.e., $k \gg \sigma_{small\_temp}(\Delta t)$. Directly deploying the next-k solution is inefficient because it does not make the best use of model capacity. Moreover, in real workloads, the number of templates can be quite large, as shown in Table 3. Training a model for each template

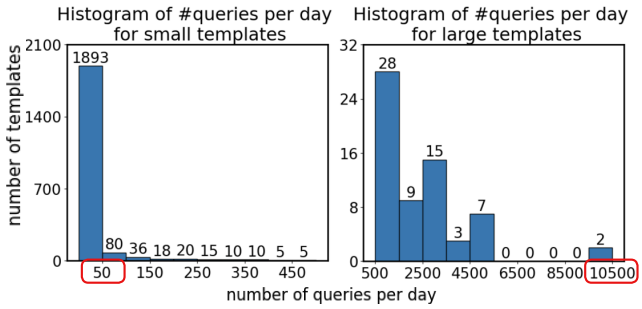

Figure 6: An example from the Telemetry workload where the average number of queries per day is imbalanced.

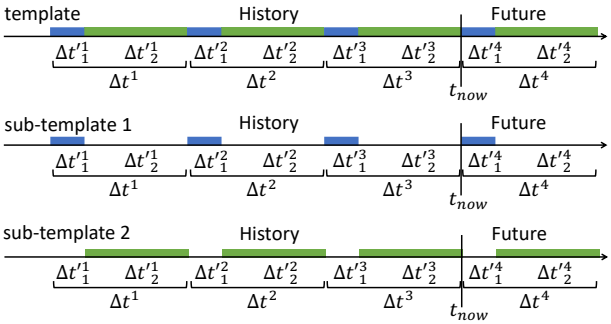

Figure 7: Illustration of template cutting.

is not scalable, resulting in a vast number of models to maintain, significant training time, and model storage overhead (see Table 9).

6.2 Template Cutting

We first address the problem with large templates with sizes greater than k in Challenge 1. We first attempted a straightforward method of repeatedly forecasting the next k queries using previously predicted k queries as input for the next round of prediction. This process continues until there are enough predicted queries to cover the next Δt interval. However, we empirically find that this approach results in poor accuracy, particularly as the number of prediction iterations increases. For instance, with $k = 1000$, the 10 largest templates in the Telemetry workload require 4 to 10 rounds of iterative predictions to cover a day's forecast, yielding an average accuracy of only 41.7%. As a result, we opted for a different approach.

A continuous sub-sampling of a regular pattern in time series data still follows a regular pattern [39]. This is because the sub-sampling process is essentially selecting a subset of the original pattern at regular intervals. As long as these intervals are consistent, the resulting pattern will also be regular. Based on the above observation, we propose template cutting to split the group of queries for a large template into sub-groups, a.k.a. *sub-templates*, so that each is no larger than k. We cut templates based on the arrival time, as illustrated in Figure 7. Specifically, we divide the Δt interval into consistent sub-intervals, $\Delta t'_1, \Delta t'_2, ..., \Delta t'_s$, and use them to divide the group of queries. Note that the sub-intervals do not have to be equal length in time. Figure 7 shows two sub-intervals $\Delta t'_1$ and $\Delta t'_2$ of Δt. After splitting, we now have two sub-templates. The first only contains queries that fall into the $\Delta t'_1$ time-frame of every Δt interval (represented as blue bars in the figure), whereas the second contains queries in the $\Delta t'_2$ time-frame of every Δt interval (represented as green bars). Assuming an accurate model for each sub-template, it will learn the new patterns (only having queries in its corresponding sub-interval) in the sub-template, and predict future queries based on the new patterns (only predicting queries in its corresponding sub-interval). Combining the forecasted queries from both models will result in the forecasted queries for the entire Δt interval.

For template cutting, we need to estimate the number of queries (see §6.5) that will arrive for a given template in next Δt, denoted as $\widetilde{\sigma}_i(\Delta t)$. Since we need to cut Δt into smaller sub-intervals, we first choose a sufficiently fine-grained time window $\Delta t''$ to evenly divide Δt into finer intervals $\Delta t''_1, \Delta t''_2, ..., \Delta t''_m$. For example, if Δt = 1 day, we may pick $\Delta t''$ = 1 hour and have 24 small time intervals. We first use $\widetilde{\sigma}_i(\Delta t)$ to identify templates larger than k. For each such template, we examine the finer time granularity $\Delta t''$ and find the cutting points of sub-templates in the boundary of finer time intervals. As depicted in Algorithm 1, it begins with $\Delta t''_1$ and greedily searches for a cutting point that results in the largest sub-template with a size no greater than k by utilizing the estimated $\widetilde{\sigma}_i(\Delta t''_j)$. After the cut is made, it moves on iteratively to search for the cutting point for the next sub-template.

Algorithm 1 Template Cutting Algorithm

Input: k; $\widetilde{\sigma}_i(\Delta t''_j)$ and $\widetilde{\sigma}_i(\Delta t)$ for template $temp_i$.
Initialize the final sub-template set $S = \emptyset$.
for $temp_i$ in all templates **do**
  Initialize the sub-template set for $temp_i$: $s_i = \emptyset$
  if $\widetilde{\sigma}_i(\Delta t) > k$ **then**
    Current_sub_template $C = \emptyset$
    Current_remaining_size $R = k$
    for $\Delta t''_j$ in Δt **do**
      if $\widetilde{\sigma}_i(\Delta t''_j) \leq R$ **then**
        # fit into the current sub-template
        $C = C \cup \{temp_i(\Delta t''_j)\}$, $R = R - \widetilde{\sigma}_i(\Delta t'')$
      else
        # initialize a new sub-template
        $s_i = s_i \cup \{C\}$
        $C = \{temp_i(\Delta t''_j)\}, R = k - \widetilde{\sigma}_i(\Delta t''_j)$
  else
    $s_i = \{temp_i\}$
  $S = S \cup s_i$
Output: the final sub-template set S

6.3 Template Packing

As mentioned in *Challenge 2*, there are also many templates in the workload with very small sizes. We propose to 'pack' multiple such smaller templates into bins such that the total number of expected queries per bin is no greater than k. We formulate template packing as an integer linear programming problem:

$$
\begin{aligned}
\text{minimize} \quad & \#bins = \sum_j y_j \\
\text{subject to} \quad & \#bins \geq 1 \\
& \sum_{i \in bin_j} (\widetilde{\sigma}_i(\Delta t)) \leq k, \ \forall temp_i \in bin_j, \ \forall bin_j \\
& \#templates\_in\_bin_j \leq d, \ \forall bin_j \\
& \sum_j x_{ij} = 1, \forall temp_i \\
& y_j, x_{ij} \in \{0, 1\}, \forall temp_i, bin_j
\end{aligned}
$$

Where $y_j = 1$ if bin j is used and $x_{ij} = 1$ if template i is put into bin j. We modified a classical first-fit bin-packing algorithm [36] to solve this problem by preferring a bin with fewer templates when multiple bins can fit a template. This is necessary because an excess of templates in a bin can affect the accuracy of the per-bin model due to the complex mixture of various template patterns. To mitigate this, we quantitatively set the bin size, or the maximum number of templates per bin, to a constant d.

After packing, we can now create a model for each bin, which learns the distinct patterns for the templates present in the bin and predicts future queries for these templates. It is worth mentioning that, after packing, the bins are usually not fully occupied (i.e., with sizes smaller than k), an effective model for the bin will accurately predict queries for these templates in the next Δt interval, even with normal minor variations of template size in real workloads.

6.4 Per-Bin Models

We now adapt the per-template models into per-bin models to solve the next-Δt forecasting problem. We still use the Sibyl-LSTMs as the forecasting models, but have to make the following adaptations:

The feature map is more complex for per-bin model, since it includes queries from all templates within a bin. To handle multiple templates within each bin during featurization, it is necessary to include the template id as a feature and the feature map must also encompass the parameter values from all templates within the bin. We concatenate all the parameters from various templates in the feature map (e.g., two templates in a bin, with 3 and 2 parameters each, make a feature map of 5 parameters in the query feature vector).

The forecasting task becomes more challenging when modeling a mixed patterns from various templates in a bin. The per-bin model must accurately forecast template id and all parameter values within a bin for correctly reconstructing with the correct template and its parameter values.

6.5 Estimating Template Size

We now discuss how to estimate the template size used in the template cutting and packing. To do so, we train an additional one-layer LSTM. We use the finer $\Delta t''$ granularity (introduced earlier) to collect $\sigma_i(\Delta t''_1), \sigma_i(\Delta t''_2), ..., \sigma_i(\Delta t''_m)$ for each Δt interval in the historical workload, and train the LSTM model to forecast the future arrival rates $\hat{\sigma}_i(\Delta t''_1), \hat{\sigma}_i(\Delta t''_2), ..., \hat{\sigma}_i(\Delta t''_m)$. Summing up these predictions, we can compute $\hat{\sigma}_i(\Delta t)$. To ensure our next-Δt forecasting models can stably predict multiple successive Δt windows without costly retraining, we also set a longer forecasting horizon ΔT, e.g., ΔT = 1 week given Δt = 1 day. This provides a longer preview of future arrival rates, $\hat{\sigma}_i(\Delta t^1), \hat{\sigma}_i(\Delta t^2), ..., \hat{\sigma}_i(\Delta t^l)$. We then conservatively use the upper-bound of all forecasted $\hat{\sigma}_i(\Delta t^j)$ to approximate the template size in next Δt, i.e., $\widetilde{\sigma}_i(\Delta t) = \max(\hat{\sigma}_i(\Delta t^j)), \forall \Delta t^j \in \Delta T$. Similarly, we can approximate the template size for smaller interval $\Delta t''_x$ as $\widetilde{\sigma}_i(\Delta t''_x) = \max(\hat{\sigma}_i(\Delta t''^j_x)), \forall \Delta t''^j \in \Delta t$.

Note that this LSTM model is only used to estimate the size of a template for template cutting and packing. The forecasted arrival time from Sibyl-LSTMs still determines the actual number of queries in the next-Δt prediction. Note that while one-layer LSTM model is sufficient for our intended purpose, we can also use QueryBot 5000 [33] to estimate the template size.

7 FEEDBACK LOOP

Real workloads can shift, and new evolving patterns can emerge that the pre-trained models have never seen, which leads to an accuracy degradation. Sibyl offers a feedback loop to adapt to workload changes. Firstly, it tracks forecasting accuracy, detects changes in workloads, and automatically refines the models to enhance their performance. Secondly, it monitors both new and existing templates and keeps track of their sizes.

As Sibyl receives new queries continuously, it also receives the ground truth queries for the previous forecasting. This allows Sibyl to monitor the forecasting accuracy and decide whether to fine-tune the models or not. To identify the workload shifts that trigger the accuracy degradation, we set an accuracy threshold. The threshold can be decided by the lower bound of the forecasting accuracy expectation by applications or DBA. In our study, we fine-tune a model if the model accuracy is constantly lower than the threshold $\alpha = 75\%$ in a few forecasting rounds. During fine-tuning, as Sibyl collects new training data, new categorical values might emerge. We extend the dictionary of the parameter values by assigning new categorical values for them, and then use feature hashing to encode them.

In addition to detecting pattern changes in existing query templates, Sibyl also has the capability to continuously identify emerging templates (i.e., unseen templates that are recurrent in the new observation) as well as inactive templates (i.e., templates that have no queries showing up for a prolonged period of time). For a new template, we collect training data for it while continuously receiving new queries. Then we either fit the new template into an existing bin, if the bin capacity allows it, and fine-tune the existing per-bin model, or otherwise initialize a new bin for it and train a new model on the collected training data. For an inactive template, since the template has no queries showing up, during periodic fine-tuning on the new observed data, the model will automatically not forecast queries for the template anymore. Simultaneously, Sibyl keeps track of the size of each template and maintains models for the template size prediction (§6.5). If the total size of templates for a bin steadily exceeds the bin capacity, we divide the bin and re-adjust the assignment of templates into sub-bins so that the total size of templates in each sub-bin is no greater than k, and train new models for sub-bins from scratch.

8 EFFECTIVENESS MEASUREMENT

As discussed in §3, the majority of workload optimization tools that Sibyl targets at assume normal workloads instead of timed-workloads as input, so we only consider the normal workloads from the forecast when measuring prediction accuracy. Given the predicted workload $\hat{W}$ and the ground-truth future workload W, we use recall, precision, and F1 score as our evaluation metric with a customizable function $match(W, \hat{W})$ that defines how the ground-truth queries in W are matched with the queries in $\hat{W}$, and can be tailored to a specific application. More formally,

$$Recall = \frac{|match(W, \hat{W})|}{|W|}, \quad Precision = \frac{|match(W, \hat{W})|}{|\hat{W}|}$$

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

While one can use a strict matching where the predicated queries exactly match with the ground truth queries, such a matching is rarely needed. For a large number of applications, such as index tuning, view recommendation, partitioning of tables, and determination of the MDC of tables, a forecasted workload is required cover most of the ground-truth queries, and a *containment* based metric can be used for matching a ground-truth query with a predicted query. In Sibyl, we measure containment only using predicates in the queries. For equality predicate, the model gives a single value, hence we perform exact matching. For range predicates, a match is considered if the predicted range contains the ground-truth range. In the case of an IN clause, a match is considered if the predicted value is a superset of the ground-truth value. Given the forecasted workload $\hat{W}$ and the ground-truth workload W, we use the containment relationship to define a bipartite graph G, where each query q in W and each query $\hat{q}$ in $\hat{W}$ serve as the vertices, and an edge exists between q and $\hat{q}$, if q is contained by $\hat{q}$. Then we define the containment-based match $match_{\subseteq}(W, \hat{W})$ as the maximum bipartite matching of G. Practically, we use a popular textbook greedy bipartite matching algorithm to approximate the optimal result.

To better understand the effectiveness of forecasting in terms of containment, we also want to measure the degree of the containment relationship for each *matched* ground-truth and predicted query pair. We develop a new metric called *average containment-diff ratio* (*cnt-diff*) computed as follows. For a predicted range $\hat{R}$ and a ground-truth range R, the *cnt-diff ratio* is defined as $\frac{|\hat{R}-R|}{|R|}$, where $-$ is the range difference operation. In cases of a half-bounded range predicate, such as $col > a$, all the observed parameter values related to col are used to obtain the upper bound max_{col} and lower bound min_{col}, and the range (a, ∞) is changed to (a, max_{col}) before computing the cnt-diff ratio. Similarly, for IN-clause predicates, given a predicted set $\hat{S}$ and ground-truth set S, the cnt-diff ratio is defined as $\frac{|\hat{S}-S|}{|S|}$, where $-$ is the set difference operation. A good containment-based match should have a cnt-diff ratio close to zero. Finally, the average cnt-diff ratio is computed across all range predicates and IN-clause predicates for all the *matched queries* (we do not compute cnt-diff ratio for unmatched queries).

Finally, as discussed in §2, sometimes there exist a small percentage of unpredictable parameters in a workload. While the predicted values for these parameters will unlikely match the ground truth, they will still reflect the randomness of these parameters in the predicted workload. The workload optimization tools may be able to handle them some time. For example, in some view selections, a recurrent predicate with random parameter values will be ignored or converted into a group-by-column. For partitioning recommendation, random parameter values indicate either a non-ideal column for partitioning or a hash partitioning scheme for the column. An index recommender can take the randomness as a hint for needing an index structure better for point queries (e.g. hash-based indexes). Random parameters cannot be predicted. We believe the right approach is to be able to identify them and not apply any constraints on them. We do not want to unnecessarily penalize a forecasting method for these unpredictable parameters. As a result, we do not consider the unpredictable parameters when reporting the accuracy measurements in this paper.

9 EVALUATION

In this section, we study the parameter predictability (§9.1), evaluate the effectiveness and efficiency of Sibyl for both next-k (§9.2) and next-Δt forecasting problems (§9.3), discuss the effectiveness of fine-tunning (§9.4), and demonstrate the real benefit gained by Sibyl for the view recommendation and index selection applications (§9.5.1).

Model Alternatives. We compare with three alternative models: RF, vanilla LSTM, and a heuristic *history-based model*, which assumes a static workload and uses the last k queries or the queries in the last Δt window as the next-k or next-Δt forecast.

Implementation. Workload templatization was implemented in Java using Calcite parser (v1.32.0) [10]. We implemented the rest of Sibyl in Python (v3.10.8) and built the ML models with Scikit-learn (v1.1.3) [43] and TensorFlow/Keras framework (v2.11.0) [7]. For all ML models, we split each dataset such that the first 75% of the sequence is used for training and the last 25% for testing. For RF, we set the number of decision trees the same as the output window size k. For Vanilla LSTM and Sibyl-LSTMs, we set the number of cells in each LSTM layer the same as the output window size k and train with batches of 512 samples until convergence or reaching the maximum number of training epochs 20. We set the input window size equal to the output window size k during model training and testing. We initialize the learning rate as $1e^{-3}$ with decay rate 0.9, and use Adam optimization [30] and Huber loss [24] implemented by TensorFlow/Keras. In our experiments, we use the accuracy threshold $\alpha = 75\%$ to identify workload shift, and set the bin size $d = 50$ for the per-bin models, unless otherwise specified.

Experiment Setup. We used 6-core 206-GHz Xeon E5-2690V4 machines with Ubuntu 20.04 OS and one NVIDIA V100 GPU (16GB) for all experiments. Models for different templates or bins can be parallelized across machines to reduce the elapsed time. For inference, the models are indexed by their template or bin id, which allows loading only the required models at forecasting time.

9.1 Parameter Predictability

As discussed in §2, although most parameters in time-evolving queries are predictable, a few exhibit random behavior. Figure 8 details the prediction accuracy histograms for all the parameters in the Telemetry workload with the four different models in the next-1000

forecast. For each parameter, given the forecasted parameter sequence $\hat{p}_{n+1}, \hat{p}_{n+2}, ..., \hat{p}_{n+k}$ and the ground truth $p_{n+1}, p_{n+2}, ..., p_{n+k}$, the parameter accuracy is defined as $\frac{|\hat{p}_{n+i}=p_{n+i}|}{k}, \forall i \in [1, k]$. We can observe that more parameters can be accurately predicted using the ML-based models than the history-based model. Our Sibyl-LSTMs performs the best. In Figure 9, we present a visualization of the actual versus forecasted values for the three predictable parameters (DATE, STRING, and ID-related fields, respectively) for one of the biggest templates in the Telemetry workload, as an example. We observed that the forecasted results closely aligned with the ground truth, particularly for periodic and trend patterns.

Figure 8 shows that there are some unpredictable parameters for which even ML-based models cannot achieve over 75% prediction accuracy. We visualized the time-series patterns for these parameters and observed the similar random patterns as shown in Figure 2(d). For the remaining experiments, we ignore the unpredictable parameters as discussed in §8, and report accuracy using the containment-based match definition (also introduced in §8).

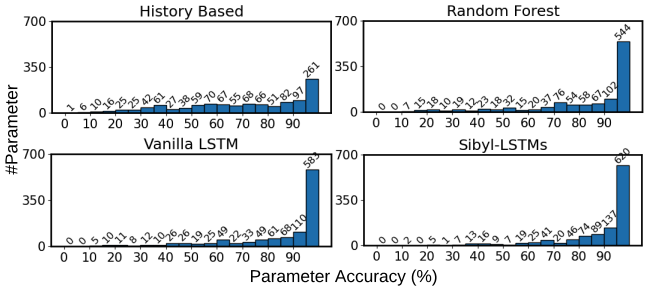

Figure 8: Histogram of prediction accuracy for parameters in the Telemetry workload, with forecasting window $k = 1000$.

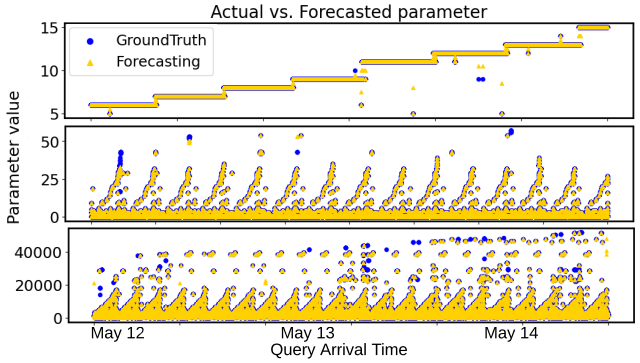

Figure 9: Forecasting visualization.

9.2 Next-k Forecasting

We first show the forecasting accuracy of the per-template models for the next-k forecasting problem. We report the results on the time-evolving templates, ignoring the templates where the parameter values do not change over time.

9.2.1 ***Comparison of Models.*** Table 6 shows the average recall results for the per-template models in each workload with different window sizes k. We note that the Sales workload has relatively a smaller number of queries, leading to smaller k values. This is necessary to ensure sufficient training samples when sliding the input window over the query traces. The results show that Sibyl-LSTMs clearly outperforms all the other models. The history-based approach has very low accuracy, especially for the Telemetry, SCOPE and BusTracker workloads. Vanilla LSTM generally works better than the RF model. To ensure that the predicted queries do not overshoot for range and IN-clause predicates in the containment-based matches, we report the cnt-diff (see §8) for the forecast in Table 7. Note that cnt-diff is calculated only on correct predictions. The results show that Sibyl-LSTMs often matches or surpasses cnt-diff ratios of other ML-based approaches, indicating that Sibyl-LSTMs does not attain the superior model accuracy by over-prediction.

9.2.2 ***The Effect of*** k. The selection of k depends on computational and memory resources available. We set the max forecasting window size as 1000 to avoid the out-of-memory error given the machine memory constraint. As shown in Table 6, the selection of k also affects model accuracy. A smaller k can result in a more accurate forecast but may also increase the risk of model instability or over-fitting. A larger k predicts for a large forecasting window at once, but it poses challenges on accuracy and model scalability. The experimental results show that Sibyl-LSTMs has better model scalability – when scaling up the prediction window size, the variance of the accuracy is smaller compared to the other baselines.

Table 6: Accuracy results (%) for next-k forecasting problem.

	Telemetry			SCOPE			BusTracker			Sales		
k	100	500	1000	100	500	1000	100	500	1000	100	200	500
History-based	27.4	17.0	31.8	7.0	13.4	32.8	12.8	11.2	7.9	47.8	64.9	72.8
Random Forest	85.4	82.3	80.5	83.7	83.2	82.6	91.4	90.2	88.6	79.6	75.3	71.2
Vanilla LSTM	91.0	90.3	90.1	89.3	88.7	88.2	92.0	92.3	91.8	84.9	85.3	80.7
Sibyl-LSTMs	**95.8**	**96.7**	**95.4**	**94.6**	**95.4**	**94.7**	**96.0**	**96.2**	**95.8**	**92.4**	**91.7**	**88.2**

Table 7: Cnt-diff ratio (%) for the next-k forecasting problem.

	Telemetry			SCOPE			BusTracker			Sales		
k	100	500	1000	100	500	1000	100	500	1000	100	200	500
History-based	3.50	4.24	1.01	1.14	1.19	0.73	1.88	2.86	3.04	0.50	0.49	0.55
Random Forest	0.15	0.04	0.16	0.08	0.08	0.12	0.22	0.11	0.22	0.25	0.09	0.28
Vanilla LSTM	0.32	0.11	0.21	0.11	0.18	0.07	0.31	0.31	0.35	0.11	0.19	0.04
Sibyl-LSTMs	0.19	0.22	0.16	0.16	0.13	0.10	0.25	0.16	0.19	0.36	0.26	0.14

9.3 Next-Δt Forecasting

In Figure 10, we present the evaluation results of per-bin models for the next-Δt problem, varying Δt to 1 hour, 6 hours, 12 hours, and 1 day. These time intervals are typical in our targeted application [27, 28]. We set k (in template cutting and packing algorithms) to 1000 for the Telemetry, SCOPE, and BusTracker workloads, and 500 for Sales, as it has fewer queries.

9.3.1 ***Comparison of Models.*** Sibyl surpasses other forecasting models and maintains stable accuracy across various Δt settings. Vanilla LSTM and Random Forecast perform poorly on the Sales, which has more outliers and more unstable patterns. For Telemetry, the history-based method performs well with the 12-hour interval

due to the workload's recurrent queries that have the same parameter values within a day (between the past 12-hour window and the future 12-hour window). But this method is ineffective with the one-day interval, as many query parameter values change when crossing the day boundary. The history-based method yields unsatisfactory results for the other three workloads that exhibit more rapid and intricate evolution and involve time-related parameters that operate on a finer time scale. Therefore, it is imperative to use an ML-based forecasting model to handle the evolving workload.

Other ML alternatives: As noted earlier, our primary contribution lies in leveraging an effective ML model for our complex task, instead of developing new or exhaustively testing possible ML algorithms. Here, we consider two more common alternatives although additional models are possible. (1) In another adaption of RF, referred to as RF+, we concatenate all values from q_{n-k+1} to q_n as a single input vector and output all parameters of q_{n+1} to q_{n+k} together. As Table 8 shows, RF+ achieves similar low accuracy as the baseline RF in Figure 10. (2) We also consider a Transformer based decoder-only model (TRF), with a 12-layer multi-head attention architecture. Table 8 shows TRF attains a slightly better accuracy than Sibyl-LSTMs, but at the cost of 18.2× more training time and 4.4× more model storage overhead. We chose Sibyl-LSTMs for Sibyl, because it offers a trade-off between accuracy and efficiency.

Table 8: The recall results (%) comparison with other ML alternatives for the next-Δt forecasting on the Telemetry workload.

Δt	1 hour	6 hour	12 hour	1 day
RF+	75.5	75.9	75.8	77.5
TRF	92.6	92.5	93.0	92.1
Sibyl	90.8	92.4	91.1	91.3

9.3.2 ***The Effect of*** Δt. The accuracy of next-Δt forecasting results is influenced not only by the model's ability to accurately forecast queries but also by its ability to accurately forecast arrival times. As the forecasting time window increases, the accuracy of the results changes. Typically, time-series forecasting with shorter horizons is easier and more accurate. However, predictions for smaller time granularity, such as 1 hour instead of 1 day, tend to be noisier and subject to greater fluctuations in query arrival rates per hour than per day, which makes forecasting arrival hours more challenging than forecasting arrival days. As a result, the majority of forecasting accuracy results with a one-day time window are higher than other time-window settings.

9.3.3 ***Time and Storage Efficiency***. Table 9 demonstrates the time and storage savings achieved through the implementation of template cutting and packing as well as the per-bin models for next-day forecasting. Note that the training times reported in the table are aggregation across all templates/bins. By parallelizing on multiple machines, the elapsed training times can be significantly shortened. We note that the average time and storage overhead of a single per-bin model is higher than that of a per-template model due to larger model capacity and a higher average number of queries per bin than per template. However, template cutting and packing significantly reduce the number of models by up to 23×. Moreover, employing per-bin models results in a significant reduction in training time of up to 13.6× and storage space by up to 6.0× when compared to per-template models.

We note that there is a trade-off between efficiency and accuracy. Comparing the accuracy results in Table 6 and Figure 10, the accuracy of per-bin models is slightly lower than per-template models. Because the next-Δt forecasting is a harder problem to solve than the next-k forecasting, as mentioned in §6.4. It has a higher requirement for the per-bin models to forecast the query arrival time precisely, depending on which we can identify the queries in the next time interval correctly. Different from the per-template models, the per-bin models are required to forecast template ids in a bin and more queries in per bin than per template.

Compared to the training overhead, the total per-bin model prediction times on GPU are negligible: 3.9s, 241s, 1.6s, and 0.031s for Telemetry, SCOPE, Bustracker, and Sales, respectively.

Table 9: The aggregate training time and model storage overhead for per-template and per-bin models. ↓ means the reduction ratio.

	Telemetry	SCOPE	BusTracker	Sales
# per-template models	2157	168197	258	23
aggregate training time† (h)	119.8	9344.3	14.3	1.3
total model storage† (GB)	54.0	4205.0	6.5	0.6
# per-bin models (↓)	124 (17.4×)	7716 (21.8×)	50 (5.2×)	1 (23×)
aggregate training time† (h) (↓)	11.0 (10.9×)	685.9 (13.6×)	4.4 (3.3×)	0.1 (13.0×)
total model storage† (GB) (↓)	12.4 (4.4×)	771.6 (5.4×)	5.0 (1.3×)	0.1 (6.0×)

† The one-epoch average training time is 10s for a per-template model and 16s for a per-bin model. The average storage overhead is 25MB for a per-template model and 100MB for a per-bin model.

9.3.4 ***Comparison with Previous Work***. We now compare Sibyl with QueryBot5000 [33], TEALED [23] [3] and Q-Learning [38] on the common BusTracker workload. Although none of the three methods were originally designed for future query forecasting, we adapted them to solve the next-Δt forecasting problem. QueryBot5000 and TEALED forecast only the arrival rates for templates in the next Δt. To generate queries, we take the most recent n historical queries from each template, where n is the predicted query rate by QueryBot5000 and TEALED. Q-Learning only forecasts the next *one* query. We adapt it by continuously forecasting using the last predicted query as input until collecting m queries, where m is the number of queries in the last Δt. We call the adapted methods as QueryBot5000+, TEALED+ and Q-Learning+, respectively.

We compare the recall results and prediction overhead for next-1 day forecasting in Table 10. Sibyl significantly outperforms QueryBot5000+ as the latter fails to forecast the time-evolving parameter values. QueryBot5000+ and TEALED+ show slightly better accuracy than the history-based method shown in Figure 10, as they accurately forecast the query arrival rate. Q-Learning+ only suggests a similar query to the ground truth and has very low recall results when applying the over-and-over prediction. Q-Learning+ also has the highest prediction overhead because it forecasts only one query at a time, while other methods have comparable low prediction overhead.

[3] It is our re-implementation because the code/executable of TEALED is not available.

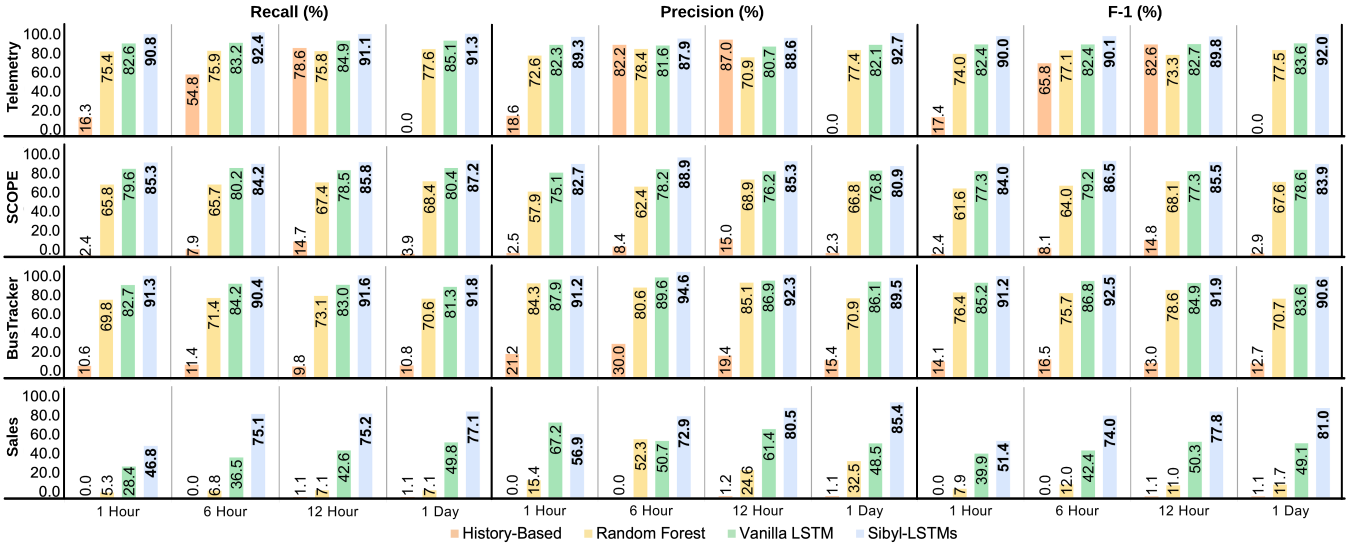

Figure 10: Accuracy results for the next-Δt forecasting problem.

Table 10: The recall results (%) and prediction overhead comparison for the next-Δt forecasting on BusTracker.

Δt	1 hour	6 hour	12 hour	1 day	prediction overhead (1 day)
QueryBot5000+	12.4	11.7	12.0	13.9	1.1s
TEALED+	11.0	11.5	10.6	12.0	5.2s
Q-Learning+	0.0	0.0	0.0	0.0	6251.4s
Sibyl	91.3	90.4	91.6	91.8	1.6s

9.3.5 ***The Effect of Bin Size.*** We now assess the impact of bin size d (the maximum number of templates per bin) on the recall results of next-1 day forecasting on the Telemetry workload, in Table 11. While per-template models (d=1) provide high accuracy, they are not time and storage efficient as discussed in § 9.3.3. Larger d values reduce the number of models needed but compromise accuracy due to the complex mixture of patterns. We empirically set $d = 50$ in Sibyl to balance accuracy and efficiency.

Table 11: The recall results (%) of different d settings for the next-1 day forecasting on the Telemetry workload.

d	1	10	30	50	70	100
#bins	2157	442	217	124	120	119
Recall	93.2	92.0	91.6	91.3	91.1	91.1

9.4 Effectiveness of Fine-tuning Models

In this experiment, we demonstrate the effectiveness of Sibyl's feedback loop (see §7). Figure 11 displays the detection of workload shift in the Telemetry workload and the execution of a per-bin model fine-tuning for the next-day forecasting. Figure 11(a) depicts a pattern change of a parameter in the Telemetry workload starting from May 13 (highlighted in light blue), which Sibyl detects by observing the decline in accuracy. The model accuracy on the shifted pattern is 51.9%, which falls below the threshold $\alpha = 75\%$, triggering model fine-tuning. In Figure 11(b), we observe that Sibyl fine-tunes the Sibyl-LSTMs by incrementally training on newly observed data, rather than training from scratch. The average fine-tuning time per epoch is under 10s for a per-template model and 16s for a per-bin model, due to the smaller size of new observed data. In Sibyl, we limit the maximum number of fine-tuning epochs to 20. As shown in Figure 11, the model converges in just two epochs with 6.4 seconds of overhead, improving accuracy to 95.0%, which is close to the pre-trained accuracy of 95.4%.

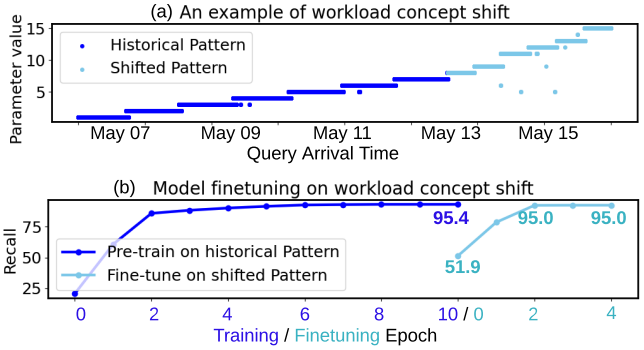

Figure 11: Fine-tuning on Telemetry workload shift.

9.5 Applications of Workload Forecasting

We now show how workload forecast can be applied to two classical workload-based optimization applications: view selection and index selection. The purpose of our experiments is to show how an existing view/index selection algorithm can directly use the forecasted workload without modification to its algorithm to produce better views/indexes, rather than introducing a new view/index selection algorithm. Thus, we employ the well-known view selection algorithm [9] and the PostgreSQL index recommender tool [1].

9.5.1 ***Application to View Selection.*** We train Sibyl-LSTMs, QueryBot5000+, and Q-Learning+ using 2237 Sales queries over 20 consecutive days. Then, we employ the view selection algorithm to create materialized views for the subsequent day. As the baseline, we conduct view recommendation on the preceding 7 days of *history* queries. For Sibyl-LSTMs, QueryBot5000+, and Q-Learning+, we use the predicted queries to recommend views. Then we run the ground-truth queries using the recommended views on a cloud-based data warehouse (2-compute nodes and 385GB data). Figure 12 shows the query execution speedup achieved by using the views compared to without the views. The queries forecasted by Q-Learning+ do not lead to any useful views at all, so there is no speedup. Views based on both History and QueryBot5000+ result in merely 1.06× improvement, whereas views recommended based on Sibyl leads to 1.83× speedup, roughly a 1.7× difference.

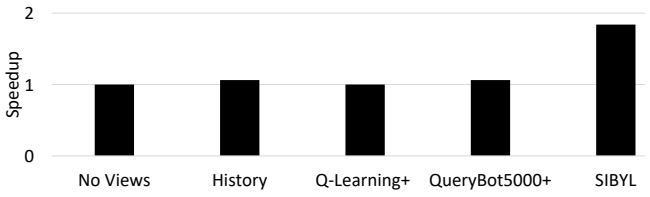

Figure 12: Speedup via views on Sales workload.

9.5.2 ***Application to Index Selection.*** In this experiment, we train Sibyl-LSTMs, QueryBot5000+, and Q-Learning+ using 741K queries from 11 days of Telemetry workload. We then run the index recommender for Day 12. Due to the large volumes of queries (≈ 47K) on the 12th day, we only focus on the 151 queries that fall in the 3AM - 4AM window. For the baseline, we run the index recommender on a random sample of 1K queries from the historical workload, following the same approach in [33]. For Sibyl-LSTMs, QueryBot5000+, and Q-Learning+, we recommend indexes on the forecasted queries. Then, we execute the ground-truth queries on the recommended indexes using a single-node PostgreSQL server on 24GB data. Figure 13 shows the speedup achieved by various methods. *History* exhibits a modest 1.2× speedup when compared to *No Index*. All three ML-based methods outperform *History*. Not surprisingly, QueryBot5000+ achieves a very good 1.72× speedup, since index recommendation is one of the target applications it is designed for [33]. Sibyl achieves a comparable 1.67× speedup. It is important to note that the accuracy requirements for query prediction are less stringent for index recommendation. As long as the predicted queries encompass the main tables and columns, the recommended index will be beneficial for future workloads. In contrast, view recommendation necessitates precise prediction of query templates and parameter values to generate useful views.

10 RELATED WORK

There has been extensive research on workload modeling and forecasting for relational databases, which can be classified into three main categories: (*i*) partial query forecasting, (*ii*) workload feature forecasting, and (*iii*) suggesting queries from the past.

Partial Query Forecasting. [25] learns vector representations for SQL statements and query plans, which captures the syntax

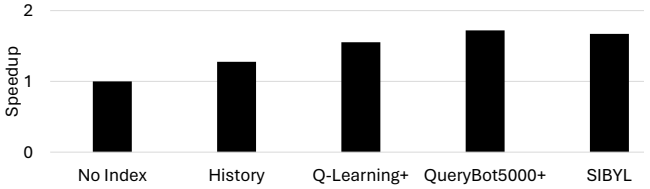

Figure 13: Speedup via indexes on Telemetry workload.

similarity among query statements but fails to predict the literals in queries. [38] leverage RNNs [37] and Q-Learning [52] to predict the next query based on the current query, with the forecasted literals as a bin of values rather than accurate values. While these works partially forecast query statements, Sibyl accurately predicts the entire future statements.

Forecasting Workload Features. [22] utilizes Markov models to predict the shifts in the workload over time, and [21] models periodic patterns of a workload by classification. [50] forecasts accessing frequency by ensemble learning and [33] predicts the arrival rate of future queries by hybrid-ensemble learning to suggest indexing. [8] predicts data access characteristics such as latency, when data will be accessed, and volume of data accessed. The paper [23] utilizes a combination of time-sensitive empirical mode decomposition (EMD) and auto LSTM encoder-decoder to forecast resource utilization and query arrival rates for DBMSs. These efforts focus on specific aspects of workload forecasting. In contrast, Sibyl is a comprehensive workload forecasting framework that predicts query statements, arrival time, arrival rate, and pattern shifts simultaneously.

Suggesting Queries From the Past. Query recommendation [17, 18] selects queries from the historical logs that overlap with ongoing interaction sessions using collaborative filtering. Query autocompletion [29] helps users complete the missing parts of a query by choosing transitions based on heuristics, such as the popularity of query fragment co-occurrence in prior logs. The paper [19] posits that database workloads are influenced by real-world events. It forecasts future workloads by identifying upcoming events and matching them with similar past events. These approaches are not ML-based, but only rely on heuristics and historical workloads. They also do not capture query evolution, thus fail to suggest new queries that are not already in the history.

11 CONCLUSION

We introduced Sibyl, an ML-based workload forecasting framework that predicts future queries across various time intervals. Unlike the prior work, Sibyl formulates the forecasting problem as a multivariate, multi-step, sequence-to-sequence prediction problem. We addressed several challenges to efficiently and accurately predict future queries, which lead to performance improvement in applications such as views and indexes selection, emphasizing Sibyl's potential in database optimization. As future work, we plan to explore other ML techniques to further reduce the training overhead and improve efficiency.

REFERENCES

[1] [n. d.]. Dexter. https://github.com/ankane/dexter.
[2] [n. d.]. IBM Db2. https://www.ibm.com/analytics/us/en/db2.
[3] [n. d.]. IBM Informix. https://www.ibm.com/products/informix.
[4] [n. d.]. Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/sql-server-2022.
[5] [n. d.]. Oracle. https://www.oracle.com/database.
[6] [n. d.]. SQL Server - Parameter Markers. https://learn.microsoft.com/sql/odbc/reference/appendixes/parameter-markers.
[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *Osdi*, Vol. 16. Savannah, GA, USA, 265–283.
[8] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Tiresias: enabling predictive autonomous storage and indexing. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3126–3136.
[9] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Morgan Kaufmann, 496–505.
[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 221–230.
[11] Nicolas Bruno, Surajit Chaudhuri, Arnd Christian König, Vivek R. Narasayya, Ravishankar Ramamurthy, and Manoj Syamala. 2011. AutoAdmin Project at Microsoft Research: Lessons Learned. *IEEE Data Eng. Bull.* 34, 4 (2011), 12–19.
[12] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *VLDB '07*. 3–14.
[13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).
[14] Dineshen Chuckravanen. [n. d.]. Approximate entropy as a measure of cognitive fatigue: an eeg pilot study. ([n. d.]).
[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423
[16] Edgar Haren. 2017. Oracle Revolutionizes Cloud with the World's First Self-Driving Database. https://blogs.oracle.com/database/post/oracle-revolutionizes-cloud-with-the-worlds-first-self-driving-database.
[17] Magdalini Eirinaki, Suju Abraham, Neoklis Polyzotis, and Naushin Shaikh. 2013. Querie: Collaborative database exploration. *IEEE Transactions on knowledge and data engineering* 26, 7 (2013), 1778–1790.
[18] Magdalini Eirinaki and Sweta Patel. 2015. QueRIE reloaded: Using matrix factorization to improve database query recommendations. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 1500–1508.
[19] Janusz R. Getta. 2018. Event Based Forecasting of Database Workloads. In *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. 1767–1773.
[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[21] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards workload-aware self-management: Predicting significant workload shifts. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 111–116.
[22] Marc Holze and Norbert Ritter. 2008. Autonomic databases: Detection of workload shifts with n-gram-models. In *Advances in Databases and Information Systems: 12th East European Conference, ADBIS 2008, Pori, Finland, September 5-9, 2008. Proceedings 12*. Springer, 127–142.
[23] Xiuqi Huang, Yunlong Cheng, Xiaofeng Gao, and Guihai Chen. 2022. TEALED: A Multi-Step Workload Forecasting Approach Using Time-Sensitive EMD and Auto LSTM Encoder-Decoder. In *Database Systems for Advanced Applications*. 706–713.
[24] Peter J Huber. 1992. Robust estimation of a location parameter. *Breakthroughs in statistics: Methodology and distribution* (1992), 492–518.
[25] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. 2018. Query2vec: An evaluation of NLP techniques for generalized workload analytics. *arXiv preprint arXiv:1801.05613* (2018).
[26] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *VLDB* 11, 7 (2018), 800–812.
[27] Alekh Jindal, Shi Qiao, Hiren Patel, Abhishek Roy, Jyoti Leeka, and Brandon Haynes. 2021. Production Experiences from Computation Reuse at Microsoft.. In *EDBT*. 623–634.
[28] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*. 191–203.
[29] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. 2010. SnipSuggest: Context-aware autocompletion for SQL. *Proceedings of the VLDB Endowment* 4, 1 (2010), 22–33.
[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[31] Xiaoling Li, Ying Jiang, Jun Hong, Yuanzhe Dong, and Lei Yao. 2016. Estimation of cognitive workload by approximate entropy of EEG. *Journal of Mechanics in Medicine and Biology* 16, 06 (2016), 1650077.
[32] Liang Lu, Xingxing Zhang, and Steve Renais. 2016. On training the recurrent neural network encoder-decoder for large vocabulary end-to-end speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 5060–5064.
[33] 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 the 2018 International Conference on Management of Data*. 631–645.
[34] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. *ACM SIGMOD Record* 51, 1 (2022), 6–13.
[35] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).
[36] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.
[37] Larry R Medsker and LC Jain. 2001. Recurrent neural networks. *Design and Applications* 5 (2001), 64–67.
[38] Venkata Vamsikrishna Meduri, Kanchan Chowdhury, and Mohamed Sarwat. 2021. Evaluation of machine learning algorithms in predicting the next SQL query from the future. *ACM Transactions on Database Systems (TODS)* 46, 1 (2021), 1–46.
[39] A.V. Oppenheim. 1999. *Discrete-Time Signal Processing*. Pearson Education.
[40] Oracle. 2006. *Oracle Database 10g Release 2: The Self-Managing Database*. Technical Report. Oracle.
[41] Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Tim Malkemus, Leslie Cranston, and Matthew Huras. 2003. Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. In *SIGMOD '03*. 637–641.
[42] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.
[43] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, null (nov 2011), 2825–2830.
[44] Steven M Pincus. 1991. Approximate entropy as a measure of system complexity. *Proceedings of the National Academy of Sciences* 88, 6 (1991), 2297–2301.
[45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[46] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 4 (2018), e1249.
[47] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD*. 99–113.
[48] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD '18* (Houston, TX, USA). 205–219.
[49] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2020. CrocodileDB in Action: Resource-Efficient Query Execution by Exploiting Time Slackness. *Proc. VLDB Endow.* 13, 12 (aug 2020), 2937–2940.
[50] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.
[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[52] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.
[53] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature Hashing for Large Scale Multitask Learning. In *ICML*

'09. 1113–1120.

[54] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *PVLDB* 12, 3 (nov 2018), 210–222.

[55] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic view generation with deep learning and reinforcement learning. In *ICDE*. 1501–1512.

[56] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB '04*. 1087–1097.