Evaluation of Machine Learning Algorithms in Predicting the Next SQL Query from the Future

VENKATA VAMSIKRISHNA MEDURI, KANCHAN CHOWDHURY, and MOHAMED SARWAT, Arizona State University, USA

Prediction of the next SQL query from the user, given her sequence of queries until the current timestep, during an ongoing interaction session of the user with the database, can help in speculative query processing and increased interactivity. While existing machine learning– (ML) based approaches use recommender systems to suggest relevant queries to a user, there has been no exhaustive study on applying temporal predictors to predict the next user issued query.

In this work, we experimentally compare ML algorithms in predicting the immediate next future query in an interaction workload, given the current user query or the sequence of queries in a user session thus far. As a part of this, we propose the adaptation of two powerful temporal predictors: (a) Recurrent Neural Networks (RNNs) and (b) a Reinforcement Learning approach called Q-Learning that uses Markov Decision Processes. We represent each query as a comprehensive set of fragment embeddings that not only captures the SQL operators, attributes, and relations but also the arithmetic comparison operators and constants that occur in the query. Our experiments on two real-world datasets show the effectiveness of temporal predictors against the baseline recommender systems in predicting the structural fragments in a query w.r.t. both quality and time. Besides showing that RNNs can be used to synthesize novel queries, we find that exact Q-Learning outperforms RNNs despite predicting the next query entirely from the historical query logs.

CCS Concepts: • **Information systems** → **Relational database model**; **Temporal data**; • **Computing methodologies** → **Q-learning**; **Feature selection**; **Supervised learning**; **Reinforcement learning**; **Neural networks**;

Additional Key Words and Phrases: Query prediction, schema-aware SQL embeddings, recurrent neural networks, recommender systems

ACM Reference format:
Venkata Vamsikrishna Meduri, Kanchan Chowdhury, and Mohamed Sarwat. 2021. Evaluation of Machine Learning Algorithms in Predicting the Next SQL Query from the Future. *ACM Trans. Database Syst.* 46, 1, Article 4 (March 2021), 46 pages.
https://doi.org/10.1145/3442338

1 INTRODUCTION

Given a sequence of SQL queries $(qu_\tau)_{\tau=1}^{i} = (qu_1, qu_2, \ldots, qu_i)$ from a user U in a querying session upon a relational database until the current timestep i, we define qu_{i+1} as the future SQL query

This work was supported by the National Science Foundation (NSF) under Grant 1845789.
Authors' addresses: V. V. Meduri, K. Chowdhury, M. Sarwat, Arizona State University, 699 South Mill Avenue, CIDSE, Brickyard Engineering, Tempe, AZ 85281; emails: {vmeduri, kchowdh1, msarwat}@asu.edu.

0362-5915/2021/03-ART4 $15.00
https://doi.org/10.1145/3442338

issued by the user at the immediate next timestep $i + 1$. Under the scenario that there are several such active user sessions concurrently posing queries to the database, what would be the ideal machine learning (ML) algorithm that can predict the next user query for each of these ongoing sessions accurately? To answer this question, we build a *next query predictor* that we assume has access to the historical query logs of all the interaction sessions held thus far against the database. To predict the next query, we exploit the *user think time* [20], which is defined as the time delay between the issuance of two successive queries. The end user who issues the queries can either be a human or a transactional/analytical application running in the background. During this user think time, an ML model can predict the next SQL query and can also be refined periodically.

Prediction of the next SQL query can lead to several benefits ranging from adaptive indexing to speculative query optimization and processing to database tuning. If we can predict the next SQL query in its entirety during the user think time, then we can speculatively execute it and prefetch its results into the memory. When the actual next query arrives, we can skip its execution and return the prefetched results directly from the memory. Even if an SQL query cannot be predicted in its entirety, knowing apriori the impending query operators and their associated schema elements (defined as $query\ fragments$ [13]) from the successive query is useful. If we know the set of tables and attributes participating in projection/selection/join/aggregate predicates, then it can allow for partial or full query plan construction during the user think time. The database can also evaluate it partially or entirely depending on the structure of the SQL query. Predicting sort (ORDER BY) and aggregate (COUNT) operator fragments can help in pre-allocation of buffer pages and configuring # CPU-bound threads to parallelize their execution. Likewise, being cognizant of the selection (or join) predicates can help us pre-load or create indexes if we know the possible range of constants in the selection predicates. The goal of this work is to compare the state-of-the-art ML algorithms and find the best approach suitable for query fragment prediction. To evaluate if the predicted query fragments have tangible benefits for database systems, we choose speculative query execution and query result prefetching as the target application. Although building a caching middleware is beyond the scope of this article, we re-generate the entire SQL query from the predicted fragments and execute it to evaluate the results of the predicted SQL query against the actual SQL query that occurs at the next timestep. The other aforementioned benefits from query prediction such as database tuning, buffer management, speculative query optimization, and adaptive indexing can be realized by building downstream applications as a part of possible future work that can utilize the best performing ML approach found from our experimental evaluation in this article.

Prior work relevant to next query prediction can be classified into four broad categories: user intent prediction for interactive data exploration, query recommendation and autocompletion, latency reduction for data exploration, and usage of ML for other relevant problems such as cardinality estimation and workload generation. More details about these categories can be found in Section 2. Among them, the closest family of learning-based techniques that can be applied to query prediction includes query recommenders. Query recommendation using session similarity-based collaborative filtering [13] suggests the most relevant queries from other prior/ongoing interaction sessions to the current user session in progress. Relevance or session similarity is computed based on the fraction of overlapping SQL query fragments across the sessions. Likewise, a matrix factorization-based collaborative filtering approach [14] was also proposed for query recommendation. However, both these approaches sample historical queries to alleviate the computational cost of fragment similarity. Active learning-based approaches [11, 12] use binary classifier variants of SVMs and decision trees to distinguish tuples of eventual interest to the user from uninteresting tuples in the underlying database. The purpose of this line of work is to discover the tuples that answer the goal (last) query from a session in as few iterations (human-database interactions) as possible. A fundamental limitation in adapting this approach to query prediction is that the tuples of eventual interest only correspond to the last query in a query session. This disallows its applicability to predict the next query, because there is only one goal query per session that is static and cannot change at each timestep. Other limitations include the need to train a separate classifier for each user session that affects scalability with growing #sessions.

Contrary to the existing approaches [11, 12] that predict static (eventual) user intent, we capture the dynamically changing user intent by predicting the next query that brings several benefits as mentioned earlier. Predicting query fragments is more scalable than tuple classification, which is why we formulate each query as its constituent fragments of co-occurring schema elements and SQL operators. Since classifiers such as Support Vector Machines (SVMs) or decision trees are inapplicable in this context, predicting the query fragments at each timestep requires powerful temporal predictors. Thus we compare the performance of Recurrent Neural Networks (RNNs) and Q-Learning against two recommender system baselines: session-similarity-based Collaborative Filtering [13] and Singular Value Decomposition– (SVD) based matrix factorization [14]. We propose two adaptations of RNNs—one that predicts the next query from the pool of historical interaction sessions and another that can synthesize a completely novel query based on the learning it has undergone. Likewise, we also propose two formulations of reward functions for exact Q-Learning—a "Boolean" reward function that reinforces predicted queries that entirely match the expected queries and a relaxed "Numeric" reward function that learns sequences of predicted queries that have a high percentage of overlapping fragments with those from the expected queries. Eirinaki et al. [13] term the co-occurrences of SQL operators and their associated schema elements as query fragments but confine them to projected attributes, relations, selection and join predicates. We extend the definition of fragments to also capture the possible ranges of selection predicate constants of all data types, while supporting more Data Manipulation Language (DML) types such as INSERT, UPDATE, and DELETE besides Select, Project, Join queries. The advantage in our extended fragment definition is that it represents an SQL query more exhaustively as compared to Reference [13], and this specifically helps in re-generating the SQL query in as much of a lossless manner as possible from the fragments. This allows us to use our extended embedding fragments as generic feature vectors that represent an SQL query as a whole. In this work, we evaluate the performance of various ML approaches in predicting the fragments/syntactic features in the next SQL query to be issued by an end user or an application, given the sequence of queries issued thus far in an interaction session. Besides that, we also propose heuristics to re-generate the SQL query from the predicted fragments and execute the re-generated SQL query to compare it against the actual next query in terms of the result tuples as well.

We use one-hot encoding to represent the SQL queries as bit vector embeddings (that record the presence or absence of SQL fragments within each query) upon which ML models can be trained and tested. This is in contrast to existing embedding techniques such as Word2Vec [33] or GloVE [41] that create floating point latent dimensions from raw natural language text. We do not use these libraries, because (a) they are agnostic to the SQL structure and the semantic importance attached to each distinct SQL-specific keyword in a query and (b) the low-dimensional latent embedding cannot be easily reverse engineered into an SQL query and hence evaluating the predicted embeddings w.r.t. SQL fragments is not straightforward. An important property of our proposed fragment embedding is that the SQL operators, schema elements, comparison operators, and constant range bins can be easily reconstructed from the predicted embedding vector.

Figure 1 illustrates the end-to-end functionality of our benchmark system. Given an SQL query qu_i issued at timestep i, we derive a fragment embedding for qu_i as a bit vector. We feed the embedding of qu_i to the next query predictor that predicts the fragments in the next query qu_{i+1}, anticipated to be issued by the end user (application) at timestep i+1. Simultaneously, qu_i is executed

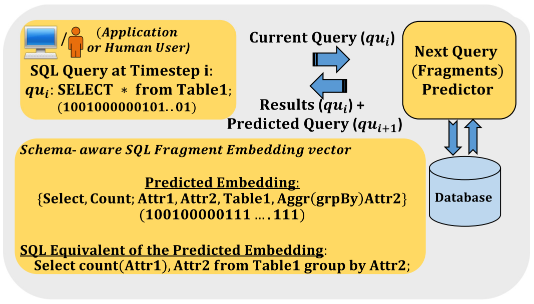

(Application or Human User)
SQL Query at Timestep i: qu_i: SELECT * from Table1; (1001000000101..01)
Current Query (qu_i)
Results (qu_i) + Predicted Query (qu_{i+1})
Next Query (Fragments) Predictor
Database
Schema- aware SQL Fragment Embedding vector
Predicted Embedding: {Select, Count; Attr1, Attr2, Table1, Aggr(grpBy)Attr2} (100100000111 .... 111)
SQL Equivalent of the Predicted Embedding: Select count(Attr1), Attr2 from Table1 group by Attr2;

Fig. 1. Predicting fragments in the next query.

and its result tuples are returned to the querying application along with the predicted embedding of qu_{i+1}. In the example shown in Figure 1, it is interesting to note that the predicted fragment embedding is self-sufficient in completely reconstructing the SQL query as it is devoid of WHERE clauses or selection predicates. We predict a selection predicate as {ATTR, OP, CONST-BIN} where ATTR, OP, and CONST-BIN denote the schema attribute, the comparison operator, and a predefined constant value range bin for ATTR, respectively. We compare the temporal predictors, i.e., RNNs and Q-Learning against recommender system baselines from the literature w.r.t. prediction quality and response time. Our evaluation is presented upon two real-world datasets in both offline and online training settings described in Section 3.3. Following is a summary of our contributions.

- We build a next query prediction framework that compares several state-of-the-art ML algorithms in predicting the fragments from the immediate next SQL query in an interaction session upon various evaluation metrics such as the prediction quality, response time and the memory consumption of the algorithms.
- Instead of using external libraries such as Word2Vec [33] or GloVe [41] that create SQL-agnostic word embeddings, we use a schema-aware SQL fragment embedding mechanism based on one-hot encoding that effectively captures all possible combinations of database operators and schema elements that can occur in SQL queries along with the comparison operators and constant bin ranges in the selection predicates of these queries. The same embedding scheme is used to generate feature vectors uniformly for all the ML algorithms.
- We propose the usage of an incrementally trained RNN for query fragment prediction. Our adaptation of RNNs has a unique ability to synthesize completely novel combinations of fragments into a meaningful SQL query, besides recommending historical queries from prior interaction sessions.
- We propose the usage of a reinforcement learning algorithm called exact Q-Learning for query fragment prediction. We model the agent, environment, reward function, state space, and action space of the Markov Decision Processes (MDPs) based on the semantics of the query prediction task. We propose two variants of reward functions, Boolean and Numeric, to learn the sequences of queries.
- Besides comparing RNN and Q-Learning against each other, we also implement the query recommendation algorithms of query similarity-based collaborative filtering and matrix factorization from [13, 14]. We uniformly parallelize the prediction phase of all the aforementioned algorithms to preserve the interactivity.
- We compare all the contending algorithms on real-world datasets by conducting two kinds of experiments that we term as *sustenance* and *singularity* to test the effectiveness of ML

models built respectively during offline and online training steps. For our sustenance experiments, we provide an SQL operator-wise breakdown of the prediction performance.
- We also discuss how an SQL query can be re-generated from the predicted fragment embeddings and evaluate the predicted queries against the actual next queries w.r.t. their execution results.

The remaining article is organized as follows: We describe the two real-world datasets we use and the cleaning heuristics we deploy upon the query logs of these datasets to obtain meaningful user sessions, followed by a detailing of our proposed schema-aware fragment embedding approach, the evaluation methods, and the ML approaches compared in the framework. Finally, we describe our experimental findings along with a detailed discussion and analysis of the results.

2 RELATED WORK

Existing literature that is relevant to query workload prediction can be classified into four broad categories: (1) user intent prediction based on interactive data exploration, (2) query recommendation and autocompletion, (3) latency reduction for data exploration, and (4) using ML for related problems such as join cardinality estimation, workload arrival rate prediction, and workload generation.

1) User intent prediction for interactive data exploration (IDE): User intent is formulated by IDE applications in terms of the data that the user is interested in. Systems such as smart drill-down [19], Indiana [16], and SeeDB [48] define *statistical interestingness* heuristics that require that diverse data matching the user interest is retrieved. One notion of statistical interestingness is the surprisingness factor that can be defined through the KL-divergence between the distributions of the data retrieved by a user thus far and the next set of tuples that she would be interested in retrieving. REACT [46] defines a set of data interestingness heuristics such as diversity, dispersion, peculiarity, and conciseness and also captures user session context based on directed acyclic graphs of context trees [34] to detect user intent. DynaCet [42] employs faceted search to identify the attributes to group by (known as facets) and drill down upon to quickly capture the user intent. A decision tree is built by choosing the facets that interest the user as the splitting attributes, which also help in ranking the tuples of eventual interest to the user. At any given point in an exploration session, based on the facets chosen by the user thus far, the facets that might interest the user are recommended from the decision tree such that the user is quickly led to her intended tuples at the leaf nodes. Active learning-based approaches [11, 12, 38, 40] represent the user intent as the last query in a session. Meduri et al. [32] use RNNs to predict the dynamic user intent but the SQL fragment embeddings support simpler next queries upon a single table without constants.

2) Query recommendation and autocompletion: While IDE applications aim to predict data that interests the users in minimal user-database interactions, query prediction moves the abstraction one level up from *data* to *queries* and thus enables broader applications such as speculative query processing, query prefetching, adaptive indexing, and so on, as discussed in Section 1, not restricted to data exploration. Query steering [5] models the transitions among the queries in a user session as a Markov chain and represents the states by exploration operators such as narrow, drilldown, relate, and move, which are equivalent operators to "selection predicate," "aggregation using group by," "join," and "substituting the constant parameters in selection predicates with different values," respectively. However, a simple Markov chain cannot incorporate a reward function to encourage or penalize the transitions during the training and prediction phase, which is why we rely on MDPs that can also capture the goal-oriented exploration that an analytical workload may have. There have also been attempts to map Natural Language keyword queries to the underlying tuples that match the human intent [31] but these techniques cannot be directly applied to

SQL query prediction. Query recommendation [6, 13], however, represents queries as bags of SQL fragments and recommends queries from the historical logs aligning the most with the ongoing interaction session in terms of overlapping query fragments. The problem of sparsity is overcome by using sparse matrix factorization techniques [14] to recommend queries even under the absence of a significantly large user history. Therefore, we compare temporal predictors against query recommenders as baseline techniques in this work.

Another related line of work on autocompletion of queries aims at automatically filling up the missing parts of a query as a user is typing it. Existing works such as Chaudhuri and Kaushik [7] and Deng et al. [10] complete keyword queries by building an initial trielike index on the data and finding the closest *active* nodes from the trie matching the partial keyword query, whose leaf node descendants form the complete keyword queries. Khoussainova et al. [22] auto-complete SQL queries by representing all possible queries as nodes in a directed acyclic graph (DAG) and ranking the transitions among the nodes in a graph based on their conditional probabilities computed from heuristics such as popularity of query fragment co-occurrence in prior logs and foreign key dependencies. The most likely transition (DAG edge) with the highest conditional probability is chosen to identify the complete query (child DAG node) from a given partial query (current DAG node). Although we do not address this problem in the article, using ML algorithms to solve it can be an interesting future direction, given that existing works are based on heuristics.

3) Latency reduction for data exploration: Initial works from the past such as LeFevre et al. [26] reduce the execution latency for an SQL query by rewriting it in such a way that it reuses the materialized views from earlier executions of historical queries. Recent works such as Liang et al. [29] propose the usage of reinforcement learning to decide upon whether or not to materialize a query result into a view by estimating its long-term utility, given the information about the set of materialized views from the past. Data canopy [49] is an effort to save on statistical query exploration by saving the already computed statistics, looking ahead and precomputing results for statistical queries likely to be asked in the future. While "Approximate Query Processing" systems such as BlinkDB [3] work with samples and save on query processing time, more recent efforts such as Verdict build query synopses [39], which help estimate the answers to the future queries based on the answer sets retrieved for the queries asked thus far in the exploration sessions. DICE [18, 20] is a related system that uses faceted exploration over a data cube to ensure that speculative execution of queries that a user might be interested in, happens in sub-second latencies. In an ongoing user session, each current (group by) query is represented by its result facet, and the possible successor facets within the data cube are bounded by pre-defined *roll-up*, *drill-down*, and *pivot* operations on the current facet and are prioritized by accuracy heuristics proposed in Reference [20]. The most likely successor queries are discovered during the user think time and their results are cached for seamless exploration. In our current work, we also exploit the user think time between successive queries to predict the SQL fragments in the next query and also to execute the SQL query re-generated from the predicted fragments. A detailed discussion on the end-to-end query prediction latencies has been reported in Section 5.4.

4) ML for related problems: State-of-the-art ML predictors have been recently used to solve several related problems such as join cardinality estimation, workload arrival rate prediction and workload generation. Kipf et al. [23] use RNNs for an orthogonal purpose of estimating join cardinality in query workloads and therefore capture join and selection predicates from SQL queries in feature vectors. Efforts have also been made to predict the arrival rate of representative query clusters (templates that exclude constants from SQL queries) in a workload [30]. RNNs have most recently been used to recommend data preparation steps in terms of the next operator to apply along with the corresponding schema element (column). Yan and He [52] use RNNs to predict the next SQL operator based on the logs from pre-crawled Jupyter data science notebooks.

Subsequently, for the predicted SQL operator, the schema element such as the columns or tables that co-occur with the operator is predicted separately by using operator-specific heuristics. On similar lines, El et al. [15] use deep reinforcement learning to generate a query workload that can be presented in a data science notebook. We differ from these existing works along the following lines. In contrast to Reference [23], which featurizes a subset of SQL operators, and Reference [52], which predicts one SQL operator at a time, we propose the creation of more comprehensive SQL embedding vectors on wide real-world schemata containing multiple relations and columns. We facilitate the prediction of an exhaustive list of SQL fragments comprising a variety of SQL operators and constants of all data types from any data distribution in contrast to Kipf et al. who encode numerical constants in selection predicates with an underlying uniform distribution assumption on the constant value space. More importantly, References [23, 52] do not allow for complete synthesis of novel SQL queries using RNNs. We accomplish this by employing discretized thresholds and syntax correction heuristics upon the numerical output vectors predicted by RNNs. Contrary to El et al. [15], who use deep reinforcement learning in an unsupervised manner for workload generation and data interestingness metrics in lieu of a reward function, we adapt exact Q-Learning in a supervised manner by learning the <state,action> pairs using a reward function that captures the sequence of queries in a workload.

3 DATASETS, EMBEDDINGS, AND EVALUATION METHODS

Following is a formal definition of our problem at hand. Given a sequence of SQL queries $(qu_\tau)_{\tau=1}^{i} = (qu_1, qu_2, \ldots, qu_i)$ from a user U in a querying session upon a relational database until the current timestep i, our goal is to predict the future query qu_{i+1} issued by the user at the immediate next timestep $i + 1$. Under the scenario that there are several such active user sessions concurrently posing queries to the database, we aim to find the best performing ML algorithm that can predict the next user query for each of these ongoing sessions. To achieve this, we adapt two temporal predictors—Exact Q-Learning and Recurrent Neural Networks—to the problem of query prediction and compare them against two collaborative filtering baselines [13, 14]. We approach this problem in two steps: First, we predict the SQL fragments in the next query. Subsequently, we re-generate the SQL query from the predicted fragments and execute it to obtain its result tuples. We evaluate the ML contenders on the basis of how closely their predicted SQL fragments and the re-generated query execution results match those of the actual next query.

We compare the ML algorithms for query prediction on two real-world datasets. While we curate the query logs from the website of database courses taught at a university to create the first dataset, the second one is obtained after pre-processing the query logs from a publicly available sample of the Bus Tracker dataset at http://www.cs.cmu.edu/~malin199/data/tiramisu-sample/ also used in Ma et al. [30]. We refer to these datasets by the names Course Website and Bus Tracker.

3.1 Description of the Datasets

Course Website is created from the interaction sessions and SQL query logs automatically generated from a website used to teach database courses at a university. The website hosts a rich repository of lecture transcripts, Q & A sessions between instructor and students, and forum discussions amongst the students themselves. Since the course website was created using the Joomla! open-source content management system [37], the website content is automatically stored in the MySQL engine as a relational database forming the backend to the website. When a student logs into the course website, she has a variety of user actions to perform using the web interface such as a scroll on the forum discussions, or a click on a user profile or lecture recording. A student can create, update and delete information on her user profile ranging from bio-data to reading lists, her comments during a discussion on a forum thread, and assignment or homework submissions.

All these actions are internally converted into SQL queries and are logged by the MySQL engine as interaction sessions. Applying next query prediction on this dataset can predict the next action that the user is about to take such as, but not confined to (a) the next search query to retrieve some course information or a search for discussions on a specific topic (DML type = SELECT), (b) posting a lecture recording/material or a response/question on the Q & A forum (DML type = INSERT), or (c) an update to the reading list or the user profile (DML type = UPDATE/DELETE).

Bus Tracker is a mobile application that updates its database periodically with the bus locations (DML type = INSERT/UPDATE) besides allowing the users to live-track the bus location, and find its route information along with the nearest bus stops to their current location (DML type = SELECT) [30]. The user queries are logged in the SQL format while the database backend is stored on a PostgreSQL server. While the database schema and SQL queries are available, the content of the tables (i.e., the tuples) is not publicly available. This is because Ma et al. [30] predict the arrival rate of query templates that exclude constants, and this obviates the need for access to the actual underlying data. In contrast to their work, we predict the fragments within the next query that also comprise ranges of constants in the selection predicates. This requires access to the relational tuples from which we generate equi-depth value range bins (histograms) for each of the attributes (columns) participating in selection predicates. Therefore, our prediction of selection predicates in the next query includes constants and comparison operators only for the Course Website dataset. For the Bus Tracker dataset, the selection predicate prediction is only confined to the attributes that participate in such predicates because of the lack of access to the database tuples.

The information about the user corresponding to each session is not stored in both the Course Website and the Bus Tracker datasets. Upon interacting with the creators of the Course Website dataset, we found that each distinct session only consists of the queries from a single user but a user can create multiple sessions. The same holds true for the BusTracker dataset as well, because a user session involves finding the location of a bus or the nearest bus stop and a user is allowed to access the mobile application several times over distinct sessions. So there is a one-to-many relationship from the users to the sessions in both the datasets. We do not need to identify the user for each session, because our adaptation of ML algorithms does not require such information. The train and test splits for evaluation are created upon a permuted set of sessions that are shuffled enough to eliminate any bias w.r.t. the users who created them, in case consecutive sessions in a dataset are assumed to be from the same user. Each session is fed independently in a user-agnostic manner to the ML algorithms during the training or test phase.

3.1.1 Session-Cleaning Heuristics. An interaction session can be defined as a sequence of queries issued by a user in a given time frame to accomplish an insert/update task (transactional) or to derive an interesting insight (analytical) from the data. Although both the Course Website and the Bus Tracker datasets contain transactional queries, they are predominantly analytical with 89% and 86% SELECT queries respectively that support goal-oriented exploration. The query logs for Course Website and Bus Tracker, stored in MySQL and PostgreSQL respectively, are pre-organized into sessions. For Course Website, a unique session ID is assigned for each interaction and is stored in an *Id* field. Another attribute, *Command*, specifies whether the interaction is an SQL query. For instance, the first few interaction steps in each session involve connecting to and initializing the database. Such queries are marked as “Connect” or “InitDB.” Every other command that involves an SQL statement is marked as “Query.” In the case of Bus Tracker, each session has a distinct ID and all the SQL queries that belong to the same session appear consecutively along with their session ID. We logged the SQL queries from the users of Course Website over a span of 2 weeks and pre-processed the logs using a set of heuristic rules to differentiate the interaction sessions of crawlers (bots) from those that are more humanlike.

Crawler generated sessions can have two properties: "repetition" and "recurrence." Repetitive interactions contain consecutive queries that have the same SQL fragments and the only variation is in the constants. For example, Q1: *select * from Posts where course_id=1;* and Q2: *select * from Posts where course_id=2.* From the Course Website query logs, we prune any session containing such consecutive queries that are entirely the same except for the constants. Even though our query prediction system does predict constants from the selection predicates, having too many of repeated query sequences will make the prediction task trivial and negatively influence the quality of the predictor. To reduce the bias in prediction and to clearly distinguish among the predictive abilities of various ML algorithms, we only retain non-trivial query sequences in our sessions. To achieve this in a time-efficient manner without having to actually convert the SQL query into its fragment vectors, we prune sessions containing consecutive queries whose textual representations have a Cosine similarity ≥ 0.8. This is a conservative similarity threshold chosen after manually examining several sample SQL query pairs. In the case of Bus Tracker dataset, almost every query session contains repetitive query patterns. Although this does not impact Ma et al. [30] whose focus is on predicting the count of query templates, for the purpose of query fragment prediction, it is important that we remove such repetitions. So from each query session, we remove such repeated query patterns to retain non-trivial query sequences.

Recurring patterns are usually concatenations of the same type of interactions that can artificially enhance the length of a session. For instance, there are sessions that are recurring blocks of two queries such as Q1: *select * from Reading_List;* Q2: *select * from Course_Instructors;* Q3: *select * from Reading_List;* Q4: *select * from Course_Instructors.* Such a sequence can prolong to as many as 200 queries that are 100 concatenations of the two SQL queries. We observed that the basic building block of recurring query patterns can contain more than two queries and it is difficult to parameterize the length of a recurring query block. Since we noticed that longer sessions are more likely to be crawler based than shorter ones, we impose a session length limit of 50 (a conservative threshold chosen after a manual examination of several random samples of user sessions) and thereby restrict the number of queries in a session to ensure that the recurring patterns are human-intended. Any session containing more than 50 queries is not included into the clean set of query logs. This also reflects typical user behavior and allows our dataset to contain non-trivial query sequences capturing realistic human interactions. This heuristic is applied uniformly for both the datasets to sufficiently eliminate bot-generated sessions from our session logs.

Our data cleaning heuristics are similar to those used in Singh et al. [45]. Botlike patterns involving repetition and recurrence can easily be learned by any ML baseline even with aggressive sampling, thus bringing no significant insight about the best performing approach. It would be unfair to the ML algorithms if their performance is compared upon the bot-generated sessions as the conclusion drawn from such an experiment would not be meaningful. This is because, without applying the pre-processing techniques, simply predicting that the next query will most likely be the same as the current query gives an extremely high prediction quality to all the ML algorithms.

Our preprocessed query logs consist of 114,607 SQL queries and 43,893 clean sessions from Course Website, while the Bus Tracker dataset contributes to 5,640 clean sessions containing 22,106 SQL queries. The raw datasets contain 214 M and 25 M queries, respectively. Table 1(a) contains an SQL operator-wise distribution of various types of queries. We can notice that SELECT queries are the most common followed by UPDATEs. A large percentage (98.58% from Course Website, 100% from Bus Tracker) of the queries are associated with projection (π) lists followed by selection (σ) predicates (97.53%, 95.21%). The join ($\bowtie$) predicates are significant enough for Bus Tracker (25.74%) but are limited for Course Website (3.66%). Sort/ORDER BY predicates (27.59%, 12.95%) and aggregate operators involving COUNT (9.79%, 13.16%) are prominent in both the datasets. LIMIT keyword occurs more often in Course Website (20.59%) as compared to Bus Tracker (0.85%).

Table 1.

(a) Operator-wise Query Distribution

Fragment	**%Queries (Course Website)**	**%Queries (Bus Tracker)**
Query (DML) type = SELECT	88.93	86.0
Query (DML) type = INSERT	2.26	3.4
Query (DML) type = UPDATE	7.63	10.6
Query (DML) type = DELETE	1.18	0.0
Projection (π)	98.58	100
Selection (σ)	97.53	95.21
Join predicates ($\bowtie$)	3.66	25.74
GROUP BY	0.004	0.0
Sort (ORDER BY)	27.59	12.95
Aggregate (MAX)	0.007	0.0
Aggregate (SUM)	0.034	0.0
Aggregate (COUNT)	9.79	13.16
LIMIT	20.59	0.854

(b) Query Fragments in the Embedding Vector

Fragment	**#Dimensions Course Website**	**#Dimensions Bus Tracker**
$Query\ Type_{vec}$	4 (SELECT/UPDATE/INSERT/DELETE)	4
$Relation\ List_{vec}$	#Tables = 113	95
Π_{vec}	#Columns = 839	770
$Aggr_{vec}$	#Columns x 5 = 4195 (AVG,MIN,MAX,SUM,COUNT)	3,850
σ_{vec}	#Columns = 839	770
$GROUP\ BY_{vec}$	#Columns = 839	770
$ORDER\ BY_{vec}$	#Columns = 839	770
$HAVING_{vec}$	#Columns = 839	770
$LIMIT_{vec}$	1	1
$\bowtie (JOIN)_{vec}$	# {LeftTable.Column, RightTable.Column} = 92045	1,355
$\sigma.OP_{vec}$	# σ.Columns (=109) x 7 = 763 ($=, \neq, \leq, \geq, <, >$,LIKE)	N/A
$\sigma.CONST_{vec}$	#Equi-depth range bins for σ.constants = 704	N/A

(c) Relation List (Table) Transition Statistics

Dataset	**No Change**	**Partial Change**	**Total Change**
Course Website	16.63%	0.209%	83.15%
Bus Tracker	6.17%	21.54%	72.28%

Since a relation-list (tables) is the most vital component that may change across consecutive queries, we collected the statistics about table transitions between successive queries in the clean sessions. Table 1(c) shows that the relation list changes completely a majority of the times (83.15%, 72.28%) between successive queries. While Bus Tracker allows for partial overlap among the relation list between consecutive queries (21.54%), Course Website has very few such instances (0.209%). This tells that the prediction of the relation list has to be accurate as there could be a limited scope for partial rewards. In the next sub-section, we will describe the construction of feature embeddings for the task of query fragment prediction.

3.2 Schema-aware Query Fragment Embeddings

We represent each SQL query by a one-hot encoded feature vector that is a bitwise representation of the fragments that occur in the query. We create individual bit vectors for each fragment, the concatenation of which produces a holistic fragment embedding for the entire query. The fragment embedding has a fixed length uniformly for all the SQL queries, because the dimensionality of the bit vector is dependent on the SQL semantics and the underlying database schema. The fragment embedding qu_{vec} for a query qu is produced by concatenating the bit vectors of several fragments in a fixed order as follows: $qu_{vec} = Query\ Type_{vec} Relation\ List_{vec} \Pi_{vec} Aggr_{vec}\ \sigma_{vec}\ GROUP\ BY_{vec}\ ORDER\ BY_{vec}\ HAVING_{vec}\ LIMIT_{vec} \bowtie_{vec} \sigma.OP_{vec} \sigma.CONST_{vec}$.

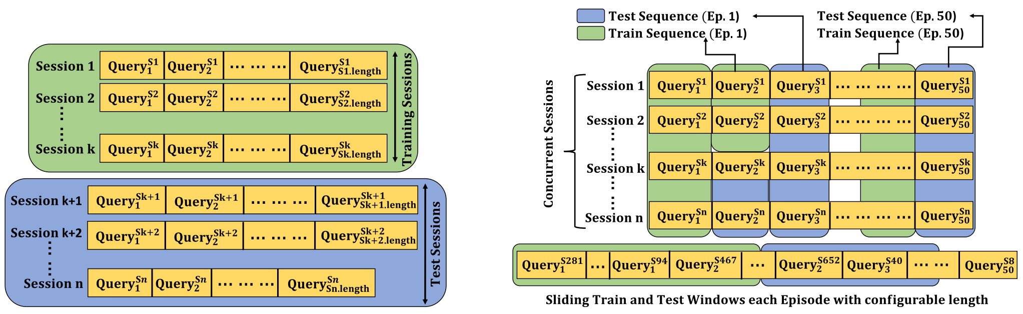
3.2.1 SQL Operator Fragments. Table 1(b) shows the number of bits pre-allocated to each fragment. Query (DML) type is indicated by 4 bits, each of which stands for one of SELECT, UPDATE, INSERT, or DELETE. Likewise, the relation list in the FROM clause can possibly include one or more tables from the underlying database schema. The attributes participating in the projection list, selection predicates, GROUP BY, ORDER BY, and HAVING clauses are captured using individual bit vectors each of which has a dimensionality equal to the number of attributes $|Attr|$ in the database schema, indicating the #columns that these operators can be associated with. $Aggr_{vec}$ is a concatenation of five most common aggregate operators (AVG, MIN, MAX, SUM, and COUNT) and thus has a dimensionality of $|Attr| * 5$ bits. $LIMIT_{vec}$ records the presence of the LIMIT keyword in the query and uses a single bit as it is not associated with a schema element. We capture both self-joins and multi-table joins through possible join fragments that can occur in a query. A join fragment is defined as the pair of columns that occur in a join predicate. Although the possible predicates can be combinatorial in the number of attributes (${}^{|Attr|}C_2$), we reduce them to 92,045 and 1,355 for the datasets by only allowing columns of the same data type to participate in a join predicate. To handle the huge dimensionality of the join fragment vectors, we prune column pairs with mis-matching data types from the candidate space of join predicates and also exclude arithmetic comparison operators such as =, <, > from a join predicate. Nevertheless, we represent the comparison operators along with the value range bins for constants in the selection predicates of a query.

3.2.2 Selection Predicate Constants and Comparison Operators. Contrary to the embedding bit vectors for the operator fragments that record the occurrence of SQL operators with the schema elements (tables or columns), the bit vectors for the comparison operators ($\sigma.OP_{vec}$) and constant range bins ($\sigma.CONST_{vec}$) make certain assumptions. While the former are created in a generic manner for the entire schema, in the case of the latter, we assume that we are privy to the set of columns that occur in the selection predicates across the entire workload. Note that this assumption is only to represent the comparison operators and constant ranges in the selection predicates. To generate the bit vector for columns that participate in the selection predicates (σ_{vec}), we do not make any assumptions. Of 839 columns in the database schema for Course Website, we notice that 109 columns participate in the union of selection predicates across the 114,607 queries. A selection predicate can be denoted as {ATTR, OP, CONST-BIN} (see Section 1) from which the bit vector for attribute, ATTR, is generic and gets a full dimensionality of the total #columns (839 for this dataset) in the schema. To represent the comparison operators ($=, \neq, \leq, \geq, <, >$,LIKE) we need 7 bits per column that can potentially turn the dimensionality into $839 \times 7 = 5{,}873$, but we restrict the representation to 109 columns, which allows our bit vector for $\sigma.OP_{vec}$ to contain $109 \times 7 = 763$ bits instead. Along similar lines, we also represent the value range bins, CONST-BINs, for the constants in the selection predicates on the set of 109 attributes as described below. As mentioned, we predict the constant bins and comparison operators only for the Course Website dataset but not the Bus Tracker dataset due to the lack of availability of actual data (tuples) for the latter.

We partition the distinct values from each of the 109 attributes of the Course Website schema that can possibly occur in the selection predicates of a query into 10 equi-depth range bins, where depth denotes the tuple frequency of a distinct column value. For example, an attribute ranging from 0 to 100 may get 10 range bins with one of the bins getting a value range of $\{0 - 40\}$ if it is terribly sparse as compared to the other ranges. We thus partition the total tuples into multiple bins such that each bin approximately contains the same number of tuples. However, it should be noted that if the cardinality of the entire set of distinct values from a column is lesser than 10, then we produce fewer than 10 value range bins for that column. A constant in the selection predicate of a query from a matching column may fall into one of these range bins or may not belong to any of these bins in which case it will belong to an 11th default bin that is used to represent NULL values. This is because some of the selection predicates check whether a column IS NULL or IS NOT NULL, which we translate into "=" for the comparison operator and the "NULL" bin for the constant. Out-of-range constants in a query that do not match any of the distinct values of a column will be defaulted to the NULL bin. Although we anticipated the dimensionality of $\sigma.CONST_{vec}$ to be $109 \times 11 = 1{,}199$, we ended up with 704 bits for the constant value range bin vector because of the inherent skew in the value distribution for some columns. Unlike Kipf et al. [23], who assume uniform distribution for a column and represent constants as normalized values between 0 and 1 to estimate join cardinalities, we construct equi-depth range bins to be resilient to skew. While the feature vectors in Reference [23] are specific to join cardinality estimation using RNNs and the constants are only applicable to numerical data types, our proposed embeddings support the prediction of an exhaustive list of SQL operators and constants of all data types, regardless of the data size. In addition to this, we apply these embeddings upon a host of ML algorithms not restricted to RNNs.

We present the creation of schema dictionaries in Online Appendix A.1. After we create the schema dictionaries, we parse each query in the interaction workload using JSQLParser[1] and obtain the operator fragments in the query. Our fragment embedding creation detects the presence of nested queries (with any number of levels) and adds additional selection or join predicates to reflect the correlation between the outer query and the inner query. In the case of nested queries containing IN and NOT IN, we add the corresponding selection predicate or join predicate fragments depending on whether the sub-query is an expression of constant value list or an actual SQL query containing a projection list. A limitation of our query fragment representation is that it includes the fragments from both the outer and the inner queries into the same vector. To support nested queries more accurately, we need to rewrite the nested queries into equivalent non-nested, multi-join queries. On similar lines, we do not explicitly support user-defined functions. To use our fragment embedding, the user-defined function needs to be inlined with the corresponding database operators and re-written as an equivalent SQL query. This support requires more of an engineering effort that we plan to include in the future.

For INSERT and UPDATE queries, the projection list is parsed as those columns into which values are inserted or updated, while DELETE queries do not have a projection list. Once the parsed fragments are obtained, we look up the bit positions for each fragment in the schema dictionaries and set them in the embedding vector of the query. It is important to note that the embedding generation time also adds to the response time as for each query, we create an embedding and feed it to the query predictor in Figure 1 to predict the next query. However, we perform the embedding vector generation in an offline step for the entire query workload, because all the ML algorithms use the same set of embedding vectors and hence the pre-processing time for embedding creation remains the same. Moreover, the sequence of queries the ML algorithms predict a successor for



(a) Sustenance - Train and Test Splits (b) Singularity - Incremental Train and Test Splits

Fig. 2. Sustenance and singularity evaluation methods.

is exactly the same for all the approaches. We report the time consumption for offline embedding generation in Section 5.4.1.

3.3 Evaluation Methods

Our evaluation is based on two kinds of experiments. To measure the effectiveness of the ML models learned from offline training, we partition the entire set of interaction sessions from each dataset into 80% train and 20% test sets as shown in Figure 2(a). After training a model on 80% of the sessions, we evaluate its prediction quality w.r.t. next query (fragments) prediction upon the remaining 20% sessions. We term this evaluation method as *sustenance*, because we check whether a pre-learned model built from offline training can yield a sustained, high-quality performance in predicting the successor for each query from the fixed test set of sessions. Likewise, we also apply a prequential *test-then-train* model used to evaluate ML models in a streaming data scenario. We use sliding windows as shown in Figure 2(b) to first test and then incrementally refine the model upon a fraction of the query sequence that is made available. We assume that several sessions are simultaneously active and are concurrently posing queries. Therefore the queries arrive from a permuted sequence of sessions in each test-then-train episode. Without loss of generality, we assume that the ith queries across all the sessions are executed before the $(i + 1)$th queries are posed from any of these sessions. Note that the queries from concurrent sessions are streamed in the exact same order for all the ML algorithms to ensure a fair comparison.

In Figure 2(b), at the beginning of each episode, an incoming batch of queries streams into the next query predictor system from concurrent sessions. The batch size is configurable, and the model that has been learned thus far is first tested on this fresh batch of queries. The testing phase can be defined as predicting the next query for each query arriving from the currently streaming batch. Once the batch is exhausted at the end of the episode, the query predictor ML model learned so far is incrementally updated based on the current batch, before the next batch of queries streams in. These experiments are titled as *singularity*, because we check whether a convergent (singular) model of high quality can be learned at all, and if so, then we measure the #episodes of training it takes before reaching such a self-managed state. We evaluate each ML algorithm w.r.t. F1-scores computed from the overlap of both fragments and execution result set of tuples between the predicted and actual next queries, training and test latencies as well as the memory consumption.

4 COMPARED ML APPROACHES FOR NEXT QUERY PREDICTION

In this section, we describe the ML approaches that we adapt for query prediction, in the following order: Query recommendation using collaborative filtering, temporal learning using RNNs, and

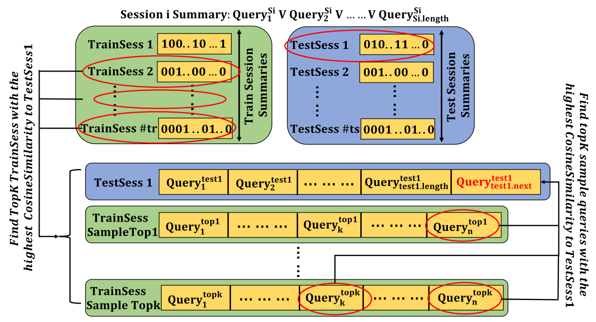

Session i Summary: Query$_1^{Si}$ V Query$_2^{Si}$ V ... ... V Query$_{Si.length}^{Si}$
TrainSess 1 100..10...1
TrainSess 2 001..00...0
TrainSess #tr 0001..01..0
Train Session Summaries
TestSess 1 010..11...0
TestSess 2 001..00...0
TestSess #ts 0001..01..0
Test Session Summaries
Find TopK TrainSess with the highest CosineSimilarity to TestSess1
TestSess 1 Query$_1^{test1}$ Query$_2^{test1}$ ... ... Query$_{test1.length}^{test1}$ Query$_{test1.next}^{test1}$
TrainSess SampleTop1 Query$_1^{top1}$ ... ... Query$_k^{top1}$ ... ... Query$_n^{top1}$
TrainSess Sample Topk Query$_1^{topk}$ ... ... Query$_k^{topk}$ ... ... Query$_n^{topk}$
Find topK sample queries with the highest CosineSimilarity to TestSess1

Fig. 3. Cosine similarity-based collaborative filtering.

Exact Q-Learning, which is a reinforcement learning algorithm. We discuss how each of these ML algorithms is applied to the task of next query prediction.

4.1 Collaborative Filtering

We implement two Collaborative Filtering (CF) approaches for query recommendation: One is a Cosine similarity-based approach followed in QueRIE [13] and the other is a matrix factorization-based approach from Eirinaki and Patel [14]. Note that we have enhanced these existing approaches w.r.t. both latency and memory consumption to be scalable over huge query logs. The proposed optimizations are fairly generic and have been used for temporal predictors as well.

4.1.1 Cosine similarity-based CF. To adapt Cosine similarity-based CF for next query prediction, we represent each user session as a summarized bit vector of query fragments that are present among all the queries in the session. As mentioned in Figure 3, we obtain a session summary by applying a bitwise OR upon the individual bit vector (fragment) embeddings of all the queries in the session. Recall from Section 3.3 that we evaluate each ML algorithm upon two kinds of experiments: sustenance (80% train, 20% test) and singularity (interleaved episodic test-then-train approach used in streaming data scenarios). For both these experiments, our trained model is the set of summaries built on the training sessions. In sustenance experiments, the train sessions are strictly non-overlapping with the test sessions. However, for singularity experiments, the train summaries keep growing with #episodes as more sessions and more queries from existing sessions keep accumulating. Hence, in each episode, some of the streaming batch of queries may belong to the set of ongoing training sessions whereas, others may belong to fresh sessions that start from the current episode.

Given a test session TestSess 1 as shown in Figure 3, with "test1.length" #queries in it so far, to predict the next query in the sequence, Query$_{next}^{test1}$, we first compute the top-K sessions from the training set whose summaries have the highest Cosine similarity with that of TestSess 1. Of these top-K train sessions, we find the top-K queries whose fragment embedding bit vectors have the highest Cosine similarity with the bit vector summary of TestSess 1. Following are a few important things to take note of:

- While matching a test session with the training sessions, we avoid the comparison of a test session with itself, as test and train sessions can be non-overlapping in streaming scenarios.
- Although the train session summaries are complete and are computed over all the train sessions and queries seen until a given episode, identifying the top-K sessions and top-K queries similar to a test session employs sampling. This is similar to QueRIE [13] that deploys random sampling upon historical sessions to obtain a pool of candidate queries

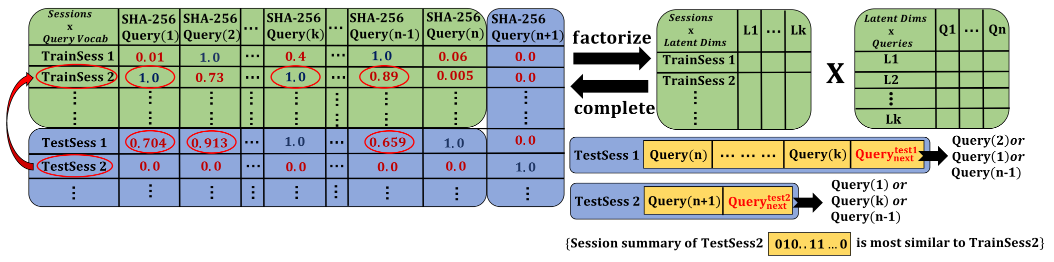


Fig. 4. NMF-SVD-based collaborative filtering.

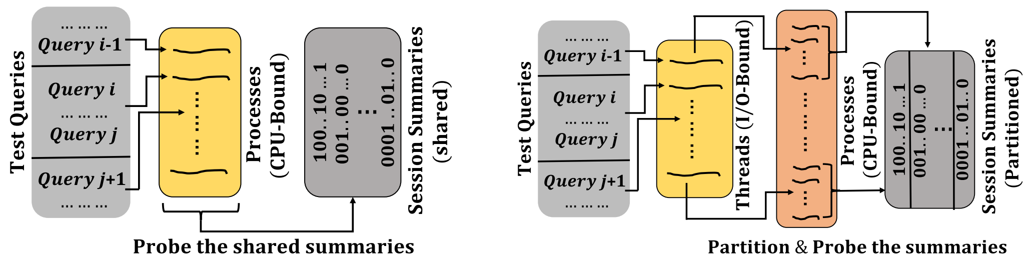
from which top-K queries are recommended. The only difference is that in streaming scenarios, the sessions and queries get updated with time that necessitates the sampling to be done before query prediction in each episode.

- For effective sampling, we keep track of the distinct queries (which have unique fragment embedding bit vectors) in each session.

4.1.2 Matrix Factorization Based CF. We represent the historical (training) queries as a matrix of |Train Sessions| × |Query Vocabulary| dimensionality, in which training sessions form rows and distinct queries constituting the query vocabulary seen thus far indicate the columns of the matrix. Similarly to Eirinaki and Patel [14], we use Non-negative Matrix Factorization-based Singular Value Decomposition (NMF-SVD) [25] from the scikit library to decompose the matrix into two latent factor matrices of dimensionalities |Train Sessions| × |Latent Dims| and |Latent Dims| × |Query Vocabulary|, respectively, as shown in Figure 4. The original matrix contains 1.0 in specific cells to indicate the queries that occur in each of the training sessions. Note that a distinct query from the vocabulary represents a unique set of query fragments. Once we multiply the factored matrices, the original matrix gets completely filled up where the cell entries represent the probability with which a query (column) may occur in a specific session (row). As mentioned, in the case of sustenance experiments, train and test sessions do not overlap with each other. However, in singularity experiments, there is a possibility that train and test sessions do overlap.

In Figure 4, TestSess 1 indicates a test session that is already present in the training set of sessions. Because of the streaming nature of queries in the singularity experiments, new queries are getting appended to that session. Let us assume that Query(k) and Query(n) along with a few other training queries are already present in TestSess 1 and now we need to predict the next query $\text{Query}_{\text{next}}^{\text{test1}}$. Since this is a row that is already present in the factorized and completed matrix, we pick the top-K cells with the highest cell probabilities from the NMF completion as the possible next queries. For TestSess 1, the top-K ($k = 3$) queries can be either Query(2) or Query(1) or Query (n-1). In the case of singularity, the updated test session with the actual succeeding query eventually becomes a part of the updated matrix, toward the end of the test-then-train episode. Hence, the matrix factorization and completion happens in each episode as a part of the training process.

TestSess 2, however, represents a test session that was unseen in the training set of sessions. While this is possible in both streaming and non-streaming scenarios, this is more likely to happen during the sustenance (80% train, 20% test sessions) experiments as the train and test sessions are strictly non-overlapping. In such situations, we can notice that a query that occurs in a test session can totally be from the training vocabulary of seen queries. Query(n+1) in Figure 4 represents one such out-of-vocabulary query that occurs in TestSess 2. If we need to predict the next query, $\text{Query}_{\text{next}}^{\text{test2}}$, for such out-of-vocabulary test session, then we cannot rely on this session alone as it is not present in the completed matrix. To tackle this cold-start problem for out-of-matrix sessions, we keep track of the summaries for a sample set of training sessions. Since a session summary is



(a) Processes for Inter-Query Parallelism

(b) Nested Parallelism with Threads (Outer, Inter-Query) and Processes (Inner, Intra-Query)

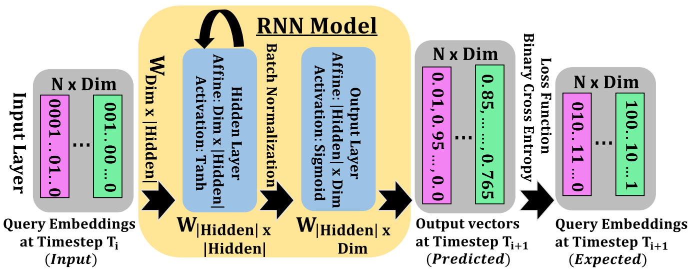
Fig. 5. Parallelism for optimized next query prediction (collaborative filtering).

a bitwise OR of all the query embeddings from the session, Cosine similarity can be computed between each sampled training session and the summary of the ongoing out-of-query-vocabulary test session. The closest training session already in the matrix can be used to predict the top-K next query candidates for the ongoing test session. In Figure 4, we find that TrainSess 2 is most alike to TestSess 2, from which the top-K queries are suggested as the next query candidates for TestSess 2. We sample session summaries to save on the computation of similar sessions.

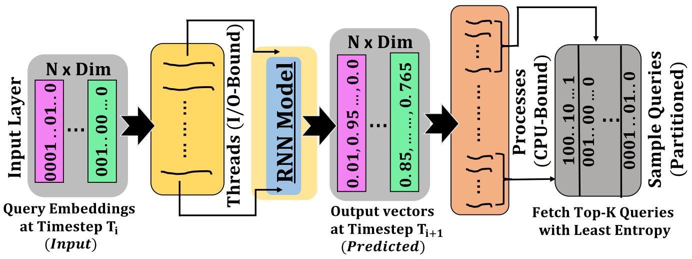
4.1.3 Optimizations. We present the computational complexities of both the adaptations of collaborative filtering in Online Appendix A.2.1. To lessen the computational costs, we implement two optimizations. These optimizations are generic and are used by temporal predictors as well.

- To simplify the computation of distinct queries during the training phase, we create a hash map of key-value pairs in which the key represents a 256-bit secure hash encryption (SHA-256) of each distinct query embedding and the value is a <sessionID,QueryID> reference to the query embedding. The computation of distinct query embeddings compares only the compact & unique SHA-256 key values instead of the embeddings that can be as long as 102,020 bits for Course Website and 9,155 bits for the Bus Tracker dataset (see Table 1(b)).
- We parallelize the prediction phase of the ML algorithms to be scalable to huge #sessions and #queries, the details of which are given below.

For all the ML algorithms that we implement, the prediction (test) phase latency increases with #queries and, hence, we parallelize the same. Python threads are I/O-bound and the Global Interpreter Lock allows for only one thread to be executed at any given point of time. Therefore, we use multiprocessing to partition the test queries among several CPU-bound processes. By default, the memory resident data structures are duplicated by processes. To avoid that, we use the multiprocessing manager to create shared data structures that can be accessed by several processes. As shown in Figure 5(a), we use processes for inter-query parallelism and to predict the next query that follows each incoming test query from an ongoing test session, the processes probe the shared session summary samples to find similar sessions. We also allow for nested parallelism in which the outer level uses threads for inter-query parallelism, and each thread in turn spawns several processes for intra-query parallelism. These processes partition the session summaries and compute the Cosine similarity between the ongoing session and the historical sessions in parallel. Once all the child processes finish the probe within their respective partitions, they notify the parent thread that computes the top-K similar historical sessions for a single test query. Typically, we use nested parallelism only if a single level of multi-process parallelism is infeasible. This is because, in the case of nested parallelism, speed-up comes only from the processes that are spawned by the threads for intra-query parallelism, and not from the threads per se.


RNN Model
Input Layer
N x Dim
0001..01..0
001..00...0
Query Embeddings at Timestep T_i (*Input*)
$W_{Dim\ x\ |Hidden|}$
Hidden Layer Affine: Dim x |Hidden| Activation: Tanh
Batch Normalization
$W_{|Hidden|\ x\ |Hidden|}$
Output Layer Affine: |Hidden| x Dim Activation: Sigmoid
$W_{|Hidden|\ x\ Dim}$
0.01, 0.95 ..., 0.0
0.85, ..., 0.765
Output vectors at Timestep T_{i+1} (*Predicted*)
Loss Function Binary Cross Entropy
010..11...0
100..10...1
Query Embeddings at Timestep T_{i+1} (*Expected*)

(a) Training Phase


Input Layer
N x Dim
0001..01..0
001..00...0
Query Embeddings at Timestep T_i (*Input*)
Threads (I/O-Bound)
RNN Model
0.01, 0.95 ..., 0.0
0.85, ..., 0.765
Output vectors at Timestep T_{i+1} (*Predicted*)
Processes (CPU-Bound)
100..10...1
001..00...0
0001..01..0
Sample Queries (Partitioned)
Fetch Top-K Queries with Least Entropy

(b) Parallelized Prediction (Test) Phase: *Historical-RNN*

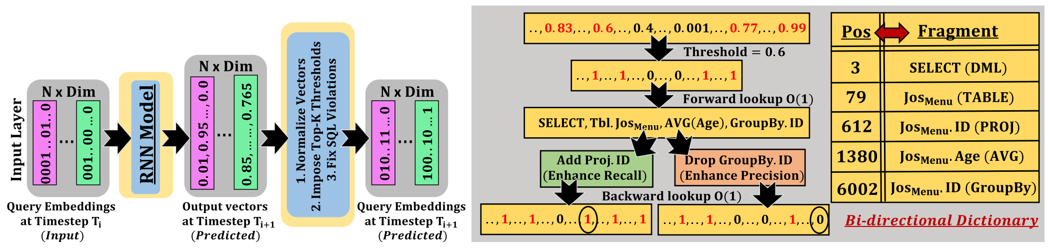
Fig. 6. Recurrent neural networks for next query prediction.

4.2 Recurrent Neural Networks

RNNs [21, 36] are powerful temporal predictors, and, hence, we adapt them to the task of next query prediction. Given an SQL fragment embedding for a query at timestep T_i, the goal is to predict the next query (fragment vector) issued by the user at timestep T_{i+1}. Figure 6 illustrates the training as well as the prediction (test) phases of this adaptation. Contrary to collaborative filtering, RNNs are streaming friendly and are easy to train incrementally.

The training data for our adapted RNN is always fed as pairs of embeddings -<$Query_i$, $Query_{i+1}$> from which $Query_i$ is treated as the query at the current timestep T_i and $Query_{i+1}$ is the query to be predicted from the next timestep T_{i+1}. We also use a batch normalization layer and drop-out regularization for stable predictions. The parameter settings for each ML approach are discussed in Section 4.4. The prediction phase of RNNs can pick the top-*K* next query candidates from the historical pool of queries seen so far. As our evaluation shows that historical RNNs (RNN-H) are poor in prediction quality and latency, we propose "RNN-Synth" (RNN-S), which synthesizes novel next queries.

4.2.1 Historical RNNs. As shown in Figure 6(b), the trained RNN Model is used to predict the next queries to the input queries that are fed at the input layer from several instances of T_i. We use inverse binary cross entropy as the similarity function to compare the numerical output vector produced by an RNN against the one-hot historical query embeddings and pick the top-*K* queries with the least entropy. Also, we use random sampling similar to CF techniques discussed in Section 4.1 to alleviate the latency associated with top-*K* next query detection. The only difference is that CF techniques build their models that entirely depend on the session structure (Cosine similarity-based CF uses session summaries as a model while NMF-SVD-based CF uses sessions as rows of the matrix). Therefore, CF techniques use two-stage sampling to first select the sample of sessions, from which queries are sampled again. In contrast, RNN models are session-agnostic as their training data consists of <current query, next query> pairs. Thus, in this case, the sample



(a) Prediction (Test) Phase (b) Synthesizing Next Query from Normalized Output Vector

Fig. 7. Recurrent Neural Networks (RNN-Synth) for Next Query Synthesis.

queries are directly picked from the entire set of distinct historical queries (which use SHA-based optimization) regardless of the sessions they belong to. We speed up query prediction by using nested parallelism.

We use Keras wrapper [9] with TensorFlow [2], which allows for sharing an RNN model among several Python threads but not among various processes. Hence, the input queries are partitioned among several threads (inter-query parallelism) and are simultaneously sent to the RNN model that emits the predicted probabilistic output vectors. Each thread spawns several processes as shown in Figure 6(b) that partition and probe the historical sample query embeddings to retrieve the top-K queries with the least entropy for a single query (intra-query parallelism). As we will show in the experiments, to tackle the shortcomings of historical RNNs, we propose *RNN-Synth*, which can synthesize novel next queries of high quality while incurring minimal latency.

4.2.2 Synthesizing Next Query Fragment Vectors Using RNN-Synth. While the training phase of RNN-Synth remains the same as that of historical RNNs as shown in Figure 6(a), the difference is in the prediction phase. Instead of relying on historical query samples, RNN-Synth synthesizes the next query (fragments) directly from the output probabilistic vectors. This requires the prediction phase to infer the *next* query embedding, which is a multi-dimensional bit vector, from the probabilistic output vector. There are a few challenges in achieving this.

- Converting a probabilistic vector into a Boolean vector requires setting a suitable probability threshold. All the dimensions whose probabilities are above this threshold will have their bits set to 1. However, in several output vectors, the highest probabilities can be lesser than 1.0 (around 0.7, for example), which necessitates different thresholds for different queries.
- It is not necessary that the Boolean vector obtained via thresholds corresponds to a meaningful SQL query. We will need to make fixes to possible SQL violations that may arise. An implicit requirement is that such fixes need to be made in real time with minimal latency.

Figure 7(a) shows the prediction phase of RNN-Synth in which N output probabilistic vectors of Dim dimensions undergo three transformation steps to produce the next query embeddings. These are (a) output vector normalization, (b) top-K threshold application on the normalized output vectors, and (c) fixing the SQL violations in the next query embeddings. For output vector normalization, we transform each probabilistic dimension such that the least and the highest values are 0 and 1, respectively. For each dimension index i ranging from 0 to Dim, $\text{Output}[i] = \frac{\text{Output[i]-min(Output)}}{\text{max(Output)-min(Output)}}$, where min(Output) and max(Output) denote the minimum and maximum probability values from the output vector. This normalization step allows us to apply uniform thresholds on all the test queries regardless of the original probability value distribution within the output embedding vector. From the [0,1] range, we choose three discrete thresholds

(0.8, 0.6, and 0.4) for top-K when $K = 3$. All the bits corresponding to dimensions whose output probabilities exceed 0.8 will be set to 1 to obtain the top-1 result. Likewise, top-2 and top-3 predictions are chosen by setting the bits for those dimensions whose output probabilities exceed 0.6 and 0.4, respectively. With increasing K, we discretize the [0,1] range at a fine granular level, thereby obtaining more thresholds. For instance, if we were to choose the top-10 next queries, then we can choose $\{0.0, 0.1, 0.2, \ldots, 0.8, 0.9\}$ as the thresholds if we want to be recall friendly. However, if we want to be conservative and retain high precision, then we can set a minimum threshold of 0.35, for instance, and choose $\{0.35, 0.4, 0.45, \ldots, 0.8\}$ as the top-10 thresholds. In our experiments, we set K to 3. Figure 7(b) shows how a threshold of 0.6 is used to convert a normalized output vector into a multi-dimensional bit vector. The values highlighted in red exceed the threshold and are hence set to 1 while the other dimensions remain unset at 0. As mentioned, this bit vector may not represent a meaningful SQL query.

To regenerate the SQL query fragments from the bit vector and vice versa, we create a bi-directional dictionary that stores the bit position and the corresponding SQL fragment as a <key,value> as well as <value,key> pair. Thus we can key in either the bit position or the fragment to obtain its counterpart from the bi-directional dictionary. In the example shown in Figure 7(b), to regenerate the SQL query from the embedding, we need several constant time O(1) forward lookups on the bi-directional dictionary. However, we can notice a violation in the SQL fragments, which is to have a Group By operation applied to the ID column from Jos_{Menu} table without having ID as a part of the projection list. Now we have two options: either include Jos_{Menu}.ID column into the projection list or drop the Group By operation on the same. While the former is a heuristic aimed at enhancing recall, the latter enhances precision. Although we support both, we present results using the former heuristic of adding missing fragments, which is better than dropping existing fragments. This is because the query embeddings are originally sparse, and dropping set bits is going to make the predicted embedding even sparser while not significantly improving prediction quality. To add or drop query fragments from the embedding, their corresponding bit positions are required for which we make another O(1) backward look-up on the bi-directional dictionary as shown in the figure. Following is a list of possible SQL violations that require similar fixes. Note that we use the terms column and attribute interchangeably.

(a). <u>Column-Table Violations</u>: This is the case where for a column that is in the projection or aggregate (AVG / MIN / MAX / SUM / COUNT) or GROUP BY or ORDER BY or HAVING clauses, we do not find the table that it belongs to within the relation list of a query. To fix such a violation being recall friendly, we add the missing table to the relation list of the query embedding. Alternatively, a conservative precision-oriented fix would be to drop the column.

(b). <u>Join Violations</u>: If one or more tables for the columns participating in a join predicate are not in the relation list of a query, then we add the missing tables to the relation list for enhanced recall. For enhanced precision, we drop the join predicate. An example join predicate can be LeftTable.LeftCol = RightTable.RightCol, and one or both of the LeftTable and the RightTable may not be in the tables (relation list) of the FROM clause.

(c). <u>Group By Violations</u>: There are two possible kinds of violations. In the first variety, the projection list contains a column that does not belong to the group by list when a GROUP BY clause is present. To fix this, we either add the projected column to the group by list for recall enhancement or drop it altogether if we were to favor precision. The second type is a symmetric violation in which a group by column neither belongs to an aggregate operation in the projection list nor is it individually projected. We fix this by adding the group by column also to the projection list (for recall) or dropping it from the group by operation (for precision).

(d). <u>Having Clause Violations</u>: If a column is present in the HAVING clause, but does not have an aggregate operator associated with it, then we either add an aggregation to the column (for recall)

or drop the column from the HAVING clause (for precision). The most likely aggregate operator among the five operations (AVG / MIN / MAX / SUM / COUNT) with the highest probability in the normalized output vector is added for enhanced recall.
(e). Selection Predicate Violations: A selection predicate fragment consists of three components: a selection column, a comparison operation (one of =, ≠, ≤, ≥, <, >, LIKE), and a constant range bin. The violations can also be of three kinds: (a) a selection column may not have either the comparison operation or the constant range bins (or both) present in the query, (b) a comparison operation occurs along with a selection column but the column or a corresponding constant range bin may not be present among the set bits for the query embedding, or (c) a constant range bin may have its bit set in the embedding when either the corresponding column or the comparison operator (or both) are absent from the predicted selection predicate.

The fixes are fairly generic and symmetric across all the three possible violations. If the missing element is a column, then we can recognize it unambiguously as the column that is associated with the comparison operator and constant range bin. Instead, if the missing element is a comparison operator or a constant range bin, then it is set to be the most probabilistic dimension and included into the query fragments for recall enhancement. For example, a selection predicate embedding sub-vector (σ_{vec}) may contain a bit set for an "Age" column from Jos_{Menu} table (Jos_{Menu}.Age) but neither the comparison operator nor the constant range bin might have been set. These are picked to be those fragments with the highest probabilities in the normalized vector, although they might not have exceeded the threshold. Assuming an example threshold 0.6, a comparison operator < and a constant range bin [15-20] may have respective probabilities 0.32 and 0.44 but these may be the most likely bins if we must pick them for the age attribute. In such a case, the corrected selection predicate in the predicted query contains the fragments {Jos_{Menu}.Age (*selection column*), < (*operator*), [15-20] (*equi-depth range bin within which the actual constant may lie)*}. Alternatively, to favor precision, the fragment causing the violation is dropped from the predicted embedding.
(f). Null Vital Fragments: If there are no bits set in the predicted embedding for most vital fragments such as DML type, tables (relation list), projection list, then we default them to the fragments within the current query for which the next query is being predicted. In cases where the relation list is present in the query but the projection list is NULL, we select the most likely column from one of the relations into the projection list.

For each test query, we need to normalize its embedding, impose a threshold and fix the violations. While normalization and threshold application require a scan on all the Dim dimensions in the embedding, fixes only require O(1) forward and backward look-up operations upon the bi-directional dictionary for specific dimensions within the embedding whose count is far below Dim. Therefore, prediction using synthesis-based RNNs is exceptionally fast. Another important thing to note here is that while we use nested parallelism for historical RNNs, we do not use any parallelism for synthesis-based RNNs. If we resort to inter-query parallelism using multiple processes, then they make several copies of the RNN model (because Keras does not allow for model sharing among processes) during the prediction phase that turns out to be memory intensive. Although threads can share the RNN model produced by Keras, they are I/O-bound and cannot bring any latency benefits for in-memory, CPU-bound tasks. Nested parallelism does not apply to synthesis-based RNNs as there is not much scope for intra-query parallelism unlike historical RNNs. Also, we do not use GPUs to speed up the training phase of RNNs to be fair to other ML algorithms that cannot exploit GPUs. We use CPUs to parallelize the test phase of all the ML algorithms barring RNN-Synth for the aforementioned reasons. The computational complexity of RNNs is discussed in Online Appendix A.2.2, and more information can be found in Horne and Hush [17] and Zhang et al. [53].

4.3 Reinforcement Learning

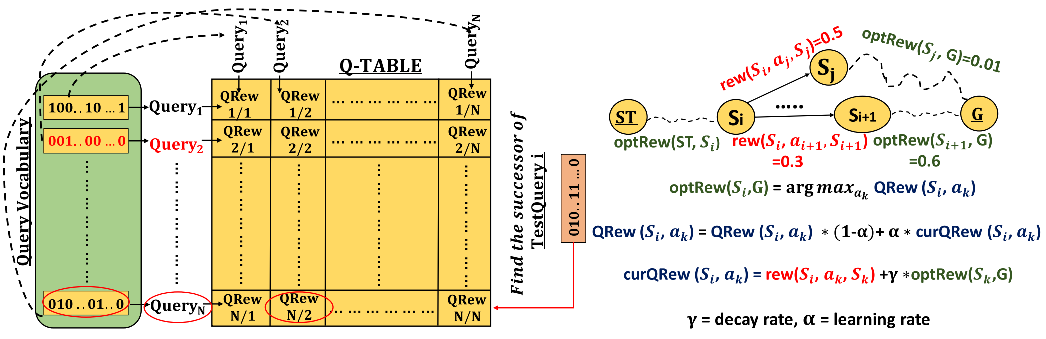
We adapt the Exact Q-Learning algorithm as the reinforcement learning paradigm for next query prediction. While we direct the reader to Russell and Norvig [43] and Watkins and Dayan [50] for complete details of the algorithm, here we describe how we adapt the algorithm to predict the next query embedding in a user interaction session. Exact Q-Learning uses MDPs to capture the temporal dependencies among various *state*s and materializes the long-term *reward*s (or penalties), also called as *Q-values*, for all possible state transitions within a Q-Table. While the rows of a Q-Table indicate all possible states, the columns in it represent the entire vocabulary of *action*s. At each state, the action that an *agent* takes transitions it to a different state and also fetches an instantaneous reward or a penalty from which its long-term reward is estimated. A long-term reward or a Q-value indicates the sum of the instantaneous reward that the agent gets for taking the action and the look-ahead reward that it gets for the optimal sequence of actions thereafter until the goal state. Given a *start state* and a *goal state*, Q-Learning can find the optimal sequence of <state,action> pairs (also called as policy) yielding the highest reward for the agent. Q-Learning is useful for query fragment prediction for the following reasons.

(1) Although queries in an OLAP session follow a sequence, they have a significant difference from traditional time-series applications. User interaction sessions against the database are goal-oriented, and the queries are progressively steered toward interesting insights drawn by the user at the end of the session.
(2) The query prediction algorithm needs to penalize intermediate queries that steer away from the eventual goal query and reward those queries that quickly take a user toward her goal.
(3) In contrast to applications such as games that have a fixed goal, each user session can have a distinct goal query that makes the problem of query prediction significantly harder.
(4) Q-Learning can capture long-term, look-ahead rewards in the form of Q-values that can estimate the cumulative effect of a transition from the current query to the next query when these queries are represented as states in the Q-table.

Recent works such as Krishnan et al. [24], Li et al. [27, 28], and Liang et al. [29] use deep reinforcement learning for various database applications such as distributed stream processing, view materialization, join order optimization, and database tuning. In contrast to exact Q-Learning, deep reinforcement learning can only approximately learn the Q-values while avoiding the materialization of Q-table. In this work, we validate the applicability of the exact Q-Learning algorithm in its most fundamental form to next query prediction as this gives an insight into the basic functionality of the algorithm and how it optimally predicts the top-K next query candidates. Also, we empirically show that for both the Course Website and Bus Tracker datasets, the Q-tables materialized are small enough to be memory resident. For larger workloads than our datasets, the Q-table can eventually be flushed to the secondary storage.

Our adaptation of exact Q-Learning for next query prediction has the following components:

- *Learning agent*: This is our next query predictor modeled as an RL agent that learns the Q-table and predicts the next query.
- *State/action space*: The set of distinct embeddings of the training queries represents the state/action space. Note that both the state and the action in the Q-table is a distinct training query as shown in Figure 8(a). Query execution is the action performed and the state reached corresponds to the query from the next timestep, and hence, we identify both the state and the action space by the set of queries. Q-Learning can also support stochastic actions in uncertain environments that require Partially Observable Markov Decision Processes.



(a) Modeling Q-Table for Next Query Prediction (b) Illustration of Bellman Update

Fig. 8. Exact Q-learning for next query prediction.

MDPs are sufficient for query prediction, because the actions here are deterministic. The environment is certain about the next SQL query (next state) that the agent goes to when it makes a specific transition from the current SQL query (current state).

- *Environment*: This hosts the reward function that captures the ideal temporal sequence of queries.
- *Reward*: This is issued by the environment as a response to an agent action to reflect whether the the next query that is being chosen by the agent indeed succeeds the current query. We support two types of reward functions—Boolean and Numeric—that will be detailed.

During the training phase, the RL agent populates the Q-table based on the temporal sequence of queries from the training sessions. In sustenance experiments, we build the Q-table based on the training sessions, whereas in the case of singularity, the Q-table is updated episodically based on the incoming batch of test-then-train queries. As we can notice from Figure 8(a), the set of distinct queries forms the query vocabulary. The Q-table that the RL agent builds is a square matrix and has both its rows and columns pointing to the queries in the vocabulary. $\text{QRew}_{i/j}$ represents the Q-value, which is the look-ahead long-term cumulative reward that the RL agent gets upon executing Query_j after Query_i. Note that the Q-values are refined over time as the RL agent gets exposed to more training sequences. Each time an RL agent sees Query_j following Query_i, it updates $\text{QRew}_{i/j}$ based on an equation called Bellman update (see Figure 8(b)). Since we only keep track of distinct queries, the number of possible <Query_i, Query_j> query pairs is also limited that keeps the size of the Q-table bounded as well. However, it is not feasible to encounter all possible pairs from the Q-table within the training data. Also, we may not encounter each pair enough number of times during training. This is because if we were to rely only on the temporal pairs from the training data, then the Q-table would be very sparse. To make the Q-table dense and to enhance the effect of the pairs seen during training, we apply a combination of tabular variants of two techniques from the Q-Learning literature called experience replay and random action exploration.

4.3.1 Tabular Variant of Experience Replay and Random Action Exploration. To improve the stability of training by learning the Q-values effectively, two prominent techniques called Experience Replay [44] and ϵ-greedy random action exploration [35, 51] are available in the RL literature. Experience replay periodically re-trains the RL agent upon the training experiences (<Query_i, Query_j> query pairs in our case) it has encountered before. ϵ-greedy random action exploration ensures that the RL agent is also exposed to transitions it is likely to miss during training. Therefore, for each state, instead of always training on the actions with the highest Q-value, it also picks random actions to train upon, with a small (ϵ) probability. While both these techniques are frequently applied in Deep Q-Learning scenarios, we apply their tabular variants to our Exact Q-Learning implementation. Deep Q-Learning uses a separate replay memory to store the training transitions, whereas we already capture all possible transitions in a Q-table that makes the application of these techniques easy and lightweight in our case. At the end of each training episode (online test-then-train or *singularity* evaluation) or after offline training (*sustenance*), the RL agent randomly samples several <state,action> pairs from the vocabulary and populates their corresponding Q-values. This is equivalent to picking a random pair of distinct queries, checking whether they succeed one another or not, and updating their Q-values based on the reward function. If the queries in a sampled pair succeed each other, then it is equivalent to applying experience replay, as we would have encountered that pair during training. Instead, if the queries in such a pair do not succeed each other, then it is equivalent to applying random action exploration as that pair would not have been seen during training. Our ϵ value is 0.5 as we equally bias toward seen and unseen pairs from the training set of sessions.

These techniques reduce the sparsity in the Q-table and also improve the accuracy of Q-values. Let us consider an example pair of successive queries $<Q_i, Q_j>$ (i.e., j=i+1) and assume that the current Q-value($<Q_i, Q_j>$) is 1.0. If we randomly pick that pair again during experience replay, as per the Bellman update equation, then the updated Q-value($<Q_i, Q_j>$) will be $1.0 \times \alpha$ + $(1\text{-}\alpha) \times$ (instantaneous reward + $\gamma \times \max_k$ Q-value($<Q_j, Q_k>$)). Substituting 0.5 for both learning rate α and discount rate γ, and 1.0 for both the instantaneous reward and $\max_k$ Q-value($<Q_j, Q_k>$), the updated Q-value($<Q_i, Q_j>$) would be 1.25. Instead, if we pick a pair of queries that do not succeed each other (random action exploration where j≠i+1) with an instantaneous reward and current Q-value($<Q_i, Q_j>$) both of which are 0.0, then the updated Q-value($<Q_i, Q_j>$) would be $0.0 \times \alpha$ + $(1\text{-}\alpha) \times$ (instantaneous reward + $\gamma \times \max_k$ Q-value($<Q_j, Q_k>$)). Making the same substitution for $\max_k$ Q-value($<Q_j, Q_k>$) that is 1.0, we would get an updated Q-value($<Q_i, Q_j>$) of 0.25. This shows that whether the queries within the randomly sampled query pair succeed each other or not and regardless of whether or not this pair was encountered during the training phase, there will always be an update to the Q-value at the current state if the Q-value at the next state is non-zero. This is because of the recursive definition of Q-values in the Bellman equation that makes the Q-value at the current state dependent on the Q-value at the next state. Reinforcement learning inherently captures the sequence information via a Q-table, and therefore, any additional observations of already observed query sequences will further reinforce the Q-values.

4.3.2 Prediction (Test) Phase. Given a test query (TestQuery i) during the prediction (test) phase as shown in Figure 8(a), to find its successor, the RL agent first checks if the query embedding for TestQuery i exists in the vocabulary of distinct queries. Note that for such comparisons, we use SHA-based optimizations from Section 4.1.3. If we do not find a matching query embedding, then we compute the Cosine similarity of the test query embedding with the embeddings of all the distinct queries in the vocabulary to find the most similar query. Once such a query is found in the vocabulary that can act as a proxy to the test query, we find its top-K next queries with the highest Q-values and return them as the successors to the test query. In the example shown in Figure 8(a), Query_N from the vocabulary is most similar to TestQuery i. Within the row for Query_N, the highest Q-value is for a transition to Query_2 that is returned as the top-1 successor of TestQuery i. For higher values of K, we use a max-heap to return the top-K successors. Similarly to Figure 5(a), we use inter-query parallelism to partition the test queries among several processes. Although we can also allow for nested parallelism by partitioning the distinct queries for Cosine similarity computation (intra-query parallelism), we found that inter-query parallelism is optimized enough and finds top-K queries with minimal latency.

4.3.3 Reward Function. The reward function reflects the temporality among queries at each timestep during a query session. Let us assume that the current user query is Query_i at timestep τ_i, and the RL agent, i.e., the next query predictor predicts a query $\text{Query}_{i+1}^{pred}$ for timestep τ_{i+1}. As mentioned, we support two types of reward functions. For the Boolean reward function, if the actual user query $\text{Query}_{i+1}^{user}$, which is the ground truth, matches the predicted query $\text{Query}_{i+1}^{pred}$, then the RL agent gets a reward of 1, else it gets a reward of 0. For Numerical reward function, we return the Cosine similarity between the embeddings of $\text{Query}_{i+1}^{user}$ and $\text{Query}_{i+1}^{pred}$ as the reward to the RL agent. Based on this function, we can understand that the instantaneous reward is a value between 0 and 1. However, this is not true for the Q-value. This is because Q-values are cumulative look-ahead rewards from a given state until the goal state. Q-values cannot be normalized, because their absolute values need to be compared among several columns to predict the top-K next queries.

4.3.4 Setting Learning Rate and Discount Factor. As shown in Figure 8(b), while navigating from the start state denoting the first query in a user session until the goal state (G) that represents the last query in the session, at any given intermediate state S_i, the RL agent chooses an action (query) that yields it the maximum cumulative future reward. There is an exploration vs. exploitation tradeoff that we can notice here. The quality of an RL agent depends on the accuracy of the Q-values that get refined with more learning episodes. We balance the exploration vs. exploitation tradeoff by setting the parameters in the Bellman update [43] (Equation (1)) used for Q-Learning,

$$\text{QRew}_{i/j} = \text{QRew}_{i/j}(1-\alpha) + \alpha * [\text{rew}_{i/j} + \gamma * \text{optRew}(S_j, G)], \tag{1}$$

where α refers to the learning rate and γ refers to the decay rate, also called the discount factor, both of which lie between 0 and 1. If the RL agent chooses Query_j to be executed after Query_i, then this decision fetches it an instantaneous reward that is summed to a discounted look-ahead optimal reward obtained from the remaining queries until the termination of the session (goal state) as illustrated in Figure 8(b). Note that if γ is set to 0, then the RL agent chooses to execute a query corresponding to S_j after S_i, instead of S_{i+1}, based on the instantaneous rewards ($0.5 > 0.3$ from the figure). Instead, if γ is set to 0.5, then we choose S_{i+1}, which yields a $\text{curQRew}(S_i, a_{i+1}) = 0.6$, which is greater than $\text{curQRew}(S_i, a_j) = 0.505$. This allows it to capture the effect of cumulative reward. However, a higher γ than this will not only take a longer time to train but will also prefer longer paths to the goal state. On similar lines, the value of α determines the extent to which the value of $\text{QRew}_{i/j}$ needs to be updated each time we encounter Query_j (corresponding to state S_j in Figure 8(b)) after Query_i that corresponds to S_i in the figure. A higher value of α biases the Q-value more toward the more recent executions of Query_j after Query_i, thus asking for more update steps, whereas a lower α does not refine the Q-table. Hence, we set both γ and α to 0.5, which allows the RL agent to explore just enough to reach an acceptable convergence to the right Q-values within the Q-table and thereby make accurate action predictions (optimal exploitation) at a given state. We present the computational complexity of Exact Q-Learning in Online Appendix A.2.3.

4.4 Parameter Settings

In this section, we discuss the parameter values that we set for each algorithm.
(a). Generic Parameters: There are a few parameters that are generic across all the ML algorithms.

- Episode Size - We set the #queries in a test-then-train singularity episode to 1,000. This was set to keep the episode wide enough to get statistically significant test metric observations.
- Top-K - We set K to 3 across all the ML algorithms. This was empirically set to ensure that even the worst performing ML algorithm does not take longer than an hour

per test-then-train episode of 1,000 queries (as mentioned above) to predict the successor queries.

- Degree of Parallelism - We set the parallelism to 48 processes. This was set based on the 12 CPUs on our server that are hyper-threading enabled and allow for 4 virtual CPUs per core. Also, note that we allow two modes of parallelism: single level of parallelism (inter-query) and nested parallelism with both inter and intra query parallelism. In the former case, we spawn 48 processes, whereas in the latter, we spawn 3 threads at the outer level each of which in turn spawns 16 processes. So at any given moment, we do not have more than 48 processes running in parallel. The #threads does not influence CPU time as they are I/O-bound and the test phase of all the ML algorithms is CPU-bound.
- Sampling Rate - In the case of Cosine similarity-based CF, the model learned is a set of session summaries. Therefore, to ensure that during the test phase, the top-K next queries are picked from training sessions that are similar to the ongoing test session, we follow two level sampling. We sample 1% of the training sessions at the outer level and we sample three queries per session at the inner level. These parameters are empirically set to ensure that CF techniques do not consume exceptionally long test times. Likewise, for NMF-SVD-based CF, the model is a matrix whose rows represent training sessions. Here we sample 1% of the training session summaries to facilitate quick comparison of an ongoing test session with the historical sessions. For RNNs and Q-Learning, the trained models and the test phase are not directly dependent on the session structure. Therefore, we randomly sample 10% of the distinct queries from the training set for historical RNNs and Q-Learning. Synthesis-based RNNs do not require sampling as their test phase does not depend on historical queries.

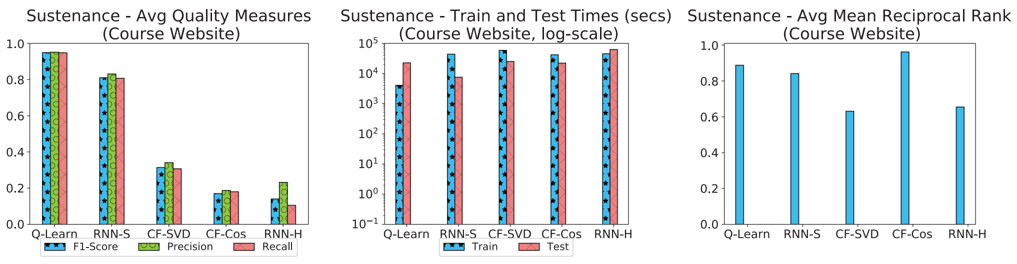
(b). Cosine similarity-based CF: We use inter-query parallelism with 48 processes to parallelize the test phase of CF as it is more effective than nested parallelism.
(c). NMF-SVD-based CF: We set the number of latent dimensions during matrix factorization to 10% of the number of distinct queries, however not exceeding 100. The 10% limit is imposed when the number of distinct queries is lesser than 100 during the initial singularity episodes. Other parameters include those we set for non-negative matrix factorization from the scikit-learn library. We use "nndsvdar" as the random value initialization procedure for sparse matrices. Likewise, we use multiplicative update solver that is more efficient and faster than coordinate descent. Remaining NMF parameters are set to default values. We use inter-query parallelism with 48 processes to parallelize the test phase of CF as it is more effective than nested parallelism.
(d). Recurrent Neural Networks: We run experiments with all the three variants of RNNs: vanilla RNNs using simple backpropagation, Long Short Term Memory (LSTM), and Gated Recurrent Units (GRU). For all these three variants, we use a single hidden layer containing 256 hidden nodes. We use 40 epochs during RNN training and dropout regularization that turns off 50% of the hidden nodes during training. As mentioned, while we use 3 threads each spawning 16 processes for historical RNNs, we use single-threaded implementation for synthesis-based RNNs. Other neural network specific parameters such as activation functions were discussed in Section 4.2.
(e). Q-Learning: As mentioned, we use 0.5 for both learning rate (α) and decay rate or discount factor (γ). We sample 100 random pairs of queries from the distinct query vocabulary for experience replay and random action exploration. We use inter-query parallelism with 48 processes to parallelize the test phase of Q-Learning, as it is more effective than nested parallelism.

5 EXPERIMENTS

All our experiments were conducted on a machine running Ubuntu 16.04 OS, with a 12-core 3.0-GHz Xeon E5-2687WV4 processor, 120-GB RAM, and 4.0-TB hard disk. Hyper-threading is

Table 2. Sustenance: Train and Test Sessions and Query Splits

Dataset	#Train Sessions	#Train Queries	#Test Sessions	#Test Queries
Course Website	35,115	91,385	8,778	23,222
Bus Tracker	4,512	17,430	1,128	4,676



(a) F1-Score, Precision & Recall (b) Train and Test Times (Secs) (c) Mean Reciprocal Rank

Fig. 9. Sustenance experiments (course website): Quality and time measures (80% train, 20% test).

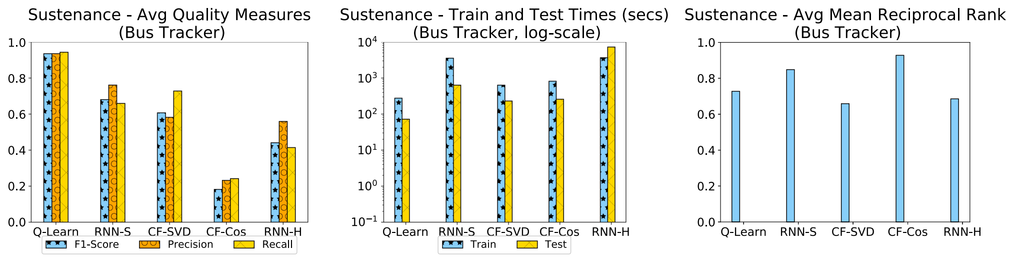
enabled with four virtual CPUs per core that results in 48 CPUs in total. We implemented embedding creation in Java using JSQLParser and the next query prediction framework in Python because of the extensive library support that exists for Python from scikit (for NMF-SVD-based CF) and Keras/TensorFlow (for RNNs). Cosine similarity-based CF and Q-Learning were implemented from scratch in Python. We plan to make the source code available on https://github.com/DataSystemsLab/SQLPredictor in the near future.

Our experimental section is organized as follows: First, we discuss the results of sustenance (80% train, 20% test) experiments followed by those from singularity (incremental test-then-train in a streaming scenario) experiments over both the Course Website and the Bus Tracker datasets. Upon each dataset, we present the operator-wise breakdown of the sustenance results. For details about sustenance and singularity evaluation methods, refer to Section 3.3.

5.1 Results of Sustenance Evaluation

As discussed in Section 3.3, we include this set of experiments under the sustenance category, because we train each of the compared ML algorithms offline on 80% of the total sessions. Upon undergoing such extensive training, we evaluate if the learned ML models can demonstrate sustained high-quality performance over a held-out test set of 20% sessions. As mentioned in Section 3.1.1, there are 43,893 clean sessions consisting of 114,607 queries in the Course Website dataset and 5,640 clean sessions comprising 22,106 queries in the Bus Tracker dataset after applying the session cleaning heuristics. Note that in Table 2 presenting the number of test and train sessions as well as queries, the session splits maintain the 80% train, 20% test proportion whereas the query splits need not exactly maintain that ratio as different sessions may contain variable #queries. From the test query count presented in the table, we need to discount the last query in each session, because the successor is not predicted for it. Therefore, the number of test queries for which the next query is predicted is 14,444 for Course Website and 3,548 for Bus Tracker.

5.1.1 Quality and Latency Metrics. Figures 9 and 10 present the average test prediction quality and latency of all the ML algorithms on the Course Website and Bus Tracker datasets, respectively. We plot the algorithms with the following abbreviated names in the same order in each figure: Q-Learn (for Q-Learning), RNN-S, CF-SVD (for NMF-SVD-based CF), Cosine similarity-based CF



(a) F1, Precision, Recall & Accuracy (b) Train and Test Times (Secs) (c) Mean Reciprocal Rank

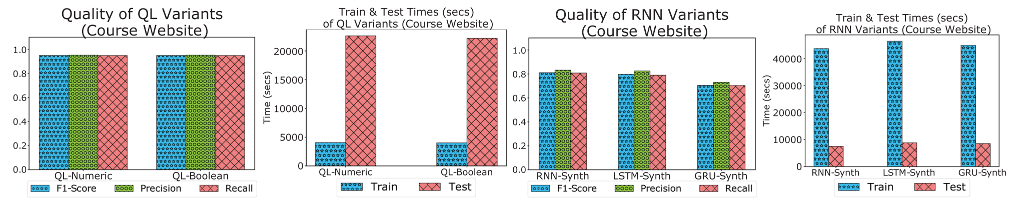
Fig. 10. Sustenance experiments (BusTracker): Quality and time measures (80% train, 20% test).

(CF-Cos), and RNN-H. We sorted the algorithms based on their decreasing order of performance in a majority of experiments. The primary takeaways from the F1-scores in Figures 9(a) and 10(a) are as follows: Exact Q-Learning consistently outperforms all other ML algorithms closely followed by synthesis-based RNNs, thus showing the effectiveness of using temporal predictors for next query prediction task over using query recommender systems.

CF-Cos consistently performs poorly, as it picks the next queries totally from the sampled training data. CF-SVD, however, relies on sampling only to identify similar training sessions to the ongoing test session. It picks the next query from the completed matrix that can assign a similarity score even to those queries that have very few historical occurrences thereby handling sparsity. CF-SVD captures the essence of session similarity more effectively than CF-Cos instead of using direct Cosine similarity score computed between sessions. Even then, CF-SVD falls short of temporal predictors while performing better only on the Bus Tracker dataset as compared to the Course Website. This is because Bus Tracker has a fairly small amount of possible next query pairs, i.e., 625 from 25 distinct query embeddings learned during training, as compared to Course Website that has 501,264 possible pairs generated from 708 distinct trained query embeddings. This makes Bus Tracker an easier dataset even for recommender systems that are temporality agnostic.

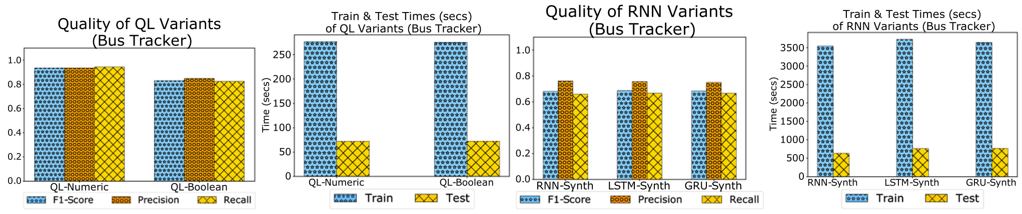
Similarly, RNN-Synth consistently outperforms historical RNN thereby demonstrating the power of synthesizing novel next query embeddings over picking a historical query that has the least entropy with the predicted output vector. This is achieved by synthesis because of its ability to identify the fragments that occur in the next query based on the probabilities of various dimensions in the output vector emitted by the RNN. Instead, RNN-H relies on least entropy heuristic that often picks historical queries that have as few bits set as possible. This is because entropy is least when only those dimensions are set that have the highest confidence. Therefore, RNN-H can predict the DML type of the next query accurately as it tends to be "SELECT" for most of the workload. The most surprising result comes from Q-Learning that achieves a test F1-score of 0.95 on Course Website and 0.98 on Bus Tracker. This is plausible because of Exact Q-Learning that accurately learns the rewards and penalties for all possible pairs of <current query, next query> temporal sequences that occur during the training phase, into a Q-table. By default, we use the numerical reward function during the training phase for Exact Q-Learning that rewards partially overlapping predictions of next queries over totally penalizing them.

Besides F1-score, we also measure the average Mean Reciprocal Rank (MRR), which is defined as $\frac{1}{\text{Rank of the Most Matching Query}}$ to identify the effectiveness of the top-K next query candidates predicted by each ML algorithm. The MRR of RNN-Synth is ≥0.8 on both the datasets (see Figures 9(c) and 10(c)). This means that from the top-3 results, the topmost result has the highest similarity with the ground truth. Likewise, the MRR of Q-Learning is high on Course Website, although it drops to 0.7 on Bus Tracker. The MRR of CF-SVD and RNN-H are low on both the datasets.



(a) QL Variants (Quality) (b) QL Variants (Time) (c) RNN Variants (Quality) (d) RNN Variants (Time)

Fig. 11. Sustenance experiments (course website): Variants of Q-learning and RNN.



(a) QL Variants (Quality) (b) QL Variants (Time) (c) RNN Variants (Quality) (d) RNN Variants (Time)

Fig. 12. Sustenance experiments (BusTracker): Variants of Q-learning and RNN.

Surprisingly, CF-Cos achieves a near 1.0 average MRR on both the datasets. Upon examining its predicted next queries, we found that in several cases, all the predicted next queries are equally bad, and by default CF-Cos picks the first query from top-3 for the computation of test F1-score. Thus, it achieves the highest MRR purely from serendipity. The latency results in Figures 9(b) and 10(b) contain some interesting patterns. The most interesting result is that RNN-Synth, despite having a single-threaded prediction phase, manages to achieve the best test latency on Course Website that is the larger dataset. This observation does not hold on Bus Tracker, because it has relatively low-dimensional embeddings, and other approaches outperform RNN-Synth during test phase. Overall, Q-Learning achieves the least cumulative train & test time on both the datasets and emerges the overall winner on both quality and latency. Note that we use the same degree of test phase parallelism for all the approaches barring RNN-Synth, and the training phase remains unparallelized for all of them.

We compare the effectiveness of the Numeric and Boolean reward functions that we use to train Q-Learning. While there is not much difference between both the QL variants in the case of Course Website (Figure 11(a)), there is a noticeable difference in the test F1-scores upon the Bus Tracker dataset (Figure 12(a)). This can be attributed to the fact that in the Course Website dataset, there are more distinct embeddings, and hence there are more unambiguous distinct sequences of <current query, next query> pairs in the corresponding query logs. In contrast, the small number of distinct embeddings in the Bus Tracker dataset indicates that a particular query embedding is often followed by different next queries in different sessions. For example Query_i may be followed by Query_j in a particular training session, whereas in a different training session Query_k may be its successor. Capturing all such possible sequences unambiguously is easier using a Numeric reward function as it computes the reward based on the Cosine similarity between a predicted next query and the ideal successor during the training phase. A Boolean reward function is comparatively more rigid and less flexible in capturing all these nuances. Another striking observation is that Q-Learning consumes a higher test time than train time for Course Website (Figure 11(a)), whereas the training time is higher for Bus Tracker (Figure 12(a)). This is because the training phase of Q-Learning builds the Q-Table, and the time consumed in populating the Q-Table is only dependent on #queries in the training sessions. However, the test time is not only dependent on the number of test queries but also on whether or not these test queries are always found in the pre-built Q-Table. When the query vocabulary is small, more often than not, every test query is found in the Q-Table and all that needs to be done to find a successor query is to perform a constant time look up on the row corresponding to the matching query. This is possible in the case of Bus Tracker but not for Course Website. Therefore, the test time is longer for the Course Website dataset, because for each test query that is not found in the Q-Table vocabulary, a comparison needs to be done with the entire query vocabulary to find the closest match. Also, the Cosine similarity computation takes longer for Course Website queries due to their high-dimensional embeddings compared to Bus Tracker.

As described in Section 4.4, we experiment with three flavors of synthesis-based RNNs. While the vanilla variant of RNN-Synth uses standard backpropagation, LSTMs and GRUs use additional memory units to capture the long range dependencies in sequence prediction. An example to such a dependency in natural language sentences can be a noun (could be a male/female/neutral gender) and a pronoun (can be he/she/it) that occur at different parts in a sentence, yet semantically connected. In next query prediction using RNNs, such dependencies translate to a particular fragment in the initial dimensions of a user query being semantically connected with another fragment that is present at a later dimension within the query embedding. Such long range dependencies may exist, but we did not notice any benefit in using LSTMs or GRUs over RNNs. As shown in Figure 11(c) and (d), upon the Course Website dataset, LSTMs and GRUs consume slightly more time to train larger models with additional parameters and logic gates than standard RNN variants, while also proving to be worse on test F1-scores. A consistent behavior is also observed on the Bus Tracker dataset (see Figure 12(c) and (d)), although the quality of LSTMs and GRUs is not worse than that of RNN-Synth unlike the case of Course Website dataset. To examine the effect of training LSTMs on query sequences instead of <Current Query, Next Query> pairs, please refer to Section 5.3.4.

5.1.2 Memory Consumption. All the ML algorithms that we adapted for next query prediction load the entire set of query embeddings into main memory. Thus, all other data structures that they create only use references to the query embeddings without having to replicate them. As mentioned, the Course Website dataset contains 114,607 queries each of which has a 102,020 bit long embedding. Likewise, the Bus Tracker dataset has 22,106 queries whose embeddings are 9,155 bits long. We use bit vectors from the PyPi library (https://pypi.org/project/BitVector/) to represent the embeddings that consume 1.46 GB of main memory for Course Website and 25.29 MB for Bus Tracker. Other in-memory auxiliary data structures include <sessionID,QueryID> references to the query embeddings. The cumulative base memory usage for in-memory embeddings including such data structures is 3.1 GB for Course Website and 414.39 MB for Bus Tracker.

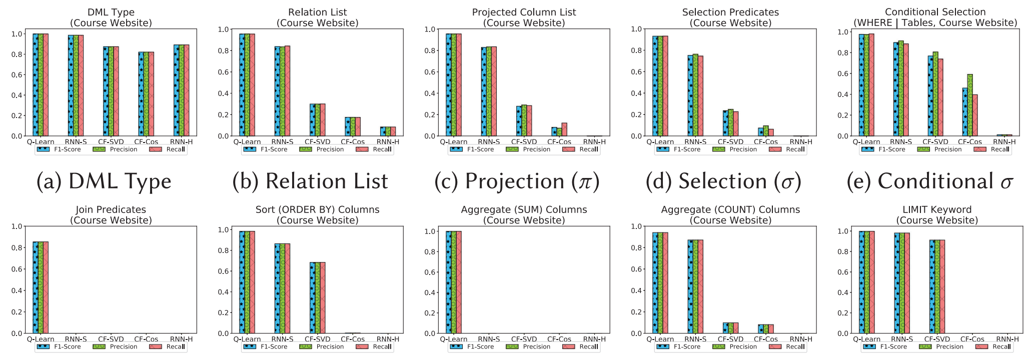
Table 3 lists the individual memory overhead that comes from the data structures created by each of the ML algorithms for next query prediction. This is in addition to the base memory required by the query embeddings as listed above. As we have described in Section 4, each ML algorithm generates distinct queries from the training set. The collection of distinct queries is also referred to as query vocabulary. Similarly, we also create session dictionaries at train time that are <key, value> pairs of the sessionID and the last queryID from that session. This is essential for all the algorithms to know that there is no need to predict the next query for the last query in a session.

As we can notice from Table 3, Q-Learning consumes the least amount of memory, because its Q-Tables are not that huge. This is because of the number of distinct queries that regulate the size of the table. There are 708 distinct query embeddings for Course Website and 25 for Bus Tracker, thereby leading to 501,264 cells and 625 cells respectively in the Q-Tables. On similar lines,

Table 3. Sustenance Experiments: Memory (MB) Overhead for ML Approaches

Approach	Data Structure	Memory (MB) (Breakdown) Course Website	Memory (MB) (Total) Course Website	Memory (MB) (Breakdown) BusTracker	Memory (MB) (Total) BusTracker
Q-Learn	Query Vocabulary	0.19	20.88	0.008	0.34
	Session Dictionaries	3.989		0.305	
	Q-Table	16.7		0.028	
CF-Cos	Query Vocabulary	0.19	21.57	0.008	2.07
	Session Dictionaries	3.989		0.305	
	Session Samples	11.153		1.164	
	Session Summaries	6.236		0.593	
CF-SVD	Query Vocabulary	0.19	830.2	0.008	4.77
	Session Dictionaries	3.989		0.305	
	Session Summary Samples	0.062		0.006	
	Sorted Session Keys	1.139		0.149	
	Matrix Factorized	824.82		4.3	
RNN-H	Query Vocabulary	0.19	423.59	0.008	142.35
	Session Dictionaries	3.989		0.305	
	Sampled History	0.152		0.007	
	Model	419.26		142.03	
RNN-S	Query Vocabulary	0.19	425.86	0.008	143.15
	Session Dictionaries	3.989		0.305	
	Schema Dictionaries	2.42		0.809	
	Model	419.26		142.03	
GRU-S	Query Vocabulary	0.19	844.8	0.008	285.4
	Session Dictionaries	3.989		0.305	
	Schema Dictionaries	2.42		0.809	
	Model	838.2		284.28	
LSTM-S	Query Vocabulary	0.19	1054.28	0.008	356.52
	Session Dictionaries	3.989		0.305	
	Schema Dictionaries	2.42		0.809	
	Model	1047.68		355.4	

Cosine similarity-based CF only stores the bit vector summaries for 43,893 and 5,640 sessions from both the datasets. Note that the session samples require more memory than the summaries for all the sessions. This is because the session samples for CF-Cos also contain the sampled queries per session (recall the two level sampling we described in Sections 4.1.1 and 4.4). In the case of CF-SVD, we can notice a significant difference in the total memory consumption between Course Website and Bus Tracker. This is because of the difference in the dimensionality of the matrix that is factorized for each dataset. As mentioned in Section 4.1.2, the rows in the matrix represent sessions while the columns represent the query vocabulary. Both the #sessions and #distinct queries in Course Website are larger than those from Bus Tracker. Another interesting observation here is that the size of the matrix for CF-SVD is much larger than the size of the Q-Table created by Q-Learning. Although the #columns for both Q-Table and CF-SVD matrix are same as the number of distinct queries, the difference lies in the #rows. The #sessions (rows in CF-SVD matrix) is much higher than #distinct queries (rows in Q-Table) in both the datasets.



(a) DML Type (b) Relation List (c) Projection (π) (d) Selection (σ) (e) Conditional σ

(f) Join ($\bowtie$) (g) Sort (ORDER BY) (h) Aggr (SUM) (i) Aggr (COUNT) (j) LIMIT

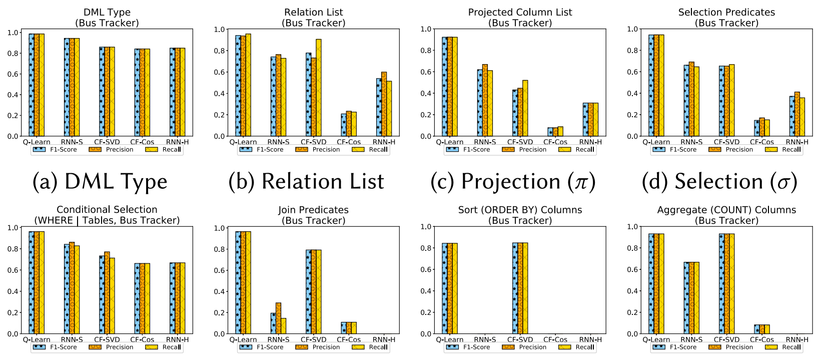
Fig. 13. Course website: Fragmentwise next query prediction quality measures.

Last, among the several variants of an RNN, historical RNN that predicts from query logs has a comparable, albeit slightly lesser memory consumption than its synthesis counterpart, RNN-S. This is because the Keras RNN models learned by both these adaptations consume the same memory but the difference is in the auxiliary data structures they create. While RNN-H (historical RNN) contains references to the sampled queries from which it picks the top-K next queries, RNN-S materializes a bi-directional schema dictionary in memory using that it fixes the SQL violations in the synthesized top-K next queries. However, it is important to note that the memory overhead for such schema dictionary is as low as 2.42 MB for Course Website and 0.8 MB for the Bus Tracker dataset, for the constant time fixes and synthesis that it supports. We also included the memory consumed by the synthesis-based variants of RNN such as GRU and LSTM. Note that the Keras/TensorFlow model generated for GRU-S is larger than that of RNN-S while the model for LSTM-S is the largest. The reason for this is that GRU uses a forget gate that is an additional component not existing in vanilla RNN, to capture long range dependencies among the input query fragments. Although GRU models consume more memory, such expense does not translate into higher prediction quality as we have observed in Section 5.1.1. LSTM is an advanced variant that uses more gates than a GRU does (a forget gate is replaced with an update gate and a reset gate) and has more parameters that makes its models even larger.

5.1.3 Fragmentwise Quality Breakdown. To understand how the predictive ability of the ML algorithms w.r.t. each fragment influences the overall next query prediction as a whole, we present a fragmentwise breakdown of the prediction quality in Figures 13 and 14. Note that the training and testing is still done on the query as a whole and not on individual fragments, as this gives a clear idea of how important each fragment is in predicting the overall query accurately.

As we can notice in Figures 13(a) and 14(a), almost all the algorithms perform well with a test F1-score of 0.8 or above in predicting the DML type (SELECT / INSERT / UPDATE / DELETE) of the next query. This is because up to 89% (Course Website) and 86% (Bus Tracker) of the queries are of SELECT type (see Table 1(a)). This bias thus enables even the poorly performing algorithms such as CF-Cos and RNN-H to have such a high predictive capacity on DML type. Note that the test F1-scores upon relation list, projection list and selection predicates maintain the same relative ordering among the various ML algorithms as they do on the entire query prediction as a whole. We term such query fragments as Bellwether[1] fragments, meaning that the predictive behavior

[1]This is a frequently used term in election outcome analysis to identify those Bellwether constituencies that locally obtain a victory pattern among the political parties that is the same as the general outcome trending across the entire nation.



(a) DML Type (b) Relation List (c) Projection (π) (d) Selection (σ)

(e) Conditional σ (f) Join ($\bowtie$) (g) Sort (ORDER BY) (h) Aggr (COUNT)

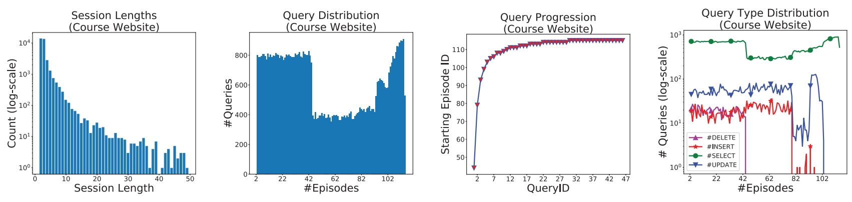
Fig. 14. Bus tracker: Fragmentwise next query prediction quality measures.

of an ML algorithm w.r.t. them is indicative of its overall predictive ability. This happens because every query has a relation list and at least 95% of the queries contain selection predicates and projection lists (Table 1(a)). So these fragments have a huge weightage in predicting the next query. Conditional selection in Figures 13(e) and 14(e) evaluates how well an ML approach can predict the selection predicates given that the relation list was accurately predicted. This is studied to understand how correlated the predictions are upon relation list and selection predicates. Regardless of how poorly an ML approach predicts the relation lists, we notice that in a majority of cases when it gets (at least a subset of) the relation list right, it also manages to predict the corresponding selection predicates accurately. Note that as we discussed in Section 3.2.2, a selection predicate refers to a column participating in a WHERE clause, a comparison operator and an equi-depth constant range bin for the column. This behavior can be attributed to the intrinsic correlation between the relation lists and selection predicates and the fact that up to 95% of the queries contain selection predicates.

In contrast to selection predicates, join predicates are scarce in Course Website (3.66%), and they are of a moderate percentage (25.74%) in the Bus Tracker dataset. Surprisingly, even though all other ML approaches including synthesis-based RNNs fail to capture them, Q-Learning can predict them with a test F1-score of 0.8 on Course Website and almost close to 1.0 on Bus Tracker. This happens because Q-Learning learns the dependencies among queries as a whole using Markov Decision Processes. This means no matter how rare a particular fragment is, as long as it gets the entire next query right, that automatically includes the rare fragments. In stark contrast, RNNs learn the individual weights of each fragment in the next query and not the query per se. This can be ineffective when the inherent occurrence percentages of some fragments is very low. Another important thing to notice is that CF-SVD also performs well in the case of rare fragments, especially for JOIN, ORDER BY, and COUNT in the case of Bus Tracker. This can be explained by the fact that the number of columns in the matrix for Bus Tracker is as low as 25. Hence, finding similar training sessions to the ongoing test session is easy under such low dimensionality, even with sampling. Note that we do not present results for GROUP BY, MAX and conditional join predicates for both the datasets. Likewise, we do not present the results for COUNT and LIMIT for Bus Tracker. This is because all the ML algorithms get a test F1-score of 0.0 on these fragments following their extremely low occurrence percentages.

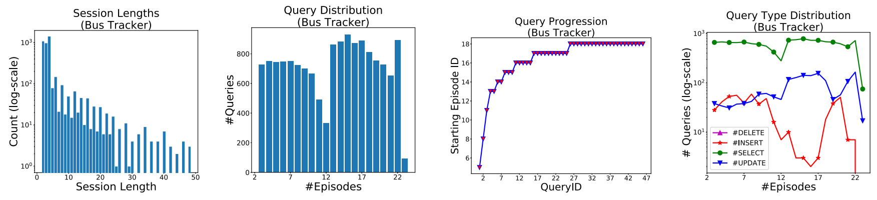
5.2 Results of Singularity Evaluation

Figures 15 and 16 show the properties of the Course Website and Bus Tracker datasets w.r.t. the singularity experiments. We notice that the session length distributions of the clean query logs



(a) Session Lengths (b) Queries (c) Query Progression (d) Query Types

Fig. 15. Singularity (dataset properties, course website): Session and query distribution.

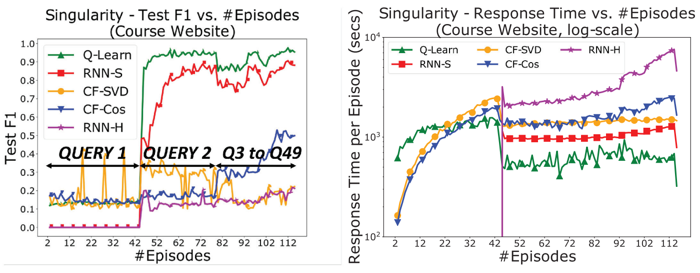


(a) Session Lengths (b) Queries (c) Query Progression (d) Query Types

Fig. 16. Singularity (dataset properties, BusTracker): Session and query distribution.

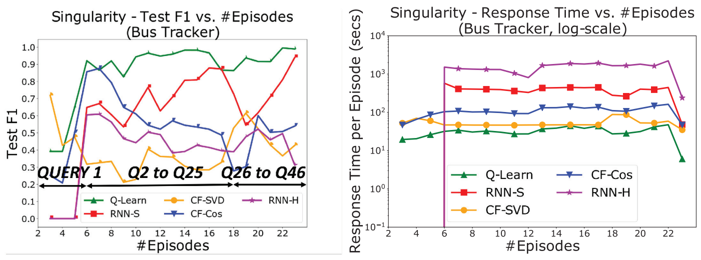
from both the datasets have long tail property with only a few sessions being long and a majority of the sessions being short. With respect to the number of queries that stream in, each episode strictly has 1,000 queries except the last episode that may contain the left over queries. However, in Figures 15(b) and 16(b), we notice that several episodes have fewer than 1,000 queries. This happens because, in each episode, there can be several queries marking the end of the session and such queries are discounted as they do not have a successor. Thus, the query count plotted in these figures only shows the number of queries in each episode for which the ML algorithms predict the next query. The query progression plotted in Figures 15(c) and 16(c) shows that the arrival rate of new queries within the concurrent sessions is low and sparse during the several initial episodes (episode length = 1,000 queries), whereas the queries start changing more frequently toward the later episodes thus indicating that prediction of the next query during the later episodes is harder as compared to prediction during the initial episodes. While all the “DELETE” queries typically occur during the first 40 episodes in the Course Website dataset, insertions and updates become fewer during the later episodes beyond 80. However, the “SELECT” queries approximately maintain the same frequency over all the episodes. While “DELETE” queries do not exist in the Bus Tracker dataset, “UPDATE” queries show some fluctuation and become fewer in the later episodes of concurrent sessions. “SELECT” and “INSERT” queries, however, maintain a reasonably high frequency during each episode all the way until the end.

As we have mentioned in Section 3.3, in singularity experiments, queries keep streaming from concurrent sessions in an episodic manner and the ML algorithms predict the next query for each query that streams in (as long as it is not the last query from a session). Toward the end of the episode, the learned model so far gets updated based on the queries from the episode. The purpose of these experiments is to test if a model can achieve a self-managed behavior or a singular point beyond which it consistently predicts the next queries accurately. Figures 17(a) and 18(a) present the average test F1 scores of various ML algorithms in each episode w.r.t. next query prediction. We can notice that in the case of Course Website, both Q-Learning and RNN-Synth show a monotonically increasing behavior in prediction quality although with some fluctuations toward the later episodes. This is because queries 3 to 49 arrive at a really rapid rate in those



(a) F1-Score (b) Response Time

Fig. 17. Singularity experiments (course website): Quality and time measures.



(a) F1-Score (b) Response Time

Fig. 18. Singularity experiments (BusTracker): Quality and time measures.

later episodes, and it presents a challenge to the algorithms in predicting the next queries. In a general sense, although we cannot say that the algorithms strictly achieve singularity, both these temporal predictors show some consistency in yielding high test F1 scores with more episodes.

Another interesting thing to note here is that until episode 42, all the concurrent sessions stream in their first query. Note that both the RNN variants, RNN-S and RNN-H, consume the training data in pairs of <current query, next query>, and they cannot make any predictions until 42nd episode. This is because, during that phase, there are no such training query pairs to learn a model from, and thus both RNN-H and RNN-S yield a 0 test F1 score. In contrast to these algorithms, Q-Learning and CF variants start producing test F1-scores despite them not being high, right from the first episode. This is because the Q-Table and session summaries start getting populated from the very beginning. Q-values are initialized to 0 until the second query is observed; but a random prediction can still happen. We present the query progression on the quality plots to enable a better understanding of how the test F1 score patterns correlate with the arrival rate of queries. The test F1-scores in the case of Bus Tracker dataset show a fluctuating, erratic behavior because of the small size of the dataset. There is no real convergence point for any of the algorithms except for Q-Learning that manages to again yield consistently high test F1-scores. Note that the test data here changes for each episode unlike the sustenance experiments that have a held-out test set. Another reason for the fluctuating F1-scores is that the Bus Tracker dataset has very few distinct embeddings. This means that the same query embedding may have different succeeding queries across different episodes, thus rendering the sampled sessions or fragment weight learning ineffective for both CF and RNN algorithms respectively. Q-Learning using numeric reward function captures these nuances more effectively and can thus adapt to the shifting successor query patterns in each episode.

Figures 17(b) and 18(b) present the response time in each episode for each ML algorithm. Note that the response time includes the next query prediction latency for all the queries in that episode summed to the re-training time of the model toward the end of the episode. We can observe a strict correlation between the query distribution in Figures 15(b) and 16(b) to the response times on those datasets. For instance, the latencies for Course Website see a decline between episodes 42 and 92 before they rise thereafter. This is because the number of queries that have successors drops in that interval based on Figure 15(b). Likewise, on the Bus Tracker dataset, we notice a decline in latency at episode 12 when the query count decreases (see Figure 16(b)). We can observe that the response time for RNN-H is the worst and keeps growing with more episodes as the sampled history of queries keeps growing. This growth is not that sharp for CF-Cos, as it follows a two-level sampling where session samples gradually increase with time at a higher rate than queries. Still, we can notice a monotonically increasing response time, because the re-training for CF algorithms is cumulative and not incremental with each episode. RNN-Synth and Q-Learning are both capable of incremental training and hence, their response time grows the least. Consistent with our sustenance latency observations, Q-Learning consumes the least latency for singularity experiments as well. This is because its cumulative train and test time only consists of Q-Table construction and Q-Table look-ups both of which can take constant time for a fully materialized Q-Table, especially if the query vocabulary within an episode already exists in the Q-Table. RNN-Synth also requires constant time prediction but it is not parallelized. Even then, its test time is actually lesser even than Q-Learning. The reason RNN-Synth comes next to Q-Learning is its longer train time as compared to Q-Learning. In the case of Bus Tracker, the query embeddings are of low dimensionality and the query vocabulary is small. This results in parallelized test phase of CF algorithms outperforming single-threaded RNN-Synth on latency. We have explained the reasons for not being able to parallelize RNN-Synth in Section 4.2.2. RNN-H still consumes the highest response time also for the Bus Tracker dataset owing to its long training and test latencies.

5.3 Query Re-generation and Result Comparison

In this section, we evaluate the performance of the ML algorithms w.r.t. the execution results of the predicted next queries and how they compare to the expected result set of tuples from executing the ideal successor queries. Among the two datasets, Course Website has underlying data, whereas the Bus Tracker dataset [30] only provides the schema but not the data. Therefore, our experiments on evaluation of query results are confined to the Course Website dataset. We use the same settings as the sustenance experiments by training on 80% of the query sessions and testing on the remaining 20% sessions. Table 4 presents some vital statistics about both the underlying data and the query workload. Among the 113 tables in the Course Website schema, 65 tables have fewer than 10 tuples. We can notice from Table 4 that the number of tables keeps reducing with increasing cardinalities thereby forming a long-tail distribution. Although we did not notice a strong bias, we have observed that the small and medium-sized tables with cardinality lesser than 1K frequently participate in the query workload. Following the train, test splits from Section 5.1, among 23K test queries, there exist 14,444 non-terminating queries that are followed by a successor query in their corresponding user sessions. As mentioned in Table 4(a), 89% of the query workload comprises SELECT queries as it is predominantly OLAP. We observed 12,883 SELECT queries among the 14K non-terminating test queries that we use for query result comparison. A majority of these queries return non-zero results upon execution (see Table 4(b)), although we compute quality metrics on queries returning both zero and non-zero results.

5.3.1 Query Re-generation. To re-generate an SQL query from the predicted SQL fragments, we follow either of the two following approaches: (a) SQL reconstruction or (b) SQL borrowing

Table 4. Table Cardinalities and Query Execution Statistics (Course Website)

(a) Cardinality Frequency

Cardinality	#Tables
0–10	65
11–100	26
101–1000	14
1K–10K	5
10K–100K	2
>100K	1

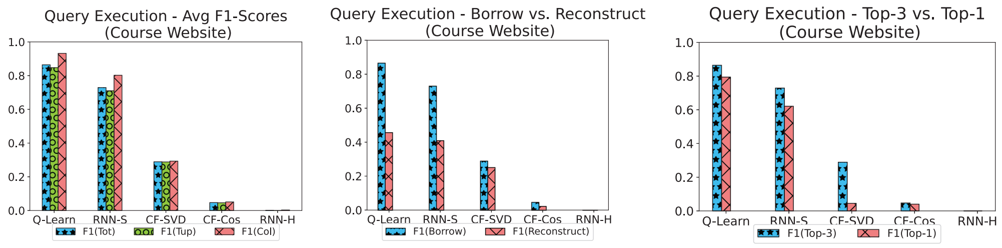
(b) Query Result Distribution (Sustenance)

#Zero Results	#Non-zero Results
3,299	9,584

that borrows a historical SQL query from the training sessions. SQL reconstruction creates an SQL query entirely based on the predicted SQL fragments, whereas the latter approach borrows an SQL query from the training sessions that exactly matches the predicted set of fragments. We use a set of heuristics to explicitly re-construct an SQL query, whereas we create a dictionary of key-value pairs to facilitate SQL borrowing though a constant time lookup. Each distinct SQL query seen during training is stored as an entry in the dictionary—the set of SQL fragments within the query becomes the key and the query itself is stored as the value. Since we can have several training queries containing the same SQL fragment set (key), we randomly pick one of them and store it as the value to make the dictionary memory-efficient. Either SQL reconstruction or borrowing is applicable to collaborative filtering-based recommender systems, historical RNNs and Q-Learning. However, in the case of synthesis-based RNNs, borrowing a training query may not always work, because RNN-Synth can predict SQL fragments that correspond to unseen SQL queries that are absent from the training sessions. Therefore, if we use SQL borrowing for RNN-Synth, then we first check if the set of predicted fragments is in the dictionary key list. If we find a matching key, then we return the corresponding value from the dictionary entry as the predicted SQL query; otherwise, we fall back to query reconstruction. This problem does not arise with other ML approaches as all of them predict SQL fragments entirely from the query vocabulary created from training sessions.

Following are the steps we follow for query reconstruction based on the predicted SQL fragments.

- Each SQL query is created by *stitching* together the predicted SQL fragments in the following order: query (DML) type, projected columns, tables, selection predicates, join predicates, group by, and order by predicates. FROM and WHERE keywords are appropriately added in between. We always use AND to create a conjunction of multiple selection (or join) predicates.
- While adding a projection column to the query, we check if the column is present in the group by list. If so, then we add the column as it is to the projected columns without any further checks. Otherwise, we check if the projected column is associated with an aggregate operator such as MIN, MAX, SUM, COUNT, and AVG and accordingly project the column with (or without) its aggregate operator in the reconstructed SQL query.
- While creating join predicates within the SQL query, we include the left table, right table, left column, and the right column based on the predicted fragments. As for the arithmetic operator between the left and right columns, we always include the "=" operator as we do not explicitly predict the arithmetic operator for join predicates.



(a) Avg Test F1-Score (Borrow) (b) Borrow vs. Reconstruct (c) Top-3 vs. Top-1

Fig. 19. Predicted query vs. next query w.r.t. execution result tuples (80% train, 20% test).

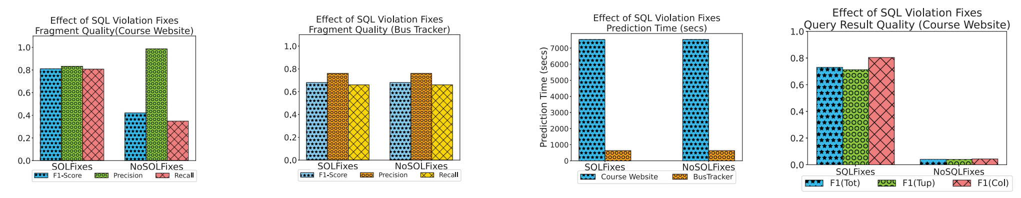
- The inclusion of the group by columns, order by columns, selection predicate columns and the arithmetic operators for selection predicates within the SQL query is straightforward as it involves copying the predicted fragments into the reconstructed SQL query.

As mentioned in Section 3.2.2, we represent the constants of all data types in selection predicates as 10 equi-depth range bins for each column and an additional "NULL" bin to accommodate IS (NOT) NULL clauses and out-of-range constants. Therefore, we apply the following heuristics in the same order to infer the selection predicates from the predicted bins. Although we do not explain the rationale behind these heuristics for space reasons, they are aimed at enhancing the recall without losing upon precision. Also, we do not include HAVING and LIMIT clauses in query reconstruction, but the inclusion of constants within those clauses can follow similar steps.

(1) If a "NULL" bin is predicted, then we include an "IS NOT NULL" or "IS NULL" predicate respectively into the query depending on whether the arithmetic operator is $\neq$ or something else.
(2) If the lower and upper bounds in the predicted bin are the same (possible for equi-depth range bins and also in cases where #distinct column values < 10), then we create the selection predicate from column name (ATTR), arithmetic operator (OP) and the bound (LOWER/UPPER).
(3) If the arithmetic operator (OP) is "=", then we create a conjunctive predicate as ATTR $\geq$ LOWER AND ATTR $\leq$ UPPER where LOWER and UPPER refer to the respective bounds in the bin.
(4) If OP is $\leq$ or $<$, then we create the predicate as ATTR OP UPPER.
(5) If OP is $\geq$ or $>$, then we create the predicate as ATTR OP LOWER.
(6) If OP is $\neq$, then we include a disjunctive predicate as ATTR $\leq$ LOWER OR ATTR $\geq$ UPPER.
(7) If OP is 'LIKE', then we include a disjunctive predicate as ATTR OP LOWER OR ATTR OP UPPER.

5.3.2 *Query Result Evaluation.* Figure 19 compares the results of the queries predicted by each of the ML algorithms w.r.t. the actual next queries upon the Course Website dataset. Following are the steps to compute the test F1-score.

- We compute the *column $F1$-score* referred to as F1(Col) based on the overlap between the columns projected in the result set of the actual and predicted query.
- We compute the *tuple $F1$-score* also called F1(Tup), based on the tuples that overlap between the predicted and actual query results only w.r.t. the matching columns.



(a) Fragment Quality (b) Fragment Quality (c) Prediction Latency (d) Result Quality

Fig. 20. Effect of SQL violation fixes on synthesis-based RNNs (80% train, 20% test).

- We compute the *total F1-score* or F1(Tot) as $\alpha\times$ F1(Col) + $(1-\alpha)\times$ F1(Tup). We set α to 0.2 in our experiments to give a higher weightage to F1(Tup). We present all the three F1-scores in most of the figures.

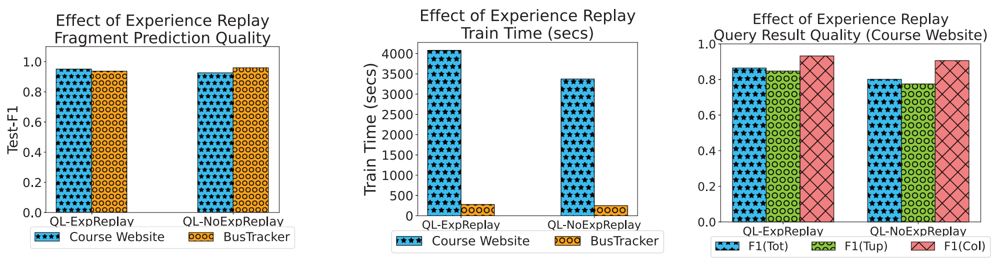
We also consider the following steps to handle special cases:

(1) If the predicted query is not of valid SQL syntax and cannot be executed, then the F1-score is 0.
(2) If only one of the predicted and the actual queries returns an empty result, then the F1-score is 0.
(3) If both the predicted and the actual queries return empty results, then F1(Col) is still computed based on the overlapping set of columns. If F1(Col) is non-zero, then F1(Tup) is 1.0 and F1(Tot) is computed as their weighted average. If the matching columns are empty, then F1(Tot) is 0.

As mentioned in Section 5.3.1, we create the predicted SQL query from the fragments in one of two ways - query borrowing or query reconstruction. Figure 19(b) reports a comparison between the total F1-scores, F1(Tot), obtained from borrowing a query and reconstructing a query. We can notice that borrowing outperforms reconstruction and this is evident from the fact that query reconstruction cannot obtain the exact constants in the selection predicates. However, surprisingly enough, Q-learning and RNN-Synth achieve high F1-scores of 0.86 and 0.72 from query borrowing. Also, we can notice that the relative performance among the ML algorithms is consistent across both borrowing and reconstruction. We report the breakdown of the total F1-scores in Figure 19(a), and we find that F1(Col) $\geq$ F1(Tup) and the fact that we give more weightage to F1(Tup) makes our reported F1(Tot) an under-estimate. In Figures 19(a) and (b), we report the F1-scores over the Top-3 predicted queries. However, it is likely that a few applications may choose to speculatively execute only the Top-1 predicted query and cache its results. Therefore, we compare the test F1-scores obtained from Top-3 predictions against those from Top-1 prediction, and we can notice from Figure 19(c), that Top-1 performs slightly worse as compared to Top-3 in the case of both Q-Learning and RNN. In the case of SVD-based collaborative filtering, the difference is significantly high.

5.3.3 *Effect of SQL Violation Fixes and Experience Replay.*

In this section, we evaluate two important enhancements that we apply to our adaptation of temporal predictors. The first one is the SQL violation fixes we proposed for synthesis-based RNNs in Section 4.2.2. The second one is experience replay from the Q-Learning literature [47, 50] that we described in Section 4.3. Figure 20 shows the effect of SQL violation fixes on both the datasets. These fixes are applied only during the prediction phase. Since these are constant-time fixes, there is no noticeable increase in latency upon using these fixes as compared to not using them (see Figure 20(c)). We can also notice that



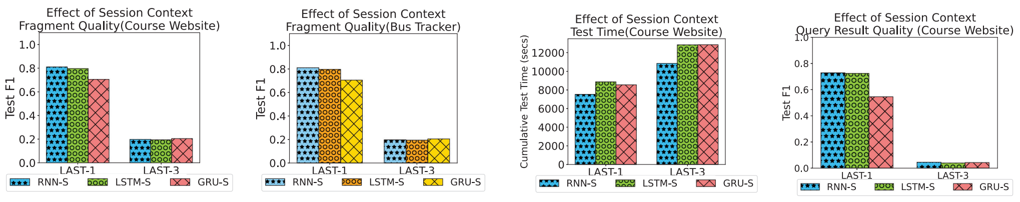
(a) F1-Score-Query Fragments (b) Training Latency (secs) (c) F1-Score-Query Results

Fig. 21. Effect of experience replay and random action selection on Q-learning (80% train, 20% test).

there is no significant benefit in using these SQL fixes for the Bus Tracker dataset (Figure 20(b)). This is because the number of distinct queries is as few as 25 for this dataset that makes the prediction task easier. If RNNs are sufficiently exposed to all possible query transitions during the training phase, then the weights of the RNNs are well refined and the RNN predictions require fewer fixes during the testing phase. In the case of Course Website dataset, we can observe that the recall in the case of query fragment prediction (Figure 20(a)) drops significantly without SQL violation fixes, but the precision is still high. This shows that the training data does not capture all possible transitions with a high enough frequency for the RNN weights of all the successor query fragments to exceed the pre-set threshold. This brings us back to the discussion of a reasonable threshold to set and also establishes that our method of applying SQL violation fixes is principled and generic enough that can enhance the recall without sacrificing precision significantly as shown in the figure. This can also be attributed to the fact that we use well-defined SQL-specific rules and always set the compatible fragments with the highest probability from the predicted output weight vector. In the case of result quality, the F1-scores are much higher while using SQL fixes as compared to not using them (Figure 20(d)). This explains the need for SQL fixes because, without them, several queries will miss essential query fragments that makes them invalid and non-executable. As mentioned in Section 5.3.2, if the predicted query cannot be executed, then the F1-score is 0. Note that despite not fixing the SQL violations, the fragment prediction F1-score of RNN-Synth for Course Website (Figure 20(a)) is greater than collaborative filtering and historical RNNs (see Figure 9(a) for comparison).

Figure 21 shows the effect of experience replay and random action selection on Exact Q-Learning. Although we cannot observe a noticeable difference in predicted query fragment quality (Figure 21(a)) for both the datasets, we can notice some decrease in the query result qualities upon not using experience replay from Figure 21(c), i.e., 0.84 vs. 0.77 for F1(Tup) and 0.86 vs. 0.8 for F1(Tot). Figure 21(b) shows that the usage of experience replay can increase the training time for larger datsets, which marks a latency vs. quality tradeoff. We sample as few as 100 transitions in each sparsity reduction iteration that can perhaps be increased, but that also increases the training time.

5.3.4 Effect of Session Context. To examine the effect of session context, we have implemented a variant of synthesis-based RNNs that concatenates the current query to the preceeding queries from the ongoing user session and feeds them to the RNN as an input to predict the next query. We confine our context representation to a total of three queries (including the current query) to save on training and test latency. We refer to this implementation as LAST-3 in Figure 22. In contrast, we term the default implementation that feeds <Input Query, Next Query> train and test pairs to the model as LAST-1. We can notice from Figure 22(a) and (b) that LAST-3 performs significantly worse



(a) Fragment Quality (b) Fragment Quality (c) Prediction Latency (d) Result Quality

Fig. 22. Effect of session context on synthesis-based RNNs (80% train, 20% test).

than LAST-1 w.r.t. fragment quality on both the datasets. Figure 22(d) shows that the query result F1-scores of LAST-3 on the Course Website dataset are close to 0. Interestingly, the implementation of synthesis-based RNNs, GRUs and LSTMs show a consistent result. Following are the reasons for this seemingly surprising result.

- The previous queries in a session do not have an equal influence on next query prediction. To formulate the context vector, we have to apply attention models [4] that can learn the weighted influence of earlier queries on the prediction.
- It has been noted in earlier works on attention models [4, 8] that increasing length in an input sequence does not necessarily produce better temporal prediction quality.
- In addition to the above reasons, for sequence prediction to work well with a longer input sequence, enough instances of <input sequence, next query> pairs should be seen by the temporal predictors to capture the effect of sequences. Our results show that <Current query, Next query> pairs are frequently observed as compared to long sequences and we can get competent performance from sequence models even without context on our datasets.

Also, we observed that the increase in prediction latencies by adding context information is greater than the increase in training latency. Therefore, we presented the prediction latencies alone in Figure 22(c) for space reasons. The implementation of attention models to capture the session context is beyond the scope of this article. To train attention models with minimum latencies and long sequences with several prior queries, we need to compress our SQL-aware embeddings and this compression strategy also needs to be reversible to re-generate the source embeddings. This is an interesting problem we plan to solve in a future work. In the case of Q-learning, we do not need to explicitly represent each *state* as a sequence with context, because the Q-table inherently captures the notion of a sequence within the Bellman equation (see Equation (1)). Q-values are long-term rewards defined recursively such that the value assigned to a transition from source to target query is dependent on the optimal look-ahead reward from the target to goal query.

5.4 Discussion

In this section, we discuss the significance of our findings and interesting directions for further research. The first half of the discussion is devoted to fragment embeddings while the second part focuses on the effectiveness of ML algorithms for next query prediction.

5.4.1 Effect of Embedding Creation. As we have mentioned in Section 3.2, we create the embedding vectors for the entire dataset comprising both the train and test queries in an offline step. This is because all the ML algorithms use the same embedding vectors and hence, the latency results we presented so far exclude the effect of embedding. In Table 5, we present the average time in seconds taken to generate the embedding for each query and the next query fragment prediction latency incurred by each of the ML algorithms. We also present the average execution latency

Table 5. Average Embedding Creation, Query Prediction, Execution, and Re-generation Latencies (s)

Dataset	**Avg. Embed Time(s)**	**Q-Learn Predict Time(s)**	**RNN-S Predict Time(s)**	**CF-SVD Predict Time(s)**	**CF-Cos Predict Time(s)**	**RNN-H Predict Time(s)**	**Avg. Exec. Time(s)**	**Avg. Borrow Time(s)**	**Avg. Reconstruct Time(s)**
Course Website	3.09	1.56	0.52	1.73	1.53	4.33	0.03	$1.18 \text{ x } 10^{-5}$	$2.72 \text{ x } 10^{-5}$
Bus Tracker	0.54	0.02	0.18	0.06	0.07	2.04	0.0002	N/A	N/A

per query along with the predicted query re-generation time to contrast it with the embedding creation and next query prediction times. The reason for this is that in real time, when a query is issued by an end user, an embedding vector is generated for the query and is passed to the next query predictor as shown in Figure 1. Simultaneously, the current query is executed by the database and the results are returned. The next query embedding is predicted, its SQL counterpart is re-generated, and the predicted SQL query passed back to the user along with the result tuple set from the current query execution. Therefore, we use Table 5 to check if the embedding creation, next query prediction, and re-generation latencies are at least partially covered by the execution time. We can notice that the query re-generation times are insignificant whether we follow the query borrowing or SQL reconstruction technique described in Section 5.3.1. Although we find that the execution time together with the query re-generation time is lesser than the sum of the embedding creation and next query prediction times, if we also consider the user think time, then we can realize the importance of next query prediction. This is because a human user always needs to observe and analyze the results of the current query to decide upon her next query to the database, which consumes a non-trivial amount of time. From Table 5, we can notice that the average embedding creation time increases as the schema gets complex. This is because the dimensionality of a query embedding that we generate is not dependent on the size of the database; rather, it is dependent on the size of the schema that primarily includes #tables and #columns per table. While the Course Website dataset consists of 113 tables and 839 columns in total, Bus Tracker consists of 95 tables and 770 columns. However, the dimensionality of Bus Tracker embedding is 9,155 bits, which is much lesser than 102,020 bits required for Course Website. The reason for such a huge difference is that the number of column pairs from various tables that can possibly participate in join predicates is 1,355 for Bus Tracker as against 92,045 for Course Website. This difference in schema complexity is also reflected in the embedding creation latencies. While it takes 3.09 s on an average to generate a query embedding for Course Website, it only requires 0.54 s for the Bus Tracker dataset.

The prediction latencies reported for various ML algorithms make use of intra-query parallelism. All the ML algorithms pre-build their models from 80% training data in an offline step. Except for RNN-Synth, each ML approach uses 48 processes to parallelize the next query prediction given the current test query. As we have explained, the sizes of the sampled sessions and the pre-built models are smaller for the Bus Tracker dataset. Also, given that the embeddings have lower dimensionality for Bus Tracker as compared to Course Website, the prediction phase for CF and Q-Learning benefits from parallelism and outperforms RNN-Synth. However, upon the larger Course Website dataset, we can clearly see the effect of constant time prediction that RNN-Synth is enabled with. It takes as less as 0.52 s per query to predict its next query embedding. RNN-H incurs the largest time, because it has to compute the entropy between the probabilistic output vector emitted by the RNN and every sampled query from the historical logs, to pick the top-K next query candidates with the least entropy. This is more expensive than the session level sampling that CF algorithms use to detect the closest session to the ongoing test session. For the Course

Website dataset, the best cumulative next query prediction latency comes from RNN-Synth (3.09 + 0.52 = 3.61 s), whereas for the Bus Tracker dataset, Q-Learning performs the best with 0.54 + 0.02 = 0.56 s. Note that we do not have any available data for Bus Tracker dataset that means that every query returns 0 tuples. Therefore, the average query execution time is as low as 0.0002 s for Bus Tracker as compared to Course Website. In reality we anticipate this time to be longer. The conclusion we draw from this analysis is that, full query prediction is worthwhile if the user think time is comparable to the embedding creation time.

To alleviate the embedding creation time, there are two possible solutions. One of them is to parallelize the embedding vector creation. In fact, our offline step of embedding creation partitions SQL queries among several threads all of which simultaneously create the embeddings via inter-query parallelism. Another solution is to avoid predicting the entire query. As we have discussed in Section 3.2.2, existing works such as Reference [23] that focus on join cardinality estimation only need to predict the join and selection predicates. Our fragment embedding is SQL-aware unlike existing SQL-agnostic numerical embedding libraries such as Word2Vec that exist for NLP. Thus, we can choose the operators to include in the embedding vectors and thereby selectively train the ML models on a subset of SQL operators that need to be predicted. This will not only reduce the embedding creation time but also speed up the training and test phases. However, this depends on the application task at hand that decides that operators ought to be predicted. For instance, prediction only upon the Bellwether fragments may be useful enough in several scenarios.

5.4.2 Effectiveness of ML Algorithms. Table 6 shows an illustrative scenario for both the datasets. Given an instance of the current SQL query, the next query predicted by each ML algorithm is shown in the table along with the test F1-score obtained upon comparing the predicted query with the ground truth that is the ideal successor to the current query. Note that these are real instances of experimental queries and the predicted queries and test F1 metrics in Table 6 are also real. From Table 6(a), we can notice that Q-Learn predicts the ideal successor including the selection predicate constant bin [674-888] that captures the expected constant (888). RNN-Synth narrowly misses the constant bin but it heavily loses out on predicting the projection list accurately. Of all the columns, it only predicts a single column, name, that brings its test F1-score down to 0.3. CF-based approaches totally predict the wrong queries but get a non-zero test F1 for getting the DML type right. On similar lines, both Q-Learning and RNN-Synth predict the next query accurately for the current query example from the Bus Tracker dataset (Table 6(b)). Note that the selection predicates for the Bus Tracker queries only contain the columns but not the constants. As we discussed in Section 3.2.2, constant prediction is excluded for Bus Tracker due to the lack of data. Also, both CF-SVD and RNN-H perform better on Bus Tracker than Course Website. This is because Bus Tracker has a more concise vocabulary that makes it easier for sampling to capture a lot of the distinct queries. CF-Cos, however, consistently performs poorly, because it relies on two-level sampling and identifying the similar sessions from training data alone is not enough. It also needs to match the ongoing test session to the sampled queries within the sessions to fetch the top-K queries, unlike CF-SVD that only needs to sample sessions. Likewise, RNN-H only needs sampling upon queries alone but not on sessions. From our experiments, we learned the following lessons that contributed to the effectiveness of ML algorithms for next query prediction.

- For ML algorithms that rely on historical query prediction, we created and stored the query vocabulary effectively. Our experiments show that in real-world datasets, the distinct queries that form the vocabulary are actually fewer as compared to the sizes of the query

Table 6. Predicted Next Queries for an Example Current Query

(a) **Course Website**

Approach	**Predicted Next Query based on Fragments**
Q-Learn	SELECT * FROM jos_community_courses WHERE id = [674 - 888] **(F1=1.0)**
RNN-S	SELECT name FROM jos_community_courses WHERE id = [889 - 1360] **(F1=0.3)**
CF-SVD	SELECT COUNT(userid) FROM jos_community_usefulness WHERE resourceid = [3348 - 3684] **(F1=0.05)**
CF-Cos	SELECT rsource_type FROM resource WHERE gid = [3510 - 3883] **(F1=0.06)**
RNN-H	SELECT MAX(resource.gid) FROM resource **(F1=0.06)**

SELECT rsource_type FROM resource WHERE gid = 888 (Current Query)
*SELECT * FROM jos_community_courses WHERE id = 888* (Next Query)

(b) **Bus Tracker**

Approach	**Predicted Next Query based on Fragments**
Q-Learn	SELECT agency_id FROM m_agency WHERE {agency_id, valid_now} **(F1=1.0)**
RNN-S	SELECT agency_id FROM m_agency WHERE {agency_id, valid_now} **(F1=1.0)**
CF-SVD	SELECT user_id FROM m_agency WHERE {agency_id, valid_now} **(F1=0.8)**
CF-Cos	SELECT COUNT(*) FROM dv_notes_message WHERE {user_id, agency_id, notice_id, route_id} **(F1=0.03)**
RNN-H	SELECT agency_timezone FROM m_agency WHERE {agency_id} **(F1=0.44)**

SELECT COUNT(*) FROM dv_notes_message WHERE {user_id, agency_id, notice_id, route_id} (Current Query)
SELECT agency_id FROM m_agency WHERE {agency_id, valid_now} (Next Query)

logs. Optimizations such as SHA-256 derived a concise representation for the embeddings and made the query vocabulary creation easier, also saving on computational complexity.

- To sample more and still save on prediction latencies, we used CPU-bound parallelism.
- Since RNNs predicted occurrence probabilities for each fragment, we used synthesis instead of relying on historical query logs. This not only brought about constant prediction time but also enabled a fine-grained control on the quality of the next query. Depending on whether we wanted to bias on enhancing the precision or recall, we could fine-tune the query correction mechanism. We also observed that even single threaded implementation of synthesis-based RNNs outperformed heavily parallelized ML algorithms in next query prediction. Such benefits became more obvious when the experiments were conducted on the larger query workload from Course Website dataset.
- Exact Q-Learning emerged as the overall winner on quality, cumulative train and test latency as well as the memory requirement. Before going for advanced variants such as deep Q-Learning that are in fact approximations, it is important that database practitioners validate the applicability of Exact Q-Learning to the research problem at hand. A surprising thing we learned is that the materialized Q-Table was much smaller than the RNN models created by out-of-the-box ML libraries such as TensorFlow.

There are several applications that can benefit from query prediction. As a future work, we intend to exploit query workload prediction to data re-organization and buffer page management in databases. We plan to extend this work to build self-managed or autonomous databases in which we can automate every component of the DBMS ranging from query optimizer, indexer to memory manager. As we have mentioned in Section 1, query fragment prediction can help in preemptive execution of partially or fully created query plans and can also help in configuring #threads for speculative parallelism of SQL operators in future user-issued queries.

6 CONCLUSION

In this article, we adapt and evaluate several ML algorithms for next query prediction during a human-database interaction session. As a part of our adaptation, we implement session cleaning heuristics, query fragment embedding vector creation, concise representations of query vocabulary and inter-and intra-query parallelism for predicting the next query completely from historical query logs. We also propose synthesis-based RNNs that can achieve constant prediction times without relying on historical queries. We propose two kinds of experiments to evaluate the ML algorithms both in an offline training scenario and an online test-then-train streaming queries scenario. Based on an exhaustive evaluation on two real-world datasets, we find that among all the ML algorithms that we implement, exact Q-Learning in conjunction with numerical reward function is the best performing algorithm on test F1-scores, cumulative train and test latencies and memory consumption. We observe that next query prediction benefits from temporal predictors over adaptations of state-of-the-art recommender systems.

REFERENCES

[1] 2011. *JSQLParser*. Retrieved from https://github.com/JSQLParser/JSqlParser.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Retrieved from http://tensorflow.org/.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. 29–42.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*.

[5] Ugur Çetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alexander Kalinin, Olga Papaemmanouil, and Stanley B. Zdonik. 2013. Query steering for interactive data exploration. In *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR'13)*.

[6] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. 2009. Query recommendations for interactive database exploration. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM'09)*. 3–18.

[7] Surajit Chaudhuri and Raghav Kaushik. 2009. Extending autocompletion to tolerate errors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 707–718.

[8] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST@EMNLP'14)*. 103–111.

[9] François Chollet. 2015. keras. Retrieved from https://keras.io/.

[10] Dong Deng, Guoliang Li, He Wen, H. V. Jagadish, and Jianhua Feng. 2016. META: An efficient matching-based method for error-tolerant autocompletion. *Proc. VLDB Endow.* 9, 10 (2016), 828–839.

[11] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. 517–528.

[12] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2016. AIDE: An active learning-based approach for interactive data exploration. *IEEE Trans. Knowl. Data Eng.* 28, 11 (2016), 2842–2856.

[13] Magdalini Eirinaki, Suju Abraham, Neoklis Polyzotis, and Naushin Shaikh. 2014. QueRIE: Collaborative database exploration. *IEEE Trans. Knowl. Data Eng.* 26, 7 (2014), 1778–1790.

[14] Magdalini Eirinaki and Sweta Patel. 2015. QueRIE reloaded: Using matrix factorization to improve database query recommendations. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data'15)*. 1500–1508.

[15] Ori Bar El, Tova Milo, and Amit Somech. 2020. Automatically generating data exploration sessions using deep reinforcement learning. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)*. 1527–1537.

[16] Antonio Giuzio, Giansalvatore Mecca, Elisa Quintarelli, Manuel Roveri, Donatello Santoro, and Letizia Tanca. 2017. *INDIANA the Database Explorer*. Technical Report. Università della Basilicata, Politecnico di Milano.

[17] Bill G. Horne and Don R. Hush. 1996. Bounds on the complexity of recurrent neural network implementations of finite state machines. *Neural Netw.* 9, 2 (Mar. 1996), 243–252. DOI:https://doi.org/10.1016/0893-6080(95)00095-X

[18] Prasanth Jayachandran, Karthik Tunga, Niranjan Kamat, and Arnab Nandi. 2014. Combining user interaction, speculative query execution and sampling in the DICE system. *Proc. VLDB* 7, 13 (2014), 1697–1700.

[19] Manas Joglekar, Hector Garcia-Molina, and Aditya G. Parameswaran. 2016. Interactive data exploration with smart drill-down. In *Proceedings of the 32nd IEEE International Conference on Data Engineering (ICDE'16)*. 906–917.

[20] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *Proceedings of the IEEE 30th International Conference on Data Engineering, Chicago (ICDE'14)*. 472–483.

[21] Andrej Karpathy. 2015. The Unreasonable Effectiveness of Recurrent Neural Networks. Retrieved from http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[22] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. 2010. SnipSuggest: Context-aware autocompletion for SQL. *Proc. VLDB Endow.* 4, 1 (2010), 22–33. DOI:https://doi.org/10.14778/1880172.1880175

[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*.

[24] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. arxiv:1808.03196. Retrieved from https://arxiv.org/abs/1808.03196.

[25] Daniel D. Lee and H. Sebastian Seung. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature* 401, 6755 (1999), 788–791.

[26] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. 2014. Opportunistic physical design for big data analytics. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. 851–862.

[27] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2118–2130.

[28] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free control for distributed stream data processing using deep reinforcement learning. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 705–718.

[29] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. 2019. Opportunistic view materialization with deep reinforcement learning. arxiv:1903.01363. Retrieved from https://arxiv.org/abs/1903.01363.

[30] 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 (SIGMOD'18)*. 631–645.

[31] Ben McCamish, Vahid Ghadakchi, Arash Termehchy, Behrouz Touri, and Liang Huang. 2018. The data interaction game. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. 83–98.

[32] Venkata Vamsikrishna Meduri, Kanchan Chowdhury, and Mohamed Sarwat. 2019. Recurrent neural networks for dynamic user intent prediction in human-database interaction. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT'19)*. 654–657.

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS'13)*. Curran Associates, Inc., 3111–3119.

[34] Tova Milo and Amit Somech. 2018. Next-step suggestions for modern interactive data analysis platforms. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*. 576–585.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[36] Christopher Olah. 2015. Understanding LSTM-based RNNs. Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[37] Inc. Open Source Matters and the Joomla community. 2005. Joomla! Retrieved from https://www.joomla.org/.

[38] Olga Papaemmanouil, Yanlei Diao, Kyriaki Dimitriadou, and Liping Peng. 2016. Interactive data exploration via machine learning models. *IEEE Data Eng. Bull.* 39, 4 (2016), 38–49.

[39] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. 587–602.

[40] Liping Peng, Enhui Huang, Yuqing Xing, Anna Liu, and Yanlei Diao. 2017. Uncertainty Sampling and Optimization for Interactive Database Exploration. *UMass Technical Report* (2017).

[41] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*.

[42] Senjuti Basu Roy, Haidong Wang, Ullas Nambiar, Gautam Das, and Mukesh K. Mohania. 2009. DynaCet: Building dynamic faceted search systems over databases. In *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 1463–1466.

[43] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2nd ed.). Pearson Education.

[44] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized experience replay. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*.

[45] Vik Singh, Jim Gray, Ani Thakar, Alexander S. Szalay, Jordan Raddick, Bill Boroski, Svetlana Lebedeva, and Brian Yanny. 2007. SkyServer traffic report—The first five years. arxiv:cs/0701173. Retrieved from https://arxiv.org/abs/cs/0701173.

[46] Amit Somech, Tova Milo, and Chai Ozeri. 2019. Predicting "What is Interesting" by mining interactive-data-analysis session logs. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT'19)*. 456–467.

[47] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2 ed.). The MIT Press.

[48] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya G. Parameswaran, and Neoklis Polyzotis. 2015. SEEDB: Efficient data-driven visualization recommendations to support visual analytics. *Proc. VLDB* 8, 13 (2015), 2182–2193.

[49] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. 557–572.

[50] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. In *Machine Learning*. 279–292.

[51] Michael Wunder, Michael L. Littman, and Monica Babes. 2010. Classes of multiagent Q-learning dynamics with epsilon-greedy exploration. In *Proceedings of the International Conference on Machine Learning (ICML'10)*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 1167–1174.

[52] Cong Yan and Yeye He. 2020. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)*. 1539–1554.

[53] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Russ R. Salakhutdinov, and Yoshua Bengio. 2016. Architectural complexity measures of recurrent neural networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 1822–1830.

Received February 2020; revised September 2020; accepted December 2020